GNU/Linux

THE MAN-PAGES BOOK

Maintainers:

Alejandro Colomar <alx@kernel.org> 2020 - present (5.09 - HEAD)
Michael Kerrisk <mtk.manpages@gmail.com> 2004 - 2021 (2.00 - 5.13)
Andries Brouwer <aeb@cwi.nl> 1995 - 2004 (1.6 - 1.70)
Rik Faith 1993 - 1995 (1.0 - 1.5)

NAME

intro − introduction to user commands

DESCRIPTION

Section 1 of the manual describes user commands and tools, for example, file manipulation tools, shells, compilers, web browsers, file and image viewers and editors, and so on.

NOTES

Linux is a flavor of UNIX, and as a first approximation all user commands under UNIX work precisely the same under Linux (and FreeBSD and lots of other UNIX-like systems).

Under Linux, there are GUIs (graphical user interfaces), where you can point and click and drag, and hopefully get work done without first reading lots of documentation. The traditional UNIX environment is a CLI (command line interface), where you type commands to tell the computer what to do. That is faster and more powerful, but requires finding out what the commands are. Below a bare minimum, to get started.

Login

In order to start working, you probably first have to open a session by giving your username and password. The program *login*(1) now starts a *shell* (command interpreter) for you. In case of a graphical login, you get a screen with menus or icons and a mouse click will start a shell in a window. See also *xterm*(1)

The shell

One types commands to the *shell*, the command interpreter. It is not built-in, but is just a program and you can change your shell. Everybody has their own favorite one. The standard one is called *sh*. See also *ash*(1), *bash*(1), *chsh*(1), *csh*(1), *dash*(1), *ksh*(1), *zsh*(1)

A session might go like:

```
knuth login: aeb
Password: ********
$ date
Tue Aug  6 23:50:44 CEST 2002
$ cal
     August 2002
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

$ ls
bin  tel
$ ls -l
total 2
drwxrwxr-x   2 aeb       1024 Aug  6 23:51 bin
-rw-rw-r--   1 aeb         37 Aug  6 23:52 tel
$ cat tel
maja    0501-1136285
peter   0136-7399214
$ cp tel tel2
$ ls -l
total 3
drwxr-xr-x   2 aeb       1024 Aug  6 23:51 bin
-rw-r--r--   1 aeb         37 Aug  6 23:52 tel
-rw-r--r--   1 aeb         37 Aug  6 23:53 tel2
$ mv tel tel1
$ ls -l
total 3
```

```
drwxr-xr-x   2 aeb       1024 Aug  6 23:51 bin
-rw-r--r--   1 aeb         37 Aug  6 23:52 tel1
-rw-r--r--   1 aeb         37 Aug  6 23:53 tel2
$ diff tel1 tel2
$ rm tel1
$ grep maja tel2
maja    0501-1136285
$
```

Here typing Control-D ended the session.

The **$** here was the command prompt—it is the shell's way of indicating that it is ready for the next command. The prompt can be customized in lots of ways, and one might include stuff like username, machine name, current directory, time, and so on. An assignment PS1="What next, master? " would change the prompt as indicated.

We see that there are commands *date* (that gives date and time), and *cal* (that gives a calendar).

The command *ls* lists the contents of the current directory—it tells you what files you have. With a *−l* option it gives a long listing, that includes the owner and size and date of the file, and the permissions people have for reading and/or changing the file. For example, the file "tel" here is 37 bytes long, owned by aeb and the owner can read and write it, others can only read it. Owner and permissions can be changed by the commands *chown* and *chmod*.

The command *cat* will show the contents of a file. (The name is from "concatenate and print": all files given as parameters are concatenated and sent to "standard output" (see *stdout*(3)), here the terminal screen.)

The command *cp* (from "copy") will copy a file.

The command *mv* (from "move"), on the other hand, only renames it.

The command *diff* lists the differences between two files. Here there was no output because there were no differences.

The command *rm* (from "remove") deletes the file, and be careful! it is gone. No wastepaper basket or anything. Deleted means lost.

The command *grep* (from "g/re/p") finds occurrences of a string in one or more files. Here it finds Maja's telephone number.

Pathnames and the current directory

Files live in a large tree, the file hierarchy. Each has a *pathname* describing the path from the root of the tree (which is called */*) to the file. For example, such a full pathname might be */home/aeb/tel*. Always using full pathnames would be inconvenient, and the name of a file in the current directory may be abbreviated by giving only the last component. That is why */home/aeb/tel* can be abbreviated to *tel* when the current directory is */home/aeb*.

The command *pwd* prints the current directory.

The command *cd* changes the current directory.

Try alternatively *cd* and *pwd* commands and explore *cd* usage: "cd", "cd .", "cd ..", "cd /", and "cd ~".

Directories

The command *mkdir* makes a new directory.

The command *rmdir* removes a directory if it is empty, and complains otherwise.

The command *find* (with a rather baroque syntax) will find files with given name or other properties. For example, "find . −name tel" would find the file *tel* starting in the present directory (which is called .). And "find / −name tel" would do the same, but starting at the root of the tree. Large searches on a multi-GB disk will be time-consuming, and it may be better to use *locate*(1)

Disks and filesystems

The command *mount* will attach the filesystem found on some disk (or floppy, or CDROM or so) to the big filesystem hierarchy. And *umount* detaches it again. The command *df* will tell you how much of your disk is still free.

Processes

On a UNIX system many user and system processes run simultaneously. The one you are talking to runs in the *foreground*, the others in the *background*. The command *ps* will show you which processes are active and what numbers these processes have. The command *kill* allows you to get rid of them. Without option this is a friendly request: please go away. And "kill −9" followed by the number of the process is an immediate kill. Foreground processes can often be killed by typing Control-C.

Getting information

There are thousands of commands, each with many options. Traditionally commands are documented on *man pages*, (like this one), so that the command "man kill" will document the use of the command "kill" (and "man man" document the command "man"). The program *man* sends the text through some *pager*, usually *less*. Hit the space bar to get the next page, hit q to quit.

In documentation it is customary to refer to man pages by giving the name and section number, as in *man*(1)Man pages are terse, and allow you to find quickly some forgotten detail. For newcomers an introductory text with more examples and explanations is useful.

A lot of GNU/FSF software is provided with info files. Type "info info" for an introduction on the use of the program *info*.

Special topics are often treated in HOWTOs. Look in */usr/share/doc/howto/en* and use a browser if you find HTML files there.

SEE ALSO

ash(1), *bash*(1), *chsh*(1), *csh*(1), *dash*(1), *ksh*(1), *locate*(1), *login*(1), *man*(1), *xterm*(1), *zsh*(1), *wait*(2), *stdout*(3), *man-pages*(7), *standards*(7)

NAME

getent − get entries from Name Service Switch libraries

SYNOPSIS

getent [*option***]...** *database key***...**

DESCRIPTION

The **getent** command displays entries from databases supported by the Name Service Switch libraries, which are configured in */etc/nsswitch.conf*. If one or more *key* arguments are provided, then only the entries that match the supplied keys will be displayed. Otherwise, if no *key* is provided, all entries will be displayed (unless the database does not support enumeration).

The *database* may be any of those supported by the GNU C Library, listed below:

ahosts When no *key* is provided, use *sethostent*(3), *gethostent*(3), and *endhostent*(3) to enumerate the hosts database. This is identical to using *hosts*(5). When one or more *key* arguments are provided, pass each *key* in succession to *getaddrinfo*(3) with the address family **AF_UNSPEC**, enumerating each socket address structure returned.

ahostsv4
Same as **ahosts**, but use the address family **AF_INET**.

ahostsv6
Same as **ahosts**, but use the address family **AF_INET6**. The call to *getaddrinfo*(3) in this case includes the **AI_V4MAPPED** flag.

aliases When no *key* is provided, use *setaliasent*(3), *getaliasent*(3), and *endaliasent*(3) to enumerate the aliases database. When one or more *key* arguments are provided, pass each *key* in succession to *getaliasbyname*(3) and display the result.

ethers When one or more *key* arguments are provided, pass each *key* in succession to *ether_aton*(3) and *ether_hostton*(3) until a result is obtained, and display the result. Enumeration is not supported on **ethers**, so a *key* must be provided.

group When no *key* is provided, use *setgrent*(3), *getgrent*(3), and *endgrent*(3) to enumerate the group database. When one or more *key* arguments are provided, pass each numeric *key* to *getgrgid*(3) and each nonnumeric *key* to *getgrnam*(3) and display the result.

gshadow
When no *key* is provided, use *setsgent*(3), *getsgent*(3), and *endsgent*(3) to enumerate the gshadow database. When one or more *key* arguments are provided, pass each *key* in succession to *getsgnam*(3) and display the result.

hosts When no *key* is provided, use *sethostent*(3), *gethostent*(3), and *endhostent*(3) to enumerate the hosts database. When one or more *key* arguments are provided, pass each *key* to *gethostbyaddr*(3) or *gethostbyname2*(3), depending on whether a call to *inet_pton*(3) indicates that the *key* is an IPv6 or IPv4 address or not, and display the result.

initgroups
When one or more *key* arguments are provided, pass each *key* in succession to *getgrouplist*(3) and display the result. Enumeration is not supported on **initgroups**, so a *key* must be provided.

netgroup
When one *key* is provided, pass the *key* to *setnetgrent*(3) and, using *getnetgrent*(3) display the resulting string triple (*hostname*, *username*, *domainname*). Alternatively, three *keys* may be provided, which are interpreted as the *hostname*, *username*, and *domainname* to match to a netgroup name via *innetgr*(3). Enumeration is not supported on **netgroup**, so either one or three *keys* must be provided.

networks
When no *key* is provided, use *setnetent*(3), *getnetent*(3), and *endnetent*(3) to enumerate the networks database. When one or more *key* arguments are provided, pass each numeric *key* to *getnetbyaddr*(3) and each nonnumeric *key* to *getnetbyname*(3) and display the result.

passwd
When no *key* is provided, use *setpwent*(3), *getpwent*(3), and *endpwent*(3) to enumerate the passwd database. When one or more *key* arguments are provided, pass each numeric *key* to

getpwuid(3) and each nonnumeric *key* to *getpwnam*(3) and display the result.

protocols
When no *key* is provided, use *setprotoent*(3), *getprotoent*(3), and *endprotoent*(3) to enumerate the protocols database. When one or more *key* arguments are provided, pass each numeric *key* to *getprotobynumber*(3) and each nonnumeric *key* to *getprotobyname*(3) and display the result.

rpc When no *key* is provided, use *setrpcent*(3), *getrpcent*(3), and *endrpcent*(3) to enumerate the rpc database. When one or more *key* arguments are provided, pass each numeric *key* to *getrpcbynumber*(3) and each nonnumeric *key* to *getrpcbyname*(3) and display the result.

services
When no *key* is provided, use *setservent*(3), *getservent*(3), and *endservent*(3) to enumerate the services database. When one or more *key* arguments are provided, pass each numeric *key* to *getservbynumber*(3) and each nonnumeric *key* to *getservbyname*(3) and display the result.

shadow
When no *key* is provided, use *setspent*(3), *getspent*(3), and *endspent*(3) to enumerate the shadow database. When one or more *key* arguments are provided, pass each *key* in succession to *getspnam*(3) and display the result.

OPTIONS

−−service *service*
−s *service*
Override all databases with the specified service. (Since glibc 2.2.5.)

−−service *database*:*service*
−s *database*:*service*
Override only specified databases with the specified service. The option may be used multiple times, but only the last service for each database will be used. (Since glibc 2.4.)

−−no−idn
−i Disables IDN encoding in lookups for **ahosts**/**getaddrinfo**(3) (Since glibc-2.13.)

−−help
−? Print a usage summary and exit.

−−usage
Print a short usage summary and exit.

−−version
−V Print the version number, license, and disclaimer of warranty for **getent**.

EXIT STATUS

One of the following exit values can be returned by **getent**:

0 Command completed successfully.

1 Missing arguments, or *database* unknown.

2 One or more supplied *key* could not be found in the *database*.

3 Enumeration not supported on this *database*.

SEE ALSO

nsswitch.conf(5)

NAME

iconv − convert text from one character encoding to another

SYNOPSIS

iconv [*options*] [−f *from-encoding*] [−t *to-encoding*] [*inputfile*]...

DESCRIPTION

The **iconv** program reads in text in one encoding and outputs the text in another encoding. If no input files are given, or if it is given as a dash (−), **iconv** reads from standard input. If no output file is given, **iconv** writes to standard output.

If no *from-encoding* is given, the default is derived from the current locale's character encoding. If no *to-encoding* is given, the default is derived from the current locale's character encoding.

OPTIONS

−−from−code= *from-encoding*
−f *from-encoding*
Use *from-encoding* for input characters.

−−to−code=*to-encoding*
−t *to-encoding*
Use *to-encoding* for output characters.

If the string **//IGNORE** is appended to *to-encoding*, characters that cannot be converted are discarded and an error is printed after conversion.

If the string **//TRANSLIT** is appended to *to-encoding*, characters being converted are transliterated when needed and possible. This means that when a character cannot be represented in the target character set, it can be approximated through one or several similar looking characters. Characters that are outside of the target character set and cannot be transliterated are replaced with a question mark (?) in the output.

−−list
−l List all known character set encodings.

−c Silently discard characters that cannot be converted instead of terminating when encountering such characters.

−−output=*outputfile*
−o *outputfile*
Use *outputfile* for output.

−−silent
−s This option is ignored; it is provided only for compatibility.

−−verbose
Print progress information on standard error when processing multiple files.

−−help
−? Print a usage summary and exit.

−−usage
Print a short usage summary and exit.

−−version
−V Print the version number, license, and disclaimer of warranty for **iconv**.

EXIT STATUS

Zero on success, nonzero on errors.

ENVIRONMENT

Internally, the **iconv** program uses the *iconv*(3) function which in turn uses *gconv* modules (dynamically loaded shared libraries) to convert to and from a character set. Before calling *iconv*(3), the **iconv** program must first allocate a conversion descriptor using *iconv_open*(3). The operation of the latter function is influenced by the setting of the **GCONV_PATH** environment variable:

- If **GCONV_PATH** is not set, *iconv_open*(3) loads the system gconv module configuration cache file created by *iconvconfig*(8) and then, based on the configuration, loads the gconv modules needed to perform the conversion. If the system gconv module configuration cache file is not available then

the system gconv module configuration file is used.

- If **GCONV_PATH** is defined (as a colon-separated list of pathnames), the system gconv module configuration cache is not used. Instead, *iconv_open*(3) first tries to load the configuration files by searching the directories in **GCONV_PATH** in order, followed by the system default gconv module configuration file. If a directory does not contain a gconv module configuration file, any gconv modules that it may contain are ignored. If a directory contains a gconv module configuration file and it is determined that a module needed for this conversion is available in the directory, then the needed module is loaded from that directory, the order being such that the first suitable module found in **GCONV_PATH** is used. This allows users to use custom modules and even replace system-provided modules by providing such modules in **GCONV_PATH** directories.

FILES

/usr/lib/gconv
: Usual default gconv module path.

/usr/lib/gconv/gconv-modules
: Usual system default gconv module configuration file.

/usr/lib/gconv/gconv-modules.cache
: Usual system gconv module configuration cache.

Depending on the architecture, the above files may instead be located at directories with the path prefix */usr/lib64*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

Convert text from the ISO/IEC 8859-15 character encoding to UTF-8:

```
$ iconv -f ISO-8859-15 -t UTF-8 < input.txt > output.txt
```

The next example converts from UTF-8 to ASCII, transliterating when possible:

```
$ echo abc ß α € àḃç | iconv -f UTF-8 -t ASCII//TRANSLIT
abc ss ? EUR abc
```

SEE ALSO

locale(1), *uconv*(1), *iconv*(3), *nl_langinfo*(3), *charsets*(7), *iconvconfig*(8)

NAME

ldd − print shared object dependencies

SYNOPSIS

ldd [*option*]... *file*...

DESCRIPTION

ldd prints the shared objects (shared libraries) required by each program or shared object specified on the command line. An example of its use and output is the following:

```
$ ldd /bin/ls
    linux-vdso.so.1 (0x00007ffcc3563000)
    libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
    libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
    libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
    /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
    libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

In the usual case, **ldd** invokes the standard dynamic linker (see *ld.so*(8)) with the **LD_TRACE_LOADED_OBJECTS** environment variable set to 1. This causes the dynamic linker to inspect the program's dynamic dependencies, and find (according to the rules described in *ld.so*(8)) and load the objects that satisfy those dependencies. For each dependency, **ldd** displays the location of the matching object and the (hexadecimal) address at which it is loaded. (The *linux-vdso* and *ld-linux* shared dependencies are special; see *vdso*(7) and *ld.so*(8).)

Security

Be aware that in some circumstances (e.g., where the program specifies an ELF interpreter other than *ld-linux.so*), some versions of **ldd** may attempt to obtain the dependency information by attempting to directly execute the program, which may lead to the execution of whatever code is defined in the program's ELF interpreter, and perhaps to execution of the program itself. (Before glibc 2.27, the upstream **ldd** implementation did this for example, although most distributions provided a modified version that did not.)

Thus, you should *never* employ **ldd** on an untrusted executable, since this may result in the execution of arbitrary code. A safer alternative when dealing with untrusted executables is:

```
$ objdump -p /path/to/program | grep NEEDED
```

Note, however, that this alternative shows only the direct dependencies of the executable, while **ldd** shows the entire dependency tree of the executable.

OPTIONS

--version
: Print the version number of **ldd**.

--verbose
-v Print all information, including, for example, symbol versioning information.

--unused
-u Print unused direct dependencies. (Since glibc 2.3.4.)

--data-relocs
-d Perform relocations and report any missing objects (ELF only).

--function-relocs
-r Perform relocations for both data objects and functions, and report any missing objects or functions (ELF only).

--help Usage information.

BUGS

ldd does not work on a.out shared libraries.

ldd does not work with some extremely old a.out programs which were built before **ldd** support was added to the compiler releases. If you use **ldd** on one of these programs, the program will attempt to

run with *argc* = 0 and the results will be unpredictable.

SEE ALSO

pldd(1), *sprof*(1), *ld.so*(8), *ldconfig*(8)

NAME

locale − get locale-specific information

SYNOPSIS

locale [*option*]
locale [*option*] **-a**
locale [*option*] **-m**
locale [*option*] *name*...

DESCRIPTION

The **locale** command displays information about the current locale, or all locales, on standard output.

When invoked without arguments, **locale** displays the current locale settings for each locale category (see *locale*(5)), based on the settings of the environment variables that control the locale (see *locale*(7)). Values for variables set in the environment are printed without double quotes, implied values are printed with double quotes.

If either the **−a** or the **−m** option (or one of their long-format equivalents) is specified, the behavior is as follows:

−−all−locales
−a Display a list of all available locales. The **−v** option causes the **LC_IDENTIFICATION** metadata about each locale to be included in the output.

−−charmaps
−m Display the available charmaps (character set description files). To display the current character set for the locale, use **locale −c charmap**.

The **locale** command can also be provided with one or more arguments, which are the names of locale keywords (for example, *date_fmt*, *ctype−class−names*, *yesexpr*, or *decimal_point*) or locale categories (for example, **LC_CTYPE** or **LC_TIME**). For each argument, the following is displayed:

- For a locale keyword, the value of that keyword to be displayed.
- For a locale category, the values of all keywords in that category are displayed.

When arguments are supplied, the following options are meaningful:

−−category−name
−c For a category name argument, write the name of the locale category on a separate line preceding the list of keyword values for that category.

For a keyword name argument, write the name of the locale category for this keyword on a separate line preceding the keyword value.

This option improves readability when multiple name arguments are specified. It can be combined with the **−k** option.

−−keyword−name
−k For each keyword whose value is being displayed, include also the name of that keyword, so that the output has the format:

```
keyword="value"
```

The **locale** command also knows about the following options:

−−verbose
−v Display additional information for some command-line option and argument combinations.

−−help
−? Display a summary of command-line options and arguments and exit.

−−usage
Display a short usage message and exit.

−−version
−V Display the program version and exit.

FILES

/usr/lib/locale/locale-archive
Usual default locale archive location.

/usr/share/i18n/locales
Usual default path for locale definition files.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

```
$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=

$ locale date_fmt
%a %b %e %H:%M:%S %Z %Y

$ locale -k date_fmt
date_fmt="%a %b %e %H:%M:%S %Z %Y"

$ locale -ck date_fmt
LC_TIME
date_fmt="%a %b %e %H:%M:%S %Z %Y"

$ locale LC_TELEPHONE
+%c (%a) %l
(%a) %l
11
1
UTF-8

$ locale -k LC_TELEPHONE
tel_int_fmt="+%c (%a) %l"
tel_dom_fmt="(%a) %l"
int_select="11"
int_prefix="1"
telephone-codeset="UTF-8"
```

The following example compiles a custom locale from the *./wrk* directory with the *localedef*(1) utility under the *$HOME/.locale* directory, then tests the result with the *date*(1) command, and then sets the environment variables **LOCPATH** and **LANG** in the shell profile file so that the custom locale will be used in the subsequent user sessions:

```
$ mkdir -p $HOME/.locale
$ I18NPATH=./wrk/ localedef -f UTF-8 -i fi_SE $HOME/.locale/fi_SE.UTF-8
$ LOCPATH=$HOME/.locale LC_ALL=fi_SE.UTF-8 date
$ echo "export LOCPATH=\$HOME/.locale" >> $HOME/.bashrc
```

```
$ echo "export LANG=fi_SE.UTF-8" >> $HOME/.bashrc
```

SEE ALSO

localedef(1), *charmap*(5), *locale*(5), *locale*(7)

NAME

localedef − compile locale definition files

SYNOPSIS

localedef [*options*] *outputpath*

localedef −−add−to−archive [*options*] *compiledpath*

localedef −−delete−from−archive [*options*] *localename ...*

localedef −−list−archive [*options*]

localedef −−help

localedef −−usage

localedef −−version

DESCRIPTION

The **localedef** program reads the indicated *charmap* and *input* files, compiles them to a binary form quickly usable by the locale functions in the C library (**setlocale**(3), *localeconv*(3), etc.), and places the output in *outputpath*.

The *outputpath* argument is interpreted as follows:

- If *outputpath* contains a slash character ('/'), it is interpreted as the name of the directory where the output definitions are to be stored. In this case, there is a separate output file for each locale category (*LC_TIME*, *LC_NUMERIC*, and so on).
- If the **−−no−archive** option is used, *outputpath* is the name of a subdirectory in */usr/lib/locale* where per-category compiled files are placed.
- Otherwise, *outputpath* is the name of a locale and the compiled locale data is added to the archive file */usr/lib/locale/locale−archive*. A locale archive is a memory-mapped file which contains all the system-provided locales; it is used by all localized programs when the environment variable **LOCPATH** is not set.

In any case, **localedef** aborts if the directory in which it tries to write locale files has not already been created.

If no *charmapfile* is given, the value *ANSI_X3.4−1968* (for ASCII) is used by default. If no *inputfile* is given, or if it is given as a dash (−), **localedef** reads from standard input.

OPTIONS

Operation-selection options

A few options direct **localedef** to do something other than compile locale definitions. Only one of these options should be used at a time.

−−add−to−archive
: Add the *compiledpath* directories to the locale archive file. The directories should have been created by previous runs of **localedef**, using **--no-archive**.

−−delete−from−archive
: Delete the named locales from the locale archive file.

−−list−archive
: List the locales contained in the locale archive file.

Other options

Some of the following options are sensible only for certain operations; generally, it should be self-evident which ones. Notice that **−f** and **−c** are reversed from what you might expect; that is, **−f** is not the same as **--force**.

−f *charmapfile*, **−−charmap=***charmapfile*
Specify the file that defines the character set that is used by the input file. If *charmapfile* contains a slash character ('/'), it is interpreted as the name of the character map. Otherwise, the file is sought in the current directory and the default directory for character maps. If the environment variable **I18NPATH** is set, *$I18NPATH/charmaps/* and *$I18NPATH/* are also searched after the current directory. The default directory for character maps is printed by **localedef --help**.

−i *inputfile*, **−−inputfile=***inputfile*
Specify the locale definition file to compile. The file is sought in the current directory and the default directory for locale definition files. If the environment variable **I18NPATH** is set, *$I18NPATH/locales/* and *$I18NPATH* are also searched after the current directory. The default directory for locale definition files is printed by **localedef --help**.

−u *repertoirefile*, **−−repertoire−map=***repertoirefile*
Read mappings from symbolic names to Unicode code points from *repertoirefile*. If *repertoirefile* contains a slash character ('/'), it is interpreted as the pathname of the repertoire map. Otherwise, the file is sought in the current directory and the default directory for repertoire maps. If the environment variable **I18NPATH** is set, *$I18NPATH/repertoiremaps/* and *$I18NPATH* are also searched after the current directory. The default directory for repertoire maps is printed by **localedef --help**.

−A *aliasfile*, **−−alias−file=***aliasfile*
Use *aliasfile* to look up aliases for locale names. There is no default aliases file.

−−force
−c Write the output files even if warnings were generated about the input file.

−−verbose
−v Generate extra warnings about errors that are normally ignored.

−−big−endian
Generate big-endian output.

−−little−endian
Generate little-endian output.

−−no−archive
Do not use the locale archive file, instead create *outputpath* as a subdirectory in the same directory as the locale archive file, and create separate output files for locale categories in it. This is helpful to prevent system locale archive updates from overwriting custom locales created with **localedef**.

−−no−hard−links
Do not create hard links between installed locales.

−−no−warnings=*warnings*
Comma-separated list of warnings to disable. Supported warnings are *ascii* and *intcurrsym*.

−−posix
Conform strictly to POSIX. Implies **--verbose**. This option currently has no other effect. POSIX conformance is assumed if the environment variable **POSIXLY_CORRECT** is set.

−−prefix= *pathname*
Set the prefix to be prepended to the full archive pathname. By default, the prefix is empty. Setting the prefix to *foo*, the archive would be placed in *foo/usr/lib/locale/locale−archive*.

−−quiet
Suppress all notifications and warnings, and report only fatal errors.

−−replace
Replace a locale in the locale archive file. Without this option, if the locale is in the archive file already, an error occurs.

−−warnings=*warnings*
Comma-separated list of warnings to enable. Supported warnings are *ascii* and *intcurrsym*.

−−help
−? Print a usage summary and exit. Also prints the default paths used by **localedef**.

−−usage
Print a short usage summary and exit.

−−version
−V Print the version number, license, and disclaimer of warranty for **localedef**.

EXIT STATUS

One of the following exit values can be returned by **localedef**:

0 Command completed successfully.

1 Warnings or errors occurred, output files were written.

4 Errors encountered, no output created.

ENVIRONMENT

POSIXLY_CORRECT
The **−−posix** flag is assumed if this environment variable is set.

I18NPATH
A colon-separated list of search directories for files.

FILES

/usr/share/i18n/charmaps
Usual default character map path.

/usr/share/i18n/locales
Usual default path for locale definition files.

/usr/share/i18n/repertoiremaps
Usual default repertoire map path.

/usr/lib/locale/locale−archive
Usual default locale archive location.

/usr/lib/locale
Usual default path for compiled individual locale data files.

outputpath/LC_ADDRESS
An output file that contains information about formatting of addresses and geography-related items.

outputpath/LC_COLLATE
An output file that contains information about the rules for comparing strings.

outputpath/LC_CTYPE
An output file that contains information about character classes.

outputpath/LC_IDENTIFICATION
An output file that contains metadata about the locale.

outputpath/LC_MEASUREMENT
An output file that contains information about locale measurements (metric versus US customary).

outputpath/LC_MESSAGES/SYS_LC_MESSAGES
An output file that contains information about the language messages should be printed in, and what an affirmative or negative answer looks like.

outputpath/LC_MONETARY
An output file that contains information about formatting of monetary values.

outputpath/LC_NAME
An output file that contains information about salutations for persons.

outputpath/LC_NUMERIC
An output file that contains information about formatting of nonmonetary numeric values.

outputpath/LC_PAPER
: An output file that contains information about settings related to standard paper size.

outputpath/LC_TELEPHONE
: An output file that contains information about formats to be used with telephone services.

outputpath/LC_TIME
: An output file that contains information about formatting of data and time values.

STANDARDS

POSIX.1-2008.

EXAMPLES

Compile the locale files for Finnish in the UTF−8 character set and add it to the default locale archive with the name **fi_FI.UTF-8**:

```
localedef −f UTF−8 −i fi_FI fi_FI.UTF−8
```

The next example does the same thing, but generates files into the *fi_FI.UTF−8* directory which can then be used by programs when the environment variable **LOCPATH** is set to the current directory (note that the last argument must contain a slash):

```
localedef −f UTF−8 −i fi_FI ./fi_FI.UTF−8
```

SEE ALSO

locale(1), *charmap*(5), *locale*(5), *repertoiremap*(5), *locale*(7)

NAME

memusage − profile memory usage of a program

SYNOPSIS

memusage [*option*]... *program* [*programoption*]...

DESCRIPTION

memusage is a bash script which profiles memory usage of the program, *program*. It preloads the **libmemusage.so** library into the caller's environment (via the **LD_PRELOAD** environment variable; see *ld.so*(8)). The **libmemusage.so** library traces memory allocation by intercepting calls to *malloc*(3), *calloc*(3), *free*(3), and *realloc*(3); optionally, calls to *mmap*(2), *mremap*(2), and *munmap*(2) can also be intercepted.

memusage can output the collected data in textual form, or it can use *memusagestat*(1) (see the **−p** option, below) to create a PNG file containing graphical representation of the collected data.

Memory usage summary

The "Memory usage summary" line output by **memusage** contains three fields:

heap total
: Sum of *size* arguments of all *malloc*(3) calls, products of arguments (*nmemb***size*) of all *calloc*(3) calls, and sum of *length* arguments of all *mmap*(2) calls. In the case of *realloc*(3) and *mremap*(2), if the new size of an allocation is larger than the previous size, the sum of all such differences (new size minus old size) is added.

heap peak
: Maximum of all *size* arguments of *malloc*(3), all products of *nmemb***size* of *calloc*(3), all *size* arguments of *realloc*(3), *length* arguments of *mmap*(2), and *new_size* arguments of *mremap*(2).

stack peak
: Before the first call to any monitored function, the stack pointer address (base stack pointer) is saved. After each function call, the actual stack pointer address is read and the difference from the base stack pointer computed. The maximum of these differences is then the stack peak.

Immediately following this summary line, a table shows the number calls, total memory allocated or deallocated, and number of failed calls for each intercepted function. For *realloc*(3) and *mremap*(2), the additional field "nomove" shows reallocations that changed the address of a block, and the additional "dec" field shows reallocations that decreased the size of the block. For *realloc*(3), the additional field "free" shows reallocations that caused a block to be freed (i.e., the reallocated size was 0).

The "realloc/total memory" of the table output by **memusage** does not reflect cases where *realloc*(3) is used to reallocate a block of memory to have a smaller size than previously. This can cause sum of all "total memory" cells (excluding "free") to be larger than the "free/total memory" cell.

Histogram for block sizes

The "Histogram for block sizes" provides a breakdown of memory allocations into various bucket sizes.

OPTIONS

−n *name*, **−−progname=***name*
: Name of the program file to profile.

−p *file*, **−−png=** *file*
: Generate PNG graphic and store it in *file*.

−d *file*, **−−data=** *file*
: Generate binary data file and store it in *file*.

−u, **−−unbuffered**
: Do not buffer output.

−b *size*, **−−buffer=***size*
: Collect *size* entries before writing them out.

−−no−timer
: Disable timer-based (**SIGPROF**) sampling of stack pointer value.

−m, **−−mmap**
Also trace *mmap*(2), *mremap*(2), and *munmap*(2).

−?, **−−help**
Print help and exit.

−−usage
Print a short usage message and exit.

−V, **−−version**
Print version information and exit.

The following options apply only when generating graphical output:

−t, **−−time−based**
Use time (rather than number of function calls) as the scale for the X axis.

−T, **−−total**
Also draw a graph of total memory use.

−−title=*name*
Use *name* as the title of the graph.

−x *size*, **−−x−size**=*size*
Make the graph *size* pixels wide.

−y *size*, **−−y−size**=*size*
Make the graph *size* pixels high.

EXIT STATUS

The exit status of **memusage** is equal to the exit status of the profiled program.

BUGS

To report bugs, see

EXAMPLES

Below is a simple program that reallocates a block of memory in cycles that rise to a peak before then cyclically reallocating the memory in smaller blocks that return to zero. After compiling the program and running the following commands, a graph of the memory usage of the program can be found in the file *memusage.png*:

```
$ memusage --data=memusage.dat ./a.out
...
Memory usage summary: heap total: 45200, heap peak: 6440, stack peak: 224
        total calls  total memory  failed calls
 malloc|         1           400             0
realloc|        40         44800             0  (nomove:40, dec:19, free:0)
 calloc|         0             0             0
   free|         1           440
Histogram for block sizes:
  192-207             1   2% ================
...
 2192-2207            1   2% ================
 2240-2255            2   4% ================================
 2832-2847            2   4% ================================
 3440-3455            2   4% ================================
 4032-4047            2   4% ================================
 4640-4655            2   4% ================================
 5232-5247            2   4% ================================
 5840-5855            2   4% ================================
 6432-6447            1   2% ================
$ memusagestat memusage.dat memusage.png
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define CYCLES 20

int
main(int argc, char *argv[])
{
    int i, j;
    size_t size;
    int *p;

    size = sizeof(*p) * 100;
    printf("malloc: %zu\n", size);
    p = malloc(size);

    for (i = 0; i < CYCLES; i++) {
        if (i < CYCLES / 2)
            j = i;
        else
            j--;

        size = sizeof(*p) * (j * 50 + 110);
        printf("realloc: %zu\n", size);
        p = realloc(p, size);

        size = sizeof(*p) * ((j + 1) * 150 + 110);
        printf("realloc: %zu\n", size);
        p = realloc(p, size);
    }

    free(p);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

memusagestat(1), *mtrace*(1), *ld.so*(8)

NAME

memusagestat − generate graphic from memory profiling data

SYNOPSIS

memusagestat [*option*]... *datafile* [*outfile*]

DESCRIPTION

memusagestat creates a PNG file containing a graphical representation of the memory profiling data in the file *datafile*; that file is generated via the *−d* (or *−−data*) option of *memusage*(1).

The red line in the graph shows the heap usage (allocated memory) and the green line shows the stack usage. The x-scale is either the number of memory-handling function calls or (if the *−t* option is specified) time.

OPTIONS

−o *file*, **−−output=** *file*
: Name of the output file.

−s *string*, **−−string**=*string*
: Use *string* as the title inside the output graph.

−t, **−−time**
: Use time (rather than number of function calls) as the scale for the X axis.

−T, **−−total**
: Also draw a graph of total memory consumption.

−x *size*, **−−x−size**=*size*
: Make the output graph *size* pixels wide.

−y *size*, **−−y−size**=*size*
: Make the output graph *size* pixels high.

−?, **−−help**
: Print a help message and exit.

−−usage
: Print a short usage message and exit.

−V, **−−version**
: Print version information and exit.

BUGS

To report bugs, see

EXAMPLES

See *memusage*(1).

SEE ALSO

memusage(1), *mtrace*(1)

NAME

mtrace − interpret the malloc trace log

SYNOPSIS

mtrace [*option*]... [*binary*] *mtracedata*

DESCRIPTION

mtrace is a Perl script used to interpret and provide human readable output of the trace log contained in the file *mtracedata*, whose contents were produced by *mtrace*(3). If *binary* is provided, the output of **mtrace** also contains the source file name with line number information for problem locations (assuming that *binary* was compiled with debugging information).

For more information about the *mtrace*(3) function and **mtrace** script usage, see *mtrace*(3).

OPTIONS

−−help Print help and exit.

−−version
Print version information and exit.

BUGS

For bug reporting instructions, please see: .

SEE ALSO

memusage(1), *mtrace*(3)

NAME

pldd − display dynamic shared objects linked into a process

SYNOPSIS

pldd *pid*
pldd *option*

DESCRIPTION

The **pldd** command displays a list of the dynamic shared objects (DSOs) that are linked into the process with the specified process ID (PID). The list includes the libraries that have been dynamically loaded using *dlopen*(3).

OPTIONS

−−help
−? Display a help message and exit.

−−usage
Display a short usage message and exit.

−−version
−V Display program version information and exit.

EXIT STATUS

On success, **pldd** exits with the status 0. If the specified process does not exist, the user does not have permission to access its dynamic shared object list, or no command-line arguments are supplied, **pldd** exists with a status of 1. If given an invalid option, it exits with the status 64.

VERSIONS

Some other systems have a similar command.

STANDARDS

None.

HISTORY

glibc 2.15.

NOTES

The command

```
lsof −p PID
```

also shows output that includes the dynamic shared objects that are linked into a process.

The *gdb*(1) *info shared* command also shows the shared libraries being used by a process, so that one can obtain similar output to **pldd** using a command such as the following (to monitor the process with the specified *pid*):

```
$ gdb −ex "set confirm off" −ex "set height 0" −ex "info shared" \
       −ex "quit" −p $pid | grep '^0x.*0x'
```

BUGS

From glibc 2.19 to glibc 2.29, **pldd** was broken: it just hung when executed. This problem was fixed in glibc 2.30, and the fix has been backported to earlier glibc versions in some distributions.

EXAMPLES

```
$ echo $$               # Display PID of shell
1143
$ pldd $$               # Display DSOs linked into the shell
1143:   /usr/bin/bash
linux−vdso.so.1
/lib64/libtinfo.so.5
/lib64/libdl.so.2
/lib64/libc.so.6
/lib64/ld−linux−x86−64.so.2
/lib64/libnss_files.so.2
```

SEE ALSO

ldd(1), *lsof* (1), *dlopen*(3), *ld.so*(8)

NAME

sprof − read and display shared object profiling data

SYNOPSIS

sprof [*option*]... *shared-object-path* [*profile-data-path*]

DESCRIPTION

The **sprof** command displays a profiling summary for the shared object (shared library) specified as its first command-line argument. The profiling summary is created using previously generated profiling data in the (optional) second command-line argument. If the profiling data pathname is omitted, then **sprof** will attempt to deduce it using the soname of the shared object, looking for a file with the name *<soname>.profile* in the current directory.

OPTIONS

The following command-line options specify the profile output to be produced:

−−call−pairs

−c Print a list of pairs of call paths for the interfaces exported by the shared object, along with the number of times each path is used.

−−flat−profile

−p Generate a flat profile of all of the functions in the monitored object, with counts and ticks.

−−graph

−q Generate a call graph.

If none of the above options is specified, then the default behavior is to display a flat profile and a call graph.

The following additional command-line options are available:

−−help

−? Display a summary of command-line options and arguments and exit.

−−usage

Display a short usage message and exit.

−−version

−V Display the program version and exit.

STANDARDS

GNU.

EXAMPLES

The following example demonstrates the use of **sprof**. The example consists of a main program that calls two functions in a shared object. First, the code of the main program:

```
$ cat prog.c
#include <stdlib.h>

void x1(void);
void x2(void);

int
main(int argc, char *argv[])
{
    x1();
    x2();
    exit(EXIT_SUCCESS);
}
```

The functions *x1*() and *x2*() are defined in the following source file that is used to construct the shared object:

```
$ cat libdemo.c
#include <unistd.h>

void
```

```
consumeCpu1(int lim)
{
    for (unsigned int j = 0; j < lim; j++)
      getppid();
}

void
x1(void) {
    for (unsigned int j = 0; j < 100; j++)
      consumeCpu1(200000);
}

void
consumeCpu2(int lim)
{
    for (unsigned int j = 0; j < lim; j++)
      getppid();
}

void
x2(void)
{
    for (unsigned int j = 0; j < 1000; j++)
      consumeCpu2(10000);
}
```

Now we construct the shared object with the real name *libdemo.so.1.0.1*, and the soname *libdemo.so.1*:

```
$ cc -g -fPIC -shared -Wl,-soname,libdemo.so.1 \
      -o libdemo.so.1.0.1 libdemo.c
```

Then we construct symbolic links for the library soname and the library linker name:

```
$ ln -sf libdemo.so.1.0.1 libdemo.so.1
$ ln -sf libdemo.so.1 libdemo.so
```

Next, we compile the main program, linking it against the shared object, and then list the dynamic dependencies of the program:

```
$ cc -g -o prog prog.c -L. -ldemo
$ ldd prog
      linux-vdso.so.1 =>  (0x00007fff86d66000)
      libdemo.so.1 => not found
      libc.so.6 => /lib64/libc.so.6 (0x00007fd4dc138000)
      /lib64/ld-linux-x86-64.so.2 (0x00007fd4dc51f000)
```

In order to get profiling information for the shared object, we define the environment variable **LD_PROFILE** with the soname of the library:

```
$ export LD_PROFILE=libdemo.so.1
```

We then define the environment variable **LD_PROFILE_OUTPUT** with the pathname of the directory where profile output should be written, and create that directory if it does not exist already:

```
$ export LD_PROFILE_OUTPUT=$(pwd)/prof_data
$ mkdir -p $LD_PROFILE_OUTPUT
```

LD_PROFILE causes profiling output to be *appended* to the output file if it already exists, so we ensure that there is no preexisting profiling data:

```
$ rm -f $LD_PROFILE_OUTPUT/$LD_PROFILE.profile
```

We then run the program to produce the profiling output, which is written to a file in the directory specified in **LD_PROFILE_OUTPUT**:

```
$ LD_LIBRARY_PATH=. ./prog
$ ls prof_data
```

```
libdemo.so.1.profile
```

We then use the **sprof −p** option to generate a flat profile with counts and ticks:

```
$ sprof -p libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 60.00      0.06     0.06      100   600.00           consumeCpu1
 40.00      0.10     0.04     1000    40.00           consumeCpu2
  0.00      0.10     0.00        1     0.00           x1
  0.00      0.10     0.00        1     0.00           x2
```

The **sprof −q** option generates a call graph:

```
$ sprof -q libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile

index % time    self  children    called     name

                0.00    0.00      100/100         x1 [1]
[0]    100.0    0.00    0.00      100          consumeCpu1 [0]
-----------------------------------------------
                0.00    0.00        1/1           <UNKNOWN>
[1]      0.0    0.00    0.00        1          x1 [1]
                0.00    0.00      100/100         consumeCpu1 [0]
-----------------------------------------------
                0.00    0.00     1000/1000        x2 [3]
[2]      0.0    0.00    0.00     1000          consumeCpu2 [2]
-----------------------------------------------
                0.00    0.00        1/1           <UNKNOWN>
[3]      0.0    0.00    0.00        1          x2 [3]
                0.00    0.00     1000/1000        consumeCpu2 [2]
-----------------------------------------------
```

Above and below, the "<UNKNOWN>" strings represent identifiers that are outside of the profiled object (in this example, these are instances of *main()*).

The **sprof −c** option generates a list of call pairs and the number of their occurrences:

```
$ sprof -c libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile
<UNKNOWN>                   x1                                1
x1                          consumeCpu1                     100
<UNKNOWN>                   x2                                1
x2                          consumeCpu2                    1000
```

SEE ALSO

gprof (1), *ldd*(1), *ld.so*(8)

NAME

time − time a simple command or give resource usage

SYNOPSIS

time [*option* ...] *command* [*argument* ...]

DESCRIPTION

The **time** command runs the specified program *command* with the given arguments. When *command* finishes, **time** writes a message to standard error giving timing statistics about this program run. These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time (the sum of the *tms_utime* and *tms_cutime* values in a *struct tms* as returned by *times*(2)), and (iii) the system CPU time (the sum of the *tms_stime* and *tms_cstime* values in a *struct tms* as returned by *times*(2)).

Note: some shells (e.g., *bash*(1)) have a built-in **time** command that provides similar information on the usage of time and possibly other resources. To access the real command, you may need to specify its pathname (something like */usr/bin/time*).

OPTIONS

−p When in the POSIX locale, use the precise traditional format

```
"real %f\nuser %f\nsys %f\n"
```

(with numbers in seconds) where the number of decimals in the output for %f is unspecified but is sufficient to express the clock tick accuracy, and at least one.

EXIT STATUS

If *command* was invoked, the exit status is that of *command*. Otherwise, it is 127 if *command* could not be found, 126 if it could be found but could not be invoked, and some other nonzero value (1–125) if something else went wrong.

ENVIRONMENT

The variables **LANG**, **LC_ALL**, **LC_CTYPE**, **LC_MESSAGES**, **LC_NUMERIC**, and **NLSPATH** are used for the text and formatting of the output. **PATH** is used to search for *command*.

GNU VERSION

Below a description of the GNU 1.7 version of **time**. Disregarding the name of the utility, GNU makes it output lots of useful information, not only about time used, but also on other resources like memory, I/O and IPC calls (where available). The output is formatted using a format string that can be specified using the *−f* option or the **TIME** environment variable.

The default format string is:

```
%Uuser %Ssystem %Eelapsed %PCPU (%Xtext+%Ddata %Mmax)k
%Iinputs+%Ooutputs (%Fmajor+%Rminor)pagefaults %Wswaps
```

When the *−p* option is given, the (portable) output format is used:

```
real %e
user %U
sys %S
```

The format string

The format is interpreted in the usual printf-like way. Ordinary characters are directly copied, tab, newline, and backslash are escaped using \t, \n, and \\, a percent sign is represented by %%, and otherwise % indicates a conversion. The program **time** will always add a trailing newline itself. The conversions follow. All of those used by *tcsh*(1) are supported.

Time

%E Elapsed real time (in [hours:]minutes:seconds).

%e (Not in *tcsh*(1)Elapsed real time (in seconds).

%S Total number of CPU-seconds that the process spent in kernel mode.

%U Total number of CPU-seconds that the process spent in user mode.

%P Percentage of the CPU that this job got, computed as (%U + %S) / %E.

Memory

%M Maximum resident set size of the process during its lifetime, in Kbytes.

%t (Not in *tcsh*(1)Average resident set size of the process, in Kbytes.

%K Average total (data+stack+text) memory use of the process, in Kbytes.

%D Average size of the process's unshared data area, in Kbytes.

%p (Not in *tcsh*(1)Average size of the process's unshared stack space, in Kbytes.

%X Average size of the process's shared text space, in Kbytes.

%Z (Not in *tcsh*(1)System's page size, in bytes. This is a per-system constant, but varies between systems.

%F Number of major page faults that occurred while the process was running. These are faults where the page has to be read in from disk.

%R Number of minor, or recoverable, page faults. These are faults for pages that are not valid but which have not yet been claimed by other virtual pages. Thus the data in the page is still valid but the system tables must be updated.

%W Number of times the process was swapped out of main memory.

%c Number of times the process was context-switched involuntarily (because the time slice expired).

%w Number of waits: times that the program was context-switched voluntarily, for instance while waiting for an I/O operation to complete.

I/O

%I Number of filesystem inputs by the process.

%O Number of filesystem outputs by the process.

%r Number of socket messages received by the process.

%s Number of socket messages sent by the process.

%k Number of signals delivered to the process.

%C (Not in *tcsh*(1)Name and command-line arguments of the command being timed.

%x (Not in *tcsh*(1)Exit status of the command.

GNU options

−f *format*, **−−format=** *format*
Specify output format, possibly overriding the format specified in the environment variable TIME.

−p, −−portability
Use the portable output format.

−o *file*, **−−output=** *file*
Do not send the results to *stderr*, but overwrite the specified file.

−a, −−append
(Used together with −o.) Do not overwrite but append.

−v, −−verbose
Give very verbose output about all the program knows about.

−q, −−quiet
Don't report abnormal program termination (where *command* is terminated by a signal) or nonzero exit status.

GNU standard options

−−help Print a usage message on standard output and exit successfully.

−V, −−version
Print version information on standard output, then exit successfully.

−− Terminate option list.

BUGS

Not all resources are measured by all versions of UNIX, so some of the values might be reported as zero. The present selection was mostly inspired by the data provided by 4.2 or 4.3BSD.

GNU time version 1.7 is not yet localized. Thus, it does not implement the POSIX requirements.

The environment variable **TIME** was badly chosen. It is not unusual for systems like *autoconf*(1) or *make*(1) to use environment variables with the name of a utility to override the utility to be used. Uses like MORE or TIME for options to programs (instead of program pathnames) tend to lead to difficulties.

It seems unfortunate that −*o* overwrites instead of appends. (That is, the −*a* option should be the default.)

Mail suggestions and bug reports for GNU **time** to *bug−time@gnu.org*. Please include the version of **time**, which you can get by running

```
time −−version
```

and the operating system and C compiler you used.

SEE ALSO

bash(1), *tcsh*(1), *times*(2), *wait3*(2)

NAME

intro − introduction to system calls

DESCRIPTION

Section 2 of the manual describes the Linux system calls. A system call is an entry point into the Linux kernel. Usually, system calls are not invoked directly: instead, most system calls have corresponding C library wrapper functions which perform the steps required (e.g., trapping to kernel mode) in order to invoke the system call. Thus, making a system call looks the same as invoking a normal library function.

In many cases, the C library wrapper function does nothing more than:

- copying arguments and the unique system call number to the registers where the kernel expects them;
- trapping to kernel mode, at which point the kernel does the real work of the system call;
- setting *errno* if the system call returns an error number when the kernel returns the CPU to user mode.

However, in a few cases, a wrapper function may do rather more than this, for example, performing some preprocessing of the arguments before trapping to kernel mode, or postprocessing of values returned by the system call. Where this is the case, the manual pages in Section 2 generally try to note the details of both the (usually GNU) C library API interface and the raw system call. Most commonly, the main DESCRIPTION will focus on the C library interface, and differences for the system call are covered in the NOTES section.

For a list of the Linux system calls, see *syscalls*(2).

RETURN VALUE

On error, most system calls return a negative error number (i.e., the negated value of one of the constants described in *errno*(3)). The C library wrapper hides this detail from the caller: when a system call returns a negative value, the wrapper copies the absolute value into the *errno* variable, and returns −1 as the return value of the wrapper.

The value returned by a successful system call depends on the call. Many system calls return 0 on success, but some can return nonzero values from a successful call. The details are described in the individual manual pages.

In some cases, the programmer must define a feature test macro in order to obtain the declaration of a system call from the header file specified in the man page SYNOPSIS section. (Where required, these feature test macros must be defined before including *any* header files.) In such cases, the required macro is described in the man page. For further information on feature test macros, see *feature_test_macros*(7).

STANDARDS

Certain terms and abbreviations are used to indicate UNIX variants and standards to which calls in this section conform. See *standards*(7).

NOTES

Calling directly

In most cases, it is unnecessary to invoke a system call directly, but there are times when the Standard C library does not implement a nice wrapper function for you. In this case, the programmer must manually invoke the system call using *syscall*(2). Historically, this was also possible using one of the _syscall macros described in *_syscall*(2).

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

SEE ALSO

_syscall(2), *syscall*(2), *syscalls*(2), *errno*(3), *intro*(3), *capabilities*(7), *credentials*(7), *feature_test_macros*(7), *mq_overview*(7), *path_resolution*(7), *pipe*(7), *pty*(7), *sem_overview*(7), *shm_overview*(7), *signal*(7), *socket*(7), *standards*(7), *symlink*(7), *system_data_types*(7), *sysvipc*(7), *time*(7)

NAME

accept, accept4 − accept a connection on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *_Nullable restrict addr,
           socklen_t *_Nullable restrict addrlen);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/socket.h>

int accept4(int sockfd, struct sockaddr *_Nullable restrict addr,
           socklen_t *_Nullable restrict addrlen, int flags);
```

DESCRIPTION

The **accept**() system call is used with connection-based socket types (**SOCK_STREAM**, **SOCK_SEQPACKET**). It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call.

The argument *sockfd* is a socket that has been created with *socket*(2), bound to a local address with *bind*(2), and is listening for connections after a *listen*(2).

The argument *addr* is a pointer to a *sockaddr* structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned *addr* is determined by the socket's address family (see *socket*(2) and the respective protocol man pages). When *addr* is NULL, nothing is filled in; in this case, *addrlen* is not used, and should also be NULL.

The *addrlen* argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by *addr*; on return it will contain the actual size of the peer address.

The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

If no pending connections are present on the queue, and the socket is not marked as nonblocking, **accept**() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, **accept**() fails with the error **EAGAIN** or **EWOULDBLOCK**.

In order to be notified of incoming connections on a socket, you can use *select*(2), *poll*(2), or *epoll*(7). A readable event will be delivered when a new connection is attempted and you may then call **accept**() to get a socket for that connection. Alternatively, you can set the socket to deliver **SIGIO** when activity occurs on a socket; see *socket*(7) for details.

If *flags* is 0, then **accept4**() is the same as **accept**(). The following values can be bitwise ORed in *flags* to obtain different behavior:

SOCK_NONBLOCK
: Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

SOCK_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

RETURN VALUE

On success, these system calls return a file descriptor for the accepted socket (a nonnegative integer). On error, −1 is returned, *errno* is set to indicate the error, and *addrlen* is left unchanged.

Error handling

Linux **accept**() (and *accept4*()) passes already-pending network errors on the new socket as an error code from **accept**(). This behavior differs from other BSD socket implementations. For reliable operation the application should detect the network errors defined for the protocol after **accept**() and treat

them like **EAGAIN** by retrying. In the case of TCP/IP, these are **ENETDOWN**, **EPROTO**, **ENOPROTOOPT**, **EHOSTDOWN**, **ENONET**, **EHOSTUNREACH**, **EOPNOTSUPP**, and **ENETUNREACH**.

ERRORS

EAGAIN or **EWOULDBLOCK**
: The socket is marked nonblocking and no connections are present to be accepted. POSIX.1-2001 and POSIX.1-2008 allow either error to be returned for this case, and do not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF
: *sockfd* is not an open file descriptor.

ECONNABORTED
: A connection has been aborted.

EFAULT
: The *addr* argument is not in a writable part of the user address space.

EINTR
: The system call was interrupted by a signal that was caught before a valid connection arrived; see *signal*(7).

EINVAL
: Socket is not listening for connections, or *addrlen* is invalid (e.g., is negative).

EINVAL
: (**accept4**()) invalid value in *flags*.

EMFILE
: The per-process limit on the number of open file descriptors has been reached.

ENFILE
: The system-wide limit on the total number of open files has been reached.

ENOBUFS
ENOMEM
: Not enough free memory. This often means that the memory allocation is limited by the socket buffer limits, not by the system memory.

ENOTSOCK
: The file descriptor *sockfd* does not refer to a socket.

EOPNOTSUPP
: The referenced socket is not of type **SOCK_STREAM**.

EPERM
: Firewall rules forbid connection.

EPROTO
: Protocol error.

In addition, network errors for the new socket and as defined for the protocol may be returned. Various Linux kernels can return other errors such as **ENOSR**, **ESOCKTNOSUPPORT**, **EPROTONOSUPPORT**, **ETIMEDOUT**. The value **ERESTARTSYS** may be seen during a trace.

VERSIONS

On Linux, the new socket returned by **accept**() does *not* inherit file status flags such as **O_NONBLOCK** and **O_ASYNC** from the listening socket. This behavior differs from the canonical BSD sockets implementation. Portable programs should not rely on inheritance or noninheritance of file status flags and always explicitly set all required flags on the socket returned from **accept**().

STANDARDS

accept()
: POSIX.1-2008.

accept4()
Linux.

HISTORY

accept()
POSIX.1-2001, SVr4, 4.4BSD (**accept**() first appeared in 4.2BSD).

accept4()
Linux 2.6.28, glibc 2.10.

NOTES

There may not always be a connection waiting after a **SIGIO** is delivered or *select*(2), *poll*(2), or *epoll*(7) return a readability event because the connection might have been removed by an asynchronous network error or another thread before **accept**() is called. If this happens, then the call will block waiting for the next connection to arrive. To ensure that **accept**() never blocks, the passed socket *sockfd* needs to have the **O_NONBLOCK** flag set (see *socket*(7)).

For certain protocols which require an explicit confirmation, such as DECnet, **accept**() can be thought of as merely dequeuing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket. Currently, only DECnet has these semantics on Linux.

The socklen_t type

In the original BSD sockets implementation (and on other older systems) the third argument of **accept**() was declared as an *int* *. A POSIX.1g draft standard wanted to change it into a *size_t* *C; later POSIX standards and glibc 2.x have *socklen_t* * .

EXAMPLES

See *bind*(2).

SEE ALSO

bind(2), *connect*(2), *listen*(2), *select*(2), *socket*(2), *socket*(7)

NAME

access, faccessat, faccessat2 − check user's permissions for a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int access(const char *pathname, int mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int faccessat(int dirfd, const char *pathname, int mode, int  flags);
              /* But see C library/kernel differences, below */

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_faccessat2,
            int dirfd, const char *pathname, int mode, int  flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

faccessat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

access() checks whether the calling process can access the file *pathname*. If *pathname* is a symbolic link, it is dereferenced.

The *mode* specifies the accessibility check(s) to be performed, and is either the value **F_OK**, or a mask consisting of the bitwise OR of one or more of **R_OK**, **W_OK**, and **X_OK**. **F_OK** tests for the existence of the file. **R_OK**, **W_OK**, and **X_OK** test whether the file exists and grants read, write, and execute permissions, respectively.

The check is done using the calling process's *real* UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., *open*(2)) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.

This allows set-user-ID programs and capability-endowed programs to easily determine the invoking user's authority. In other words, **access**() does not answer the "can I read/write/execute this file?" question. It answers a slightly different question: "(assuming I'm a setuid binary) can *the user who invoked me* read/write/execute this file?", which gives set-user-ID programs the possibility to prevent malicious users from causing them to read files which users shouldn't be able to read.

If the calling process is privileged (i.e., its real UID is zero), then an **X_OK** check is successful for a regular file if execute permission is enabled for any of the file owner, group, or other.

faccessat()

faccessat() operates in exactly the same way as **access**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **access**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *access*())

If *pathname* is absolute, then *dirfd* is ignored.

flags is constructed by ORing together zero or more of the following values:

AT_EACCESS
Perform access checks using the effective user and group IDs. By default, **faccessat**() uses the real IDs (like *access*())

AT_EMPTY_PATH (since Linux 5.8)
If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory. If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_NOFOLLOW
If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself.

See *openat*(2) for an explanation of the need for **faccessat**().

faccessat2()
The description of **faccessat**() given above corresponds to POSIX.1 and to the implementation provided by glibc. However, the glibc implementation was an imperfect emulation (see BUGS) that papered over the fact that the raw Linux **faccessat**() system call does not have a *flags* argument. To allow for a proper implementation, Linux 5.8 added the **faccessat2**() system call, which supports the *flags* argument and allows a correct implementation of the **faccessat**() wrapper function.

RETURN VALUE
On success (all requested permissions granted, or *mode* is **F_OK** and the file exists), zero is returned. On error (at least one bit in *mode* asked for a permission that is denied, or *mode* is **F_OK** and the file does not exist, or some other error occurred), −1 is returned, and *errno* is set to indicate the error.

ERRORS
EACCES
The requested access would be denied to the file, or search permission is denied for one of the directories in the path prefix of *pathname*. (See also *path_resolution*(7).)

EBADF
(**faccessat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** (**faccessat**()) nor a valid file descriptor.

EFAULT
pathname points outside your accessible address space.

EINVAL
mode was incorrectly specified.

EINVAL
(**faccessat**()) Invalid flag specified in *flags*.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname is too long.

ENOENT
A component of *pathname* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR
(**faccessat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM
Write permission was requested to a file that has the immutable flag set. See also *ioctl_iflags*(2).

EROFS
Write permission was requested for a file on a read-only filesystem.

ETXTBSY
Write access was requested to an executable which is being executed.

VERSIONS

If the calling process has appropriate privileges (i.e., is superuser), POSIX.1-2001 permits an implementation to indicate success for an **X_OK** check even if none of the execute file permission bits are set. Linux does not do this.

C library/kernel differences

The raw **faccessat**() system call takes only the first three arguments. The **AT_EACCESS** and **AT_SYMLINK_NOFOLLOW** flags are actually implemented within the glibc wrapper function for **faccessat**(). If either of these flags is specified, then the wrapper function employs *fstatat*(2) to determine access permissions, but see BUGS.

glibc notes

On older kernels where **faccessat**() is unavailable (and when the **AT_EACCESS** and **AT_SYMLINK_NOFOLLOW** flags are not specified), the glibc wrapper function falls back to the use of **access**(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

STANDARDS

access()
faccessat()
POSIX.1-2008.

faccessat2()
Linux.

HISTORY

access()
SVr4, 4.3BSD, POSIX.1-2001.

faccessat()
Linux 2.6.16, glibc 2.4.

faccessat2()
Linux 5.8.

NOTES

Warning: Using these calls to check if a user is authorized to, for example, open a file before actually doing so using *open*(2) creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. **For this reason, the use of this system call should be avoided**. (In the example just described, a safer alternative would be to temporarily switch the process's effective user ID to the real ID and then call *open*(2).)

access() always dereferences symbolic links. If you need to check the permissions on a symbolic link, use **faccessat**() with the flag **AT_SYMLINK_NOFOLLOW**.

These calls return an error if any of the access types in *mode* is denied, even if some of the other access types in *mode* are permitted.

A file is accessible only if the permissions on each of the directories in the path prefix of *pathname* grant search (i.e., execute) access. If any directory is inaccessible, then the **access**() call fails, regardless of the permissions on the file itself.

Only access bits are checked, not the file type or contents. Therefore, if a directory is found to be writable, it probably means that files can be created in the directory, and not that the directory can be written as a file. Similarly, a DOS file may be reported as executable, but the *execve*(2) call will still fail.

These calls may not work correctly on NFSv2 filesystems with UID mapping enabled, because UID

mapping is done on the server and hidden from the client, which checks permissions. (NFS versions 3 and higher perform the check on the server.) Similar problems can occur to FUSE mounts.

BUGS

Because the Linux kernel's **faccessat**() system call does not support a *flags* argument, the glibc **faccessat**() wrapper function provided in glibc 2.32 and earlier emulates the required functionality using a combination of the **faccessat**() system call and *fstatat*(2). However, this emulation does not take ACLs into account. Starting with glibc 2.33, the wrapper function avoids this bug by making use of the **faccessat2**() system call where it is provided by the underlying kernel.

In Linux 2.4 (and earlier) there is some strangeness in the handling of **X_OK** tests for superuser. If all categories of execute permission are disabled for a nondirectory file, then the only **access**() test that returns −1 is when *mode* is specified as just **X_OK**; if **R_OK** or **W_OK** is also specified in *mode*, then **access**() returns 0 for such files. Early Linux 2.6 (up to and including Linux 2.6.3) also behaved in the same way as Linux 2.4.

Before Linux 2.6.20, these calls ignored the effect of the **MS_NOEXEC** flag if it was used to *mount*(2) the underlying filesystem. Since Linux 2.6.20, the **MS_NOEXEC** flag is honored.

SEE ALSO

chmod(2), *chown*(2), *open*(2), *setgid*(2), *setuid*(2), *stat*(2), *euidaccess*(3), *credentials*(7), *path_resolution*(7), *symlink*(7)

NAME

acct − switch process accounting on or off

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int acct(const char *_Nullable *filename***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

acct():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

The **acct**() system call enables or disables process accounting. If called with the name of an existing file as its argument, accounting is turned on, and records for each terminating process are appended to *filename* as it terminates. An argument of NULL causes accounting to be turned off.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Write permission is denied for the specified file, or search permission is denied for one of the directories in the path prefix of *filename* (see also *path_resolution*(7)), or *filename* is not a regular file.

EFAULT
filename points outside your accessible address space.

EIO Error writing to the file *filename*.

EISDIR
filename is a directory.

ELOOP
Too many symbolic links were encountered in resolving *filename*.

ENAMETOOLONG
filename was too long.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
The specified file does not exist.

ENOMEM
Out of memory.

ENOSYS
BSD process accounting has not been enabled when the operating system kernel was compiled. The kernel configuration parameter controlling this feature is **CONFIG_BSD_PROCESS_ACCT**.

ENOTDIR
A component used as a directory in *filename* is not in fact a directory.

EPERM
The calling process has insufficient privilege to enable process accounting. On Linux, the **CAP_SYS_PACCT** capability is required.

EROFS
filename refers to a file on a read-only filesystem.

EUSERS
There are no more free file structures or we ran out of memory.

STANDARDS
None.

HISTORY
SVr4, 4.3BSD.

NOTES
No accounting is produced for programs running when a system crash occurs. In particular, nonterminating processes are never accounted for.

The structure of the records written to the accounting file is described in *acct*(5).

SEE ALSO
acct(5)

NAME

add_key − add a key to the kernel's key management facility

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <keyutils.h>

key_serial_t add_key(const char ****type*, const char ****description***,**
const void *payload***[.**_plen_**], size_t** *plen***,**
key_serial_t *keyring***);**

Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION

add_key() creates or updates a key of the given *type* and *description*, instantiates it with the *payload* of length *plen*, attaches it to the nominated *keyring*, and returns the key's serial number.

The key may be rejected if the provided data is in the wrong format or it is invalid in some other way.

If the destination *keyring* already contains a key that matches the specified *type* and *description*, then, if the key type supports it, that key will be updated rather than a new key being created; if not, a new key (with a different ID) will be created and it will displace the link to the extant key from the keyring.

The destination *keyring* serial number may be that of a valid keyring for which the caller has *write* permission. Alternatively, it may be one of the following special keyring IDs:

KEY_SPEC_THREAD_KEYRING
This specifies the caller's thread-specific keyring (**thread−keyring**(7)).

KEY_SPEC_PROCESS_KEYRING
This specifies the caller's process-specific keyring (**process−keyring**(7)).

KEY_SPEC_SESSION_KEYRING
This specifies the caller's session-specific keyring (**session−keyring**(7)).

KEY_SPEC_USER_KEYRING
This specifies the caller's UID-specific keyring (**user−keyring**(7)).

KEY_SPEC_USER_SESSION_KEYRING
This specifies the caller's UID-session keyring (**user−session−keyring**(7)).

Key types

The key *type* is a string that specifies the key's type. Internally, the kernel defines a number of key types that are available in the core key management code. Among the types that are available for user-space use and can be specified as the *type* argument to **add_key**() are the following:

"keyring"
Keyrings are special key types that may contain links to sequences of other keys of any type. If this interface is used to create a keyring, then *payload* should be NULL and *plen* should be zero.

"user" This is a general purpose key type whose payload may be read and updated by user-space applications. The key is kept entirely within kernel memory. The payload for keys of this type is a blob of arbitrary data of up to 32,767 bytes.

"logon" (since Linux 3.3)
This key type is essentially the same as *"user"*, but it does not permit the key to read. This is suitable for storing payloads that you do not want to be readable from user space.

This key type vets the *description* to ensure that it is qualified by a "service" prefix, by checking to ensure that the *description* contains a ':' that is preceded by other characters.

"big_key" (since Linux 3.13)
This key type is similar to *"user"*, but may hold a payload of up to 1 MiB. If the key payload is large enough, then it may be stored encrypted in tmpfs (which can be swapped out) rather than kernel memory.

For further details on these key types, see *keyrings*(7).

RETURN VALUE

On success, **add_key**() returns the serial number of the key it created or updated. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
The keyring wasn't available for modification by the user.

EDQUOT
The key quota for this user would be exceeded by creating this key or linking it to the keyring.

EFAULT
One or more of *type*, *description*, and *payload* points outside process's accessible address space.

EINVAL
The size of the string (including the terminating null byte) specified in *type* or *description* exceeded the limit (32 bytes and 4096 bytes respectively).

EINVAL
The payload data was invalid.

EINVAL
type was *"logon"* and the *description* was not qualified with a prefix string of the form *"service:"*.

EKEYEXPIRED
The keyring has expired.

EKEYREVOKED
The keyring has been revoked.

ENOKEY
The keyring doesn't exist.

ENOMEM
Insufficient memory to create a key.

EPERM
The *type* started with a period ('.'). Key types that begin with a period are reserved to the implementation.

EPERM
type was *"keyring"* and the *description* started with a period ('.'). Keyrings with descriptions (names) that begin with a period are reserved to the implementation.

STANDARDS

Linux.

HISTORY

Linux 2.6.10.

NOTES

glibc does not provide a wrapper for this system call. A wrapper is provided in the *libkeyutils* library. (The accompanying package provides the <*keyutils.h*> header file.) When employing the wrapper in that library, link with −*lkeyutils*.

EXAMPLES

The program below creates a key with the type, description, and payload specified in its command-line arguments, and links that key into the session keyring. The following shell session demonstrates the use of the program:

```
$ ./a.out user mykey "Some payload"
Key ID is 64a4dca
$ grep '64a4dca' /proc/keys
064a4dca I--Q---     1 perm 3f010000  1000  1000 user    mykey: 12
```

Program source

```
#include <keyutils.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    key_serial_t key;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s type description payload\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    key = add_key(argv[1], argv[2], argv[3], strlen(argv[3]),
                  KEY_SPEC_SESSION_KEYRING);
    if (key == -1) {
        perror("add_key");
        exit(EXIT_FAILURE);
    }

    printf("Key ID is %jx\n", (uintmax_t) key);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

keyctl(1), *keyctl*(2), *request_key*(2), *keyctl*(3), *keyrings*(7), *keyutils*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7)

The kernel source files *Documentation/security/keys/core.rst* and *Documentation/keys/request−key.rst* (or, before Linux 4.13, in the files *Documentation/security/keys.txt* and *Documentation/security/keys−request−key.txt*).

NAME

adjtimex, clock_adjtime, ntp_adjtime − tune kernel clock

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/timex.h>

int adjtimex(struct timex ****buf*);**

int clock_adjtime(clockid_t *clk_id*, **struct timex ****buf***);**

int ntp_adjtime(struct timex ****buf*);**

DESCRIPTION

Linux uses David L. Mills' clock adjustment algorithm (see RFC 5905). The system call **adjtimex**() reads and optionally sets adjustment parameters for this algorithm. It takes a pointer to a *timex* structure, updates kernel parameters from (selected) field values, and returns the same structure updated with the current kernel values. This structure is declared as follows:

```
struct timex {
    int  modes;      /* Mode selector */
    long offset;     /* Time offset; nanoseconds, if STA_NANO
                        status flag is set, otherwise
                        microseconds */
    long freq;       /* Frequency offset; see NOTES for units */
    long maxerror;   /* Maximum error (microseconds) */
    long esterror;   /* Estimated error (microseconds) */
    int  status;     /* Clock command/status */
    long constant;   /* PLL (phase-locked loop) time constant */
    long precision;  /* Clock precision
                        (microseconds, read-only) */
    long tolerance;  /* Clock frequency tolerance (read-only);
                        see NOTES for units */
    struct timeval time;
                     /* Current time (read-only, except for
                        ADJ_SETOFFSET); upon return, time.tv_usec
                        contains nanoseconds, if STA_NANO status
                        flag is set, otherwise microseconds */
    long tick;       /* Microseconds between clock ticks */
    long ppsfreq;    /* PPS (pulse per second) frequency
                        (read-only); see NOTES for units */
    long jitter;     /* PPS jitter (read-only); nanoseconds, if
                        STA_NANO status flag is set, otherwise
                        microseconds */
    int  shift;      /* PPS interval duration
                        (seconds, read-only) */
    long stabil;     /* PPS stability (read-only);
                        see NOTES for units */
    long jitcnt;     /* PPS count of jitter limit exceeded
                        events (read-only) */
    long calcnt;     /* PPS count of calibration intervals
                        (read-only) */
    long errcnt;     /* PPS count of calibration errors
                        (read-only) */
    long stbcnt;     /* PPS count of stability limit exceeded
                        events (read-only) */
    int tai;         /* TAI offset, as set by previous ADJ_TAI
                        operation (seconds, read-only,
                        since Linux 2.6.26) */
    /* Further padding bytes to allow for future expansion */
};
```

The *modes* field determines which parameters, if any, to set. (As described later in this page, the constants used for **ntp_adjtime**() are equivalent but differently named.) It is a bit mask containing a bitwise OR combination of zero or more of the following bits:

ADJ_OFFSET
Set time offset from *buf.offset*. Since Linux 2.6.26, the supplied value is clamped to the range (−0.5s, +0.5s). In older kernels, an **EINVAL** error occurs if the supplied value is out of range.

ADJ_FREQUENCY
Set frequency offset from *buf.freq*. Since Linux 2.6.26, the supplied value is clamped to the range (−32768000, +32768000). In older kernels, an **EINVAL** error occurs if the supplied value is out of range.

ADJ_MAXERROR
Set maximum time error from *buf.maxerror*.

ADJ_ESTERROR
Set estimated time error from *buf.esterror*.

ADJ_STATUS
Set clock status bits from *buf.status*. A description of these bits is provided below.

ADJ_TIMECONST
Set PLL time constant from *buf.constant*. If the **STA_NANO** status flag (see below) is clear, the kernel adds 4 to this value.

ADJ_SETOFFSET (since Linux 2.6.39)
Add *buf.time* to the current time. If *buf.status* includes the **ADJ_NANO** flag, then *buf.time.tv_usec* is interpreted as a nanosecond value; otherwise it is interpreted as microseconds.

The value of *buf.time* is the sum of its two fields, but the field *buf.time.tv_usec* must always be nonnegative. The following example shows how to normalize a *timeval* with nanosecond resolution.

```
while (buf.time.tv_usec < 0) {
    buf.time.tv_sec  -= 1;
    buf.time.tv_usec += 1000000000;
}
```

ADJ_MICRO (since Linux 2.6.26)
Select microsecond resolution.

ADJ_NANO (since Linux 2.6.26)
Select nanosecond resolution. Only one of **ADJ_MICRO** and **ADJ_NANO** should be specified.

ADJ_TAI (since Linux 2.6.26)
Set TAI (Atomic International Time) offset from *buf.constant*.

ADJ_TAI should not be used in conjunction with **ADJ_TIMECONST**, since the latter mode also employs the *buf.constant* field.

For a complete explanation of TAI and the difference between TAI and UTC, see *BIPM*

ADJ_TICK
Set tick value from *buf.tick*.

Alternatively, *modes* can be specified as either of the following (multibit mask) values, in which case other bits should not be specified in *modes*:

ADJ_OFFSET_SINGLESHOT
Old-fashioned *adjtime*(3): (gradually) adjust time by value specified in *buf.offset*, which specifies an adjustment in microseconds.

ADJ_OFFSET_SS_READ (functional since Linux 2.6.28)
Return (in *buf.offset*) the remaining amount of time to be adjusted after an earlier **ADJ_OFFSET_SINGLESHOT** operation. This feature was added in Linux 2.6.24, but did not work correctly until Linux 2.6.28.

Ordinary users are restricted to a value of either 0 or **ADJ_OFFSET_SS_READ** for *modes*. Only the superuser may set any parameters.

The *buf.status* field is a bit mask that is used to set and/or retrieve status bits associated with the NTP implementation. Some bits in the mask are both readable and settable, while others are read-only.

STA_PLL (read-write)
Enable phase-locked loop (PLL) updates via **ADJ_OFFSET**.

STA_PPSFREQ (read-write)
Enable PPS (pulse-per-second) frequency discipline.

STA_PPSTIME (read-write)
Enable PPS time discipline.

STA_FLL (read-write)
Select frequency-locked loop (FLL) mode.

STA_INS (read-write)
Insert a leap second after the last second of the UTC day, thus extending the last minute of the day by one second. Leap-second insertion will occur each day, so long as this flag remains set.

STA_DEL (read-write)
Delete a leap second at the last second of the UTC day. Leap second deletion will occur each day, so long as this flag remains set.

STA_UNSYNC (read-write)
Clock unsynchronized.

STA_FREQHOLD (read-write)
Hold frequency. Normally adjustments made via **ADJ_OFFSET** result in dampened frequency adjustments also being made. So a single call corrects the current offset, but as offsets in the same direction are made repeatedly, the small frequency adjustments will accumulate to fix the long-term skew.

This flag prevents the small frequency adjustment from being made when correcting for an **ADJ_OFFSET** value.

STA_PPSSIGNAL (read-only)
A valid PPS (pulse-per-second) signal is present.

STA_PPSJITTER (read-only)
PPS signal jitter exceeded.

STA_PPSWANDER (read-only)
PPS signal wander exceeded.

STA_PPSERROR (read-only)
PPS signal calibration error.

STA_CLOCKERR (read-only)
Clock hardware fault.

STA_NANO (read-only; since Linux 2.6.26)
Resolution (0 = microsecond, 1 = nanoseconds). Set via **ADJ_NANO**, cleared via **ADJ_MICRO**.

STA_MODE (since Linux 2.6.26)
Mode (0 = Phase Locked Loop, 1 = Frequency Locked Loop).

STA_CLK (read-only; since Linux 2.6.26)
Clock source (0 = A, 1 = B); currently unused.

Attempts to set read-only *status* bits are silently ignored.

clock_adjtime ()
The **clock_adjtime**() system call (added in Linux 2.6.39) behaves like **adjtimex**() but takes an additional *clk_id* argument to specify the particular clock on which to act.

ntp_adjtime ()

The **ntp_adjtime**() library function (described in the NTP "Kernel Application Program API", KAPI) is a more portable interface for performing the same task as **adjtimex**(). Other than the following points, it is identical to **adjtimex**():

- The constants used in *modes* are prefixed with "MOD_" rather than "ADJ_", and have the same suffixes (thus, **MOD_OFFSET**, **MOD_FREQUENCY**, and so on), other than the exceptions noted in the following points.
- **MOD_CLKA** is the synonym for **ADJ_OFFSET_SINGLESHOT**.
- **MOD_CLKB** is the synonym for **ADJ_TICK**.
- The is no synonym for **ADJ_OFFSET_SS_READ**, which is not described in the KAPI.

RETURN VALUE

On success, **adjtimex**() and **ntp_adjtime**() return the clock state; that is, one of the following values:

TIME_OK Clock synchronized, no leap second adjustment pending.

TIME_INS Indicates that a leap second will be added at the end of the UTC day.

TIME_DEL Indicates that a leap second will be deleted at the end of the UTC day.

TIME_OOP Insertion of a leap second is in progress.

TIME_WAIT
A leap-second insertion or deletion has been completed. This value will be returned until the next **ADJ_STATUS** operation clears the **STA_INS** and **STA_DEL** flags.

TIME_ERROR
The system clock is not synchronized to a reliable server. This value is returned when any of the following holds true:

- Either **STA_UNSYNC** or **STA_CLOCKERR** is set.
- **STA_PPSSIGNAL** is clear and either **STA_PPSFREQ** or **STA_PPSTIME** is set.
- **STA_PPSTIME** and **STA_PPSJITTER** are both set.
- **STA_PPSFREQ** is set and either **STA_PPSWANDER** or **STA_PPSJITTER** is set.

The symbolic name **TIME_BAD** is a synonym for **TIME_ERROR**, provided for backward compatibility.

Note that starting with Linux 3.4, the call operates asynchronously and the return value usually will not reflect a state change caused by the call itself.

On failure, these calls return −1 and set *errno* to indicate the error.

ERRORS

EFAULT
buf does not point to writable memory.

EINVAL (before Linux 2.6.26)
An attempt was made to set *buf.freq* to a value outside the range (−33554432, +33554432).

EINVAL (before Linux 2.6.26)
An attempt was made to set *buf.offset* to a value outside the permitted range. Before Linux 2.0, the permitted range was (−131072, +131072). From Linux 2.0 onwards, the permitted range was (−512000, +512000).

EINVAL
An attempt was made to set *buf.status* to a value other than those listed above.

EINVAL
The *clk_id* given to **clock_adjtime**() is invalid for one of two reasons. Either the System-V style hard-coded positive clock ID value is out of range, or the dynamic *clk_id* does not refer to a valid instance of a clock object. See *clock_gettime*(2) for a discussion of dynamic clocks.

EINVAL
An attempt was made to set *buf.tick* to a value outside the range 900000/**HZ** to 1100000/**HZ**, where **HZ** is the system timer interrupt frequency.

ENODEV
The hot-pluggable device (like USB for example) represented by a dynamic *clk_id* has disappeared after its character device was opened. See *clock_gettime*(2) for a discussion of dynamic clocks.

EOPNOTSUPP
The given *clk_id* does not support adjustment.

EPERM
buf.modes is neither 0 nor **ADJ_OFFSET_SS_READ**, and the caller does not have sufficient privilege. Under Linux, the **CAP_SYS_TIME** capability is required.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ntp_adjtime()	Thread safety	MT-Safe

STANDARDS

adjtimex()
clock_adjtime()
Linux.

The preferred API for the NTP daemon is **ntp_adjtime**().

NOTES

In struct *timex*, *freq*, *ppsfreq*, and *stabil* are ppm (parts per million) with a 16-bit fractional part, which means that a value of 1 in one of those fields actually means 2^-16 ppm, and 2^16=65536 is 1 ppm. This is the case for both input values (in the case of *freq*) and output values.

The leap-second processing triggered by **STA_INS** and **STA_DEL** is done by the kernel in timer context. Thus, it will take one tick into the second for the leap second to be inserted or deleted.

SEE ALSO

clock_gettime(2), *clock_settime*(2), *settimeofday*(2), *adjtime*(3), *ntp_gettime*(3), *capabilities*(7), *time*(7), *adjtimex*(8), *hwclock*(8)

NTP "Kernel Application Program Interface"

NAME

alarm − set an alarm clock for delivery of a signal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

unsigned int alarm(unsigned int *seconds***);**

DESCRIPTION

alarm() arranges for a **SIGALRM** signal to be delivered to the calling process in *seconds* seconds.

If *seconds* is zero, any pending alarm is canceled.

In any event any previously set **alarm**() is canceled.

RETURN VALUE

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

alarm() and *setitimer*(2) share the same timer; calls to one will interfere with use of the other.

Alarms created by **alarm**() are preserved across *execve*(2) and are not inherited by children created via *fork*(2).

sleep(3) may be implemented using **SIGALRM**; mixing calls to **alarm**() and *sleep*(3) is a bad idea.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

SEE ALSO

gettimeofday(2), *pause*(2), *select*(2), *setitimer*(2), *sigaction*(2), *signal*(2), *timer_create*(2), *timerfd_create*(2), *sleep*(3), *time*(7)

NAME

alloc_hugepages, free_hugepages − allocate or free huge pages

SYNOPSIS

void *syscall(SYS_alloc_hugepages, int *key***, void** *addr***[.***len***], size_t** *len***,**
int *prot***, int** *flag***);**
int syscall(SYS_free_hugepages, void *****addr*);**

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

The system calls **alloc_hugepages**() and **free_hugepages**() were introduced in Linux 2.5.36 and removed again in Linux 2.5.54. They existed only on i386 and ia64 (when built with **CONFIG_HUGETLB_PAGE**). In Linux 2.4.20, the syscall numbers exist, but the calls fail with the error **ENOSYS**.

On i386 the memory management hardware knows about ordinary pages (4 KiB) and huge pages (2 or 4 MiB). Similarly ia64 knows about huge pages of several sizes. These system calls serve to map huge pages into the process's memory or to free them again. Huge pages are locked into memory, and are not swapped.

The *key* argument is an identifier. When zero the pages are private, and not inherited by children. When positive the pages are shared with other applications using the same *key*, and inherited by child processes.

The *addr* argument of **free_hugepages**() tells which page is being freed: it was the return value of a call to **alloc_hugepages**(). (The memory is first actually freed when all users have released it.) The *addr* argument of **alloc_hugepages**() is a hint, that the kernel may or may not follow. Addresses must be properly aligned.

The *len* argument is the length of the required segment. It must be a multiple of the huge page size.

The *prot* argument specifies the memory protection of the segment. It is one of **PROT_READ**, **PROT_WRITE**, **PROT_EXEC**.

The *flag* argument is ignored, unless *key* is positive. In that case, if *flag* is **IPC_CREAT**, then a new huge page segment is created when none with the given key existed. If this flag is not set, then **ENOENT** is returned when no segment with the given key exists.

RETURN VALUE

On success, **alloc_hugepages**() returns the allocated virtual address, and **free_hugepages**() returns zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

ENOSYS
The system call is not supported on this kernel.

FILES

/proc/sys/vm/nr_hugepages
Number of configured hugetlb pages. This can be read and written.

/proc/meminfo
Gives info on the number of configured hugetlb pages and on their size in the three variables HugePages_Total, HugePages_Free, Hugepagesize.

STANDARDS

Linux on Intel processors.

HISTORY

These system calls are gone; they existed only in Linux 2.5.36 through to Linux 2.5.54.

NOTES

Now the hugetlbfs filesystem can be used instead. Memory backed by huge pages (if the CPU supports them) is obtained by using *mmap*(2) to map files in this virtual filesystem.

The maximal number of huge pages can be specified using the **hugepages=** boot parameter.

NAME

arch_prctl − set architecture-specific thread state

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <asm/prctl.h>      /* Definition of ARCH_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_arch_prctl, int op, unsigned long addr);
int syscall(SYS_arch_prctl, int op, unsigned long *addr);
```

Note: glibc provides no wrapper for **arch_prctl**(), necessitating the use of *syscall*(2).

DESCRIPTION

arch_prctl() sets architecture-specific process or thread state. *op* selects an operation and passes argument *addr* to it; *addr* is interpreted as either an *unsigned long* for the "set" operations, or as an *unsigned long* *, for the "get" operations.

Subfunctions for both x86 and x86-64 are:

ARCH_SET_CPUID (since Linux 4.12)
: Enable (*addr != 0*) or disable (*addr == 0*) the *cpuid* instruction for the calling thread. The instruction is enabled by default. If disabled, any execution of a *cpuid* instruction will instead generate a **SIGSEGV** signal. This feature can be used to emulate *cpuid* results that differ from what the underlying hardware would have produced (e.g., in a paravirtualization setting).

  The **ARCH_SET_CPUID** setting is preserved across *fork*(2) and *clone*(2) but reset to the default (i.e., *cpuid* enabled) on *execve*(2).

ARCH_GET_CPUID (since Linux 4.12)
: Return the setting of the flag manipulated by **ARCH_SET_CPUID** as the result of the system call (1 for enabled, 0 for disabled). *addr* is ignored.

Subfunctions for x86-64 only are:

ARCH_SET_FS
: Set the 64-bit base for the *FS* register to *addr*.

ARCH_GET_FS
: Return the 64-bit base value for the *FS* register of the calling thread in the *unsigned long* pointed to by *addr*.

ARCH_SET_GS
: Set the 64-bit base for the *GS* register to *addr*.

ARCH_GET_GS
: Return the 64-bit base value for the *GS* register of the calling thread in the *unsigned long* pointed to by *addr*.

RETURN VALUE

On success, **arch_prctl**() returns 0; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: *addr* points to an unmapped address or is outside the process address space.

EINVAL
: *op* is not a valid operation.

ENODEV
: **ARCH_SET_CPUID** was requested, but the underlying hardware does not support CPUID faulting.

EPERM
: *addr* is outside the process address space.

STANDARDS

Linux/x86-64.

NOTES

arch_prctl() is supported only on Linux/x86-64 for 64-bit programs currently.

The 64-bit base changes when a new 32-bit segment selector is loaded.

ARCH_SET_GS is disabled in some kernels.

Context switches for 64-bit segment bases are rather expensive. As an optimization, if a 32-bit TLS base address is used, **arch_prctl**() may use a real TLS entry as if *set_thread_area*(2) had been called, instead of manipulating the segment base register directly. Memory in the first 2 GB of address space can be allocated by using *mmap*(2) with the **MAP_32BIT** flag.

Because of the aforementioned optimization, using **arch_prctl**() and *set_thread_area*(2) in the same thread is dangerous, as they may overwrite each other's TLS entries.

FS may be already used by the threading library. Programs that use **ARCH_SET_FS** directly are very likely to crash.

SEE ALSO

mmap(2), *modify_ldt*(2), *prctl*(2), *set_thread_area*(2)

AMD X86-64 Programmer's manual

NAME

bdflush − start, flush, or tune buffer-dirty-flush daemon

SYNOPSIS

#include <sys/kdaemon.h>

[[deprecated]] int bdflush(int *func***, long ****address***);**
[[deprecated]] int bdflush(int *func***, long** *data***);**

DESCRIPTION

Note: Since Linux 2.6, this system call is deprecated and does nothing. It is likely to disappear altogether in a future kernel release. Nowadays, the task performed by **bdflush**() is handled by the kernel *pdflush* thread.

bdflush() starts, flushes, or tunes the buffer-dirty-flush daemon. Only a privileged process (one with the **CAP_SYS_ADMIN** capability) may call **bdflush**().

If *func* is negative or 0, and no daemon has been started, then **bdflush**() enters the daemon code and never returns.

If *func* is 1, some dirty buffers are written to disk.

If *func* is 2 or more and is even (low bit is 0), then *address* is the address of a long word, and the tuning parameter numbered (*func*−2)/2 is returned to the caller in that address.

If *func* is 3 or more and is odd (low bit is 1), then *data* is a long word, and the kernel sets tuning parameter numbered (*func*−3)/2 to that value.

The set of parameters, their values, and their valid ranges are defined in the Linux kernel source file *fs/buffer.c*.

RETURN VALUE

If *func* is negative or 0 and the daemon successfully starts, **bdflush**() never returns. Otherwise, the return value is 0 on success and −1 on failure, with *errno* set to indicate the error.

ERRORS

EBUSY
An attempt was made to enter the daemon code after another process has already entered.

EFAULT
address points outside your accessible address space.

EINVAL
An attempt was made to read or write an invalid parameter number, or to write an invalid value to a parameter.

EPERM
Caller does not have the **CAP_SYS_ADMIN** capability.

STANDARDS

Linux.

HISTORY

Since glibc 2.23, glibc no longer supports this obsolete system call.

SEE ALSO

sync(1), *fsync*(2), *sync*(2)

NAME

bind − bind a name to a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

DESCRIPTION

When a socket is created with *socket*(2), it exists in a name space (address family) but has no address assigned to it. **bind**() assigns the address specified by *addr* to the socket referred to by the file descriptor *sockfd*. *addrlen* specifies the size, in bytes, of the address structure pointed to by *addr*. Traditionally, this operation is called “assigning a name to a socket”.

It is normally necessary to assign a local address using **bind**() before a **SOCK_STREAM** socket may receive connections (see *accept*(2)).

The rules used in name binding vary between address families. Consult the manual entries in Section 7 for detailed information. For **AF_INET**, see *ip*(7); for **AF_INET6**, see *ipv6*(7); for **AF_UNIX**, see *unix*(7); for **AF_APPLETALK**, see *ddp*(7); for **AF_PACKET**, see *packet*(7); for **AF_X25**, see *x25*(7); and for **AF_NETLINK**, see *netlink*(7).

The actual structure passed for the *addr* argument will depend on the address family. The *sockaddr* structure is defined as something like:

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

The only purpose of this structure is to cast the structure pointer passed in *addr* in order to avoid compiler warnings. See EXAMPLES below.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
The address is protected, and the user is not the superuser.

EADDRINUSE
The given address is already in use.

EADDRINUSE
(Internet domain sockets) The port number was specified as zero in the socket address structure, but, upon attempting to bind to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of */proc/sys/net/ipv4/ip_local_port_range* *ip*(7).

EBADF
sockfd is not a valid file descriptor.

EINVAL
The socket is already bound to an address.

EINVAL
addrlen is wrong, or *addr* is not a valid address for this socket’s domain.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

The following errors are specific to UNIX domain (**AF_UNIX**) sockets:

EACCES
Search permission is denied on a component of the path prefix. (See also *path_resolution*(7).)

EADDRNOTAVAIL
A nonexistent interface was requested or the requested address was not local.

EFAULT
addr points outside the user's accessible address space.

ELOOP
Too many symbolic links were encountered in resolving *addr*.

ENAMETOOLONG
addr is too long.

ENOENT
A component in the directory prefix of the socket pathname does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

EROFS
The socket inode would reside on a read-only filesystem.

STANDARDS
POSIX.1-2008.

HISTORY
POSIX.1-2001, SVr4, 4.4BSD (**bind**() first appeared in 4.2BSD).

BUGS
The transparent proxy options are not described.

EXAMPLES
An example of the use of **bind**() with Internet domain sockets can be found in *getaddrinfo*(3).

The following example shows how to bind a stream socket in the UNIX (**AF_UNIX**) domain, and accept connections:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define MY_SOCK_PATH "/somepath"
#define LISTEN_BACKLOG 50

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(void)
{
    int                 sfd, cfd;
    socklen_t           peer_addr_size;
    struct sockaddr_un  my_addr, peer_addr;

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        handle_error("socket");

    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sun_family = AF_UNIX;
    strncpy(my_addr.sun_path, MY_SOCK_PATH,
```

```
                sizeof(my_addr.sun_path) - 1);

        if (bind(sfd, (struct sockaddr *) &my_addr,
                 sizeof(my_addr)) == -1)
            handle_error("bind");

        if (listen(sfd, LISTEN_BACKLOG) == -1)
            handle_error("listen");

        /* Now we can accept incoming connections one
           at a time using accept(2). */

        peer_addr_size = sizeof(peer_addr);
        cfd = accept(sfd, (struct sockaddr *) &peer_addr,
                     &peer_addr_size);
        if (cfd == -1)
            handle_error("accept");

        /* Code to deal with incoming connection(s)... */

        if (close(sfd) == -1)
            handle_error("close");

        if (unlink(MY_SOCK_PATH) == -1)
            handle_error("unlink");
    }
```

SEE ALSO

accept(2), *connect*(2), *getsockname*(2), *listen*(2), *socket*(2), *getaddrinfo*(3), *getifaddrs*(3), *ip*(7), *ipv6*(7), *path_resolution*(7), *socket*(7), *unix*(7)

NAME

bpf − perform a command on an extended BPF map or program

SYNOPSIS

#include <linux/bpf.h>

int bpf(int *cmd***, union bpf_attr ****attr***, unsigned int** *size***);**

DESCRIPTION

The **bpf**() system call performs a range of operations related to extended Berkeley Packet Filters. Extended BPF (or eBPF) is similar to the original ("classic") BPF (cBPF) used to filter network packets. For both cBPF and eBPF programs, the kernel statically analyzes the programs before loading them, in order to ensure that they cannot harm the running system.

eBPF extends cBPF in multiple ways, including the ability to call a fixed set of in-kernel helper functions (via the **BPF_CALL** opcode extension provided by eBPF) and access shared data structures such as eBPF maps.

Extended BPF Design/Architecture

eBPF maps are a generic data structure for storage of different data types. Data types are generally treated as binary blobs, so a user just specifies the size of the key and the size of the value at map-creation time. In other words, a key/value for a given map can have an arbitrary structure.

A user process can create multiple maps (with key/value-pairs being opaque bytes of data) and access them via file descriptors. Different eBPF programs can access the same maps in parallel. It's up to the user process and eBPF program to decide what they store inside maps.

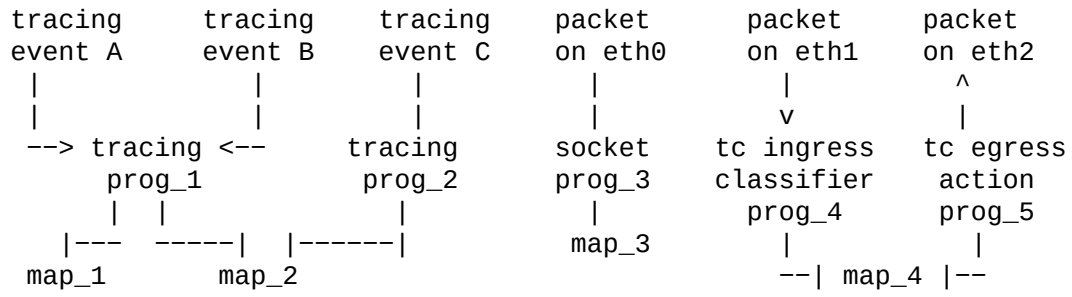
There's one special map type, called a program array. This type of map stores file descriptors referring to other eBPF programs. When a lookup in the map is performed, the program flow is redirected in-place to the beginning of another eBPF program and does not return back to the calling program. The level of nesting has a fixed limit of 32, so that infinite loops cannot be crafted. At run time, the program file descriptors stored in the map can be modified, so program functionality can be altered based on specific requirements. All programs referred to in a program-array map must have been previously loaded into the kernel via **bpf**(). If a map lookup fails, the current program continues its execution. See **BPF_MAP_TYPE_PROG_ARRAY** below for further details.

Generally, eBPF programs are loaded by the user process and automatically unloaded when the process exits. In some cases, for example, *tc-bpf*(8), the program will continue to stay alive inside the kernel even after the process that loaded the program exits. In that case, the tc subsystem holds a reference to the eBPF program after the file descriptor has been closed by the user-space program. Thus, whether a specific program continues to live inside the kernel depends on how it is further attached to a given kernel subsystem after it was loaded via **bpf**().

Each eBPF program is a set of instructions that is safe to run until its completion. An in-kernel verifier statically determines that the eBPF program terminates and is safe to execute. During verification, the kernel increments reference counts for each of the maps that the eBPF program uses, so that the attached maps can't be removed until the program is unloaded.

eBPF programs can be attached to different events. These events can be the arrival of network packets, tracing events, classification events by network queueing disciplines (for eBPF programs attached to a *tc*(8) classifier), and other types that may be added in the future. A new event triggers execution of the eBPF program, which may store information about the event in eBPF maps. Beyond storing data, eBPF programs may call a fixed set of in-kernel helper functions.

The same eBPF program can be attached to multiple events and different eBPF programs can access the same map:



Arguments

The operation to be performed by the **bpf**() system call is determined by the *cmd* argument. Each operation takes an accompanying argument, provided via *attr*, which is a pointer to a union of type *bpf_attr* (see below). The unused fields and padding must be zeroed out before the call. The *size* argument is the size of the union pointed to by *attr*.

The value provided in *cmd* is one of the following:

BPF_MAP_CREATE
Create a map and return a file descriptor that refers to the map. The close-on-exec file descriptor flag (see *fcntl*(2)) is automatically enabled for the new file descriptor.

BPF_MAP_LOOKUP_ELEM
Look up an element by key in a specified map and return its value.

BPF_MAP_UPDATE_ELEM
Create or update an element (key/value pair) in a specified map.

BPF_MAP_DELETE_ELEM
Look up and delete an element by key in a specified map.

BPF_MAP_GET_NEXT_KEY
Look up an element by key in a specified map and return the key of the next element.

BPF_PROG_LOAD
Verify and load an eBPF program, returning a new file descriptor associated with the program. The close-on-exec file descriptor flag (see *fcntl*(2)) is automatically enabled for the new file descriptor.

The *bpf_attr* union consists of various anonymous structures that are used by different **bpf**() commands:

```
union bpf_attr {
    struct {    /* Used by BPF_MAP_CREATE */
        __u32         map_type;
        __u32         key_size;    /* size of key in bytes */
        __u32         value_size;  /* size of value in bytes */
        __u32         max_entries; /* maximum number of entries
                                      in a map */
    };

    struct {    /* Used by BPF_MAP_*_ELEM and BPF_MAP_GET_NEXT_KEY
                   commands */
        __u32         map_fd;
        __aligned_u64 key;
        union {
            __aligned_u64 value;
            __aligned_u64 next_key;
        };
        __u64         flags;
    };

    struct {    /* Used by BPF_PROG_LOAD */
        __u32         prog_type;
        __u32         insn_cnt;
        __aligned_u64 insns;      /* 'const struct bpf_insn *' */
        __aligned_u64 license;    /* 'const char *' */
        __u32         log_level;  /* verbosity level of verifier */
        __u32         log_size;   /* size of user buffer */
        __aligned_u64 log_buf;    /* user supplied 'char *'
                                     buffer */
        __u32         kern_version;
                                  /* checked when prog_type=kprobe
                                     (since Linux 4.1) */
```

```
        };
    } __attribute__((aligned(8)));
```

eBPF maps

Maps are a generic data structure for storage of different types of data. They allow sharing of data between eBPF kernel programs, and also between kernel and user-space applications.

Each map type has the following attributes:

- type
- maximum number of elements
- key size in bytes
- value size in bytes

The following wrapper functions demonstrate how various **bpf**() commands can be used to access the maps. The functions use the *cmd* argument to invoke different operations.

BPF_MAP_CREATE

The **BPF_MAP_CREATE** command creates a new map, returning a new file descriptor that refers to the map.

```
int
bpf_create_map(enum bpf_map_type map_type,
               unsigned int key_size,
               unsigned int value_size,
               unsigned int max_entries)
{
    union bpf_attr attr = {
        .map_type    = map_type,
        .key_size    = key_size,
        .value_size  = value_size,
        .max_entries = max_entries
    };

    return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
}
```

The new map has the type specified by *map_type*, and attributes as specified in *key_size*, *value_size*, and *max_entries*. On success, this operation returns a file descriptor. On error, −1 is returned and *errno* is set to **EINVAL**, **EPERM**, or **ENOMEM**.

The *key_size* and *value_size* attributes will be used by the verifier during program loading to check that the program is calling **bpf_map_*_elem**() helper functions with a correctly initialized *key* and to check that the program doesn't access the map element *value* beyond the specified *value_size*. For example, when a map is created with a *key_size* of 8 and the eBPF program calls

```
bpf_map_lookup_elem(map_fd, fp - 4)
```

the program will be rejected, since the in-kernel helper function

```
bpf_map_lookup_elem(map_fd, void *key)
```

expects to read 8 bytes from the location pointed to by *key*, but the *fp − 4* (where *fp* is the top of the stack) starting address will cause out-of-bounds stack access.

Similarly, when a map is created with a *value_size* of 1 and the eBPF program contains

```
value = bpf_map_lookup_elem(...);
*(u32 *) value = 1;
```

the program will be rejected, since it accesses the *value* pointer beyond the specified 1 byte *value_size* limit.

Currently, the following values are supported for *map_type*:

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,  /* Reserve 0 as invalid map type */
```

```
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        /* See /usr/include/linux/bpf.h for the full list. */
    };
```

map_type selects one of the available map implementations in the kernel. For all map types, eBPF programs access maps with the same **bpf_map_lookup_elem**() and **bpf_map_update_elem**() helper functions. Further details of the various map types are given below.

BPF_MAP_LOOKUP_ELEM

The **BPF_MAP_LOOKUP_ELEM** command looks up an element with a given *key* in the map referred to by the file descriptor *fd*.

```
    int
    bpf_lookup_elem(int fd, const void *key, void *value)
    {
        union bpf_attr attr = {
            .map_fd = fd,
            .key    = ptr_to_u64(key),
            .value  = ptr_to_u64(value),
        };

        return bpf(BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr));
    }
```

If an element is found, the operation returns zero and stores the element's value into *value*, which must point to a buffer of *value_size* bytes.

If no element is found, the operation returns −1 and sets *errno* to **ENOENT**.

BPF_MAP_UPDATE_ELEM

The **BPF_MAP_UPDATE_ELEM** command creates or updates an element with a given *key/value* in the map referred to by the file descriptor *fd*.

```
    int
    bpf_update_elem(int fd, const void *key, const void *value,
                    uint64_t flags)
    {
        union bpf_attr attr = {
            .map_fd = fd,
            .key    = ptr_to_u64(key),
            .value  = ptr_to_u64(value),
```

```
        .flags  = flags,
    };

    return bpf(BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr));
}
```

The *flags* argument should be specified as one of the following:

BPF_ANY
Create a new element or update an existing element.

BPF_NOEXIST
Create a new element only if it did not exist.

BPF_EXIST
Update an existing element.

On success, the operation returns zero. On error, −1 is returned and *errno* is set to **EINVAL**, **EPERM**, **ENOMEM**, or **E2BIG**. **E2BIG** indicates that the number of elements in the map reached the *max_entries* limit specified at map creation time. **EEXIST** will be returned if *flags* specifies **BPF_NOEXIST** and the element with *key* already exists in the map. **ENOENT** will be returned if *flags* specifies **BPF_EXIST** and the element with *key* doesn't exist in the map.

BPF_MAP_DELETE_ELEM
The **BPF_MAP_DELETE_ELEM** command deletes the element whose key is *key* from the map referred to by the file descriptor *fd*.

```
int
bpf_delete_elem(int fd, const void *key)
{
    union bpf_attr attr = {
        .map_fd = fd,
        .key    = ptr_to_u64(key),
    };

    return bpf(BPF_MAP_DELETE_ELEM, &attr, sizeof(attr));
}
```

On success, zero is returned. If the element is not found, −1 is returned and *errno* is set to **ENOENT**.

BPF_MAP_GET_NEXT_KEY
The **BPF_MAP_GET_NEXT_KEY** command looks up an element by *key* in the map referred to by the file descriptor *fd* and sets the *next_key* pointer to the key of the next element.

```
int
bpf_get_next_key(int fd, const void *key, void *next_key)
{
    union bpf_attr attr = {
        .map_fd   = fd,
        .key      = ptr_to_u64(key),
        .next_key = ptr_to_u64(next_key),
    };

    return bpf(BPF_MAP_GET_NEXT_KEY, &attr, sizeof(attr));
}
```

If *key* is found, the operation returns zero and sets the *next_key* pointer to the key of the next element. If *key* is not found, the operation returns zero and sets the *next_key* pointer to the key of the first element. If *key* is the last element, −1 is returned and *errno* is set to **ENOENT**. Other possible *errno* values are **ENOMEM**, **EFAULT**, **EPERM**, and **EINVAL**. This method can be used to iterate over all elements in the map.

close(map_fd)
Delete the map referred to by the file descriptor *map_fd*. When the user-space program that created a map exits, all maps will be deleted automatically (but see NOTES).

eBPF map types
The following map types are supported:

BPF_MAP_TYPE_HASH
Hash-table maps have the following characteristics:

- Maps are created and destroyed by user-space programs. Both user-space and eBPF programs can perform lookup, update, and delete operations.
- The kernel takes care of allocating and freeing key/value pairs.
- The **map_update_elem**() helper will fail to insert new element when the *max_entries* limit is reached. (This ensures that eBPF programs cannot exhaust memory.)
- **map_update_elem**() replaces existing elements atomically.

Hash-table maps are optimized for speed of lookup.

BPF_MAP_TYPE_ARRAY
Array maps have the following characteristics:

- Optimized for fastest possible lookup. In the future the verifier/JIT compiler may recognize lookup() operations that employ a constant key and optimize it into constant pointer. It is possible to optimize a non-constant key into direct pointer arithmetic as well, since pointers and *value_size* are constant for the life of the eBPF program. In other words, **array_map_lookup_elem**() may be 'inlined' by the verifier/JIT compiler while preserving concurrent access to this map from user space.
- All array elements pre-allocated and zero initialized at init time
- The key is an array index, and must be exactly four bytes.
- **map_delete_elem**() fails with the error **EINVAL**, since elements cannot be deleted.
- **map_update_elem**() replaces elements in a **nonatomic** fashion; for atomic updates, a hash-table map should be used instead. There is however one special case that can also be used with arrays: the atomic built-in **__sync_fetch_and_add()** can be used on 32 and 64 bit atomic counters. For example, it can be applied on the whole value itself if it represents a single counter, or in case of a structure containing multiple counters, it could be used on individual counters. This is quite often useful for aggregation and accounting of events.

Among the uses for array maps are the following:

- As "global" eBPF variables: an array of 1 element whose key is (index) 0 and where the value is a collection of 'global' variables which eBPF programs can use to keep state between events.
- Aggregation of tracing events into a fixed set of buckets.
- Accounting of networking events, for example, number of packets and packet sizes.

BPF_MAP_TYPE_PROG_ARRAY (since Linux 4.2)
A program array map is a special kind of array map whose map values contain only file descriptors referring to other eBPF programs. Thus, both the *key_size* and *value_size* must be exactly four bytes. This map is used in conjunction with the **bpf_tail_call**() helper.

This means that an eBPF program with a program array map attached to it can call from kernel side into

```
void bpf_tail_call(void *context, void *prog_map,
                   unsigned int index);
```

and therefore replace its own program flow with the one from the program at the given program array slot, if present. This can be regarded as kind of a jump table to a different eBPF program. The invoked program will then reuse the same stack. When a jump into the new program has been performed, it won't return to the old program anymore.

If no eBPF program is found at the given index of the program array (because the map slot doesn't contain a valid program file descriptor, the specified lookup index/key is out of bounds, or the limit of 32 nested calls has been exceed), execution continues with the current eBPF program. This can be used as a fall-through for default cases.

A program array map is useful, for example, in tracing or networking, to handle individual system calls or protocols in their own subprograms and use their identifiers as an individual map index. This approach may result in performance benefits, and also makes it possible to overcome the maximum instruction limit of a single eBPF program. In dynamic environments, a user-space daemon might atomically replace individual subprograms at run-time with newer versions to alter overall program behavior, for instance, if global policies change.

eBPF programs

The **BPF_PROG_LOAD** command is used to load an eBPF program into the kernel. The return value for this command is a new file descriptor associated with this eBPF program.

```
char bpf_log_buf[LOG_BUF_SIZE];

int
bpf_prog_load(enum bpf_prog_type type,
              const struct bpf_insn *insns, int insn_cnt,
              const char *license)
{
    union bpf_attr attr = {
        .prog_type = type,
        .insns     = ptr_to_u64(insns),
        .insn_cnt  = insn_cnt,
        .license   = ptr_to_u64(license),
        .log_buf   = ptr_to_u64(bpf_log_buf),
        .log_size  = LOG_BUF_SIZE,
        .log_level = 1,
    };

    return bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
}
```

prog_type is one of the available program types:

```
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,        /* Reserve 0 as invalid
                                    program type */
    BPF_PROG_TYPE_SOCKET_FILTER,
    BPF_PROG_TYPE_KPROBE,
    BPF_PROG_TYPE_SCHED_CLS,
    BPF_PROG_TYPE_SCHED_ACT,
    BPF_PROG_TYPE_TRACEPOINT,
    BPF_PROG_TYPE_XDP,
    BPF_PROG_TYPE_PERF_EVENT,
    BPF_PROG_TYPE_CGROUP_SKB,
    BPF_PROG_TYPE_CGROUP_SOCK,
    BPF_PROG_TYPE_LWT_IN,
    BPF_PROG_TYPE_LWT_OUT,
    BPF_PROG_TYPE_LWT_XMIT,
    BPF_PROG_TYPE_SOCK_OPS,
    BPF_PROG_TYPE_SK_SKB,
    BPF_PROG_TYPE_CGROUP_DEVICE,
    BPF_PROG_TYPE_SK_MSG,
    BPF_PROG_TYPE_RAW_TRACEPOINT,
    BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
    BPF_PROG_TYPE_LWT_SEG6LOCAL,
    BPF_PROG_TYPE_LIRC_MODE2,
    BPF_PROG_TYPE_SK_REUSEPORT,
```

```
        BPF_PROG_TYPE_FLOW_DISSECTOR,
        /* See /usr/include/linux/bpf.h for the full list. */
};
```

For further details of eBPF program types, see below.

The remaining fields of *bpf_attr* are set as follows:

- *insns* is an array of *struct bpf_insn* instructions.
- *insn_cnt* is the number of instructions in the program referred to by *insns*.
- *license* is a license string, which must be GPL compatible to call helper functions marked *gpl_only*. (The licensing rules are the same as for kernel modules, so that also dual licenses, such as "Dual BSD/GPL", may be used.)
- *log_buf* is a pointer to a caller-allocated buffer in which the in-kernel verifier can store the verification log. This log is a multi-line string that can be checked by the program author in order to understand how the verifier came to the conclusion that the eBPF program is unsafe. The format of the output can change at any time as the verifier evolves.
- *log_size* size of the buffer pointed to by *log_buf*. If the size of the buffer is not large enough to store all verifier messages, −1 is returned and *errno* is set to **ENOSPC**.
- *log_level* verbosity level of the verifier. A value of zero means that the verifier will not provide a log; in this case, *log_buf* must be a null pointer, and *log_size* must be zero.

Applying *close*(2) to the file descriptor returned by **BPF_PROG_LOAD** will unload the eBPF program (but see NOTES).

Maps are accessible from eBPF programs and are used to exchange data between eBPF programs and between eBPF programs and user-space programs. For example, eBPF programs can process various events (like kprobe, packets) and store their data into a map, and user-space programs can then fetch data from the map. Conversely, user-space programs can use a map as a configuration mechanism, populating the map with values checked by the eBPF program, which then modifies its behavior on the fly according to those values.

eBPF program types

The eBPF program type (*prog_type*) determines the subset of kernel helper functions that the program may call. The program type also determines the program input (context)—the format of *struct bpf_context* (which is the data blob passed into the eBPF program as the first argument).

For example, a tracing program does not have the exact same subset of helper functions as a socket filter program (though they may have some helpers in common). Similarly, the input (context) for a tracing program is a set of register values, while for a socket filter it is a network packet.

The set of functions available to eBPF programs of a given type may increase in the future.

The following program types are supported:

BPF_PROG_TYPE_SOCKET_FILTER (since Linux 3.19)

Currently, the set of functions for **BPF_PROG_TYPE_SOCKET_FILTER** is:

```
bpf_map_lookup_elem(map_fd, void *key)
                    /* look up key in a map_fd */
bpf_map_update_elem(map_fd, void *key, void *value)
                    /* update key/value */
bpf_map_delete_elem(map_fd, void *key)
                    /* delete key in a map_fd */
```

The *bpf_context* argument is a pointer to a *struct __sk_buff*.

BPF_PROG_TYPE_KPROBE (since Linux 4.1)

[To be documented]

BPF_PROG_TYPE_SCHED_CLS (since Linux 4.1)

[To be documented]

BPF_PROG_TYPE_SCHED_ACT (since Linux 4.1)

[To be documented]

Events

Once a program is loaded, it can be attached to an event. Various kernel subsystems have different ways to do so.

Since Linux 3.19, the following call will attach the program *prog_fd* to the socket *sockfd*, which was created by an earlier call to *socket*(2):

```
setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_BPF,
           &prog_fd, sizeof(prog_fd));
```

Since Linux 4.1, the following call may be used to attach the eBPF program referred to by the file descriptor *prog_fd* to a perf event file descriptor, *event_fd*, that was created by a previous call to *perf_event_open*(2):

```
ioctl(event_fd, PERF_EVENT_IOC_SET_BPF, prog_fd);
```

RETURN VALUE

For a successful call, the return value depends on the operation:

BPF_MAP_CREATE
: The new file descriptor associated with the eBPF map.

BPF_PROG_LOAD
: The new file descriptor associated with the eBPF program.

All other commands
: Zero.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

E2BIG The eBPF program is too large or a map reached the *max_entries* limit (maximum number of elements).

EACCES
: For **BPF_PROG_LOAD**, even though all program instructions are valid, the program has been rejected because it was deemed unsafe. This may be because it may have accessed a disallowed memory region or an uninitialized stack/register or because the function constraints don't match the actual types or because there was a misaligned memory access. In this case, it is recommended to call **bpf**() again with *log_level = 1* and examine *log_buf* for the specific reason provided by the verifier.

EAGAIN
: For **BPF_PROG_LOAD**, indicates that needed resources are blocked. This happens when the verifier detects pending signals while it is checking the validity of the bpf program. In this case, just call **bpf**() again with the same parameters.

EBADF
: *fd* is not an open file descriptor.

EFAULT
: One of the pointers (*key* or *value* or *log_buf* or *insns*) is outside the accessible address space.

EINVAL
: The value specified in *cmd* is not recognized by this kernel.

EINVAL
: For **BPF_MAP_CREATE**, either *map_type* or attributes are invalid.

EINVAL
: For **BPF_MAP_*_ELEM** commands, some of the fields of *union bpf_attr* that are not used by this command are not set to zero.

EINVAL
: For **BPF_PROG_LOAD**, indicates an attempt to load an invalid program. eBPF programs can be deemed invalid due to unrecognized instructions, the use of reserved fields, jumps out of range, infinite loops or calls of unknown functions.

ENOENT
For **BPF_MAP_LOOKUP_ELEM** or **BPF_MAP_DELETE_ELEM**, indicates that the element with the given *key* was not found.

ENOMEM
Cannot allocate sufficient memory.

EPERM
The call was made without sufficient privilege (without the **CAP_SYS_ADMIN** capability).

STANDARDS
Linux.

HISTORY
Linux 3.18.

NOTES
Prior to Linux 4.4, all **bpf**() commands require the caller to have the **CAP_SYS_ADMIN** capability. From Linux 4.4 onwards, an unprivileged user may create limited programs of type **BPF_PROG_TYPE_SOCKET_FILTER** and associated maps. However they may not store kernel pointers within the maps and are presently limited to the following helper functions:

- get_random
- get_smp_processor_id
- tail_call
- ktime_get_ns

Unprivileged access may be blocked by writing the value 1 to the file */proc/sys/kernel/unprivileged_bpf_disabled*.

eBPF objects (maps and programs) can be shared between processes. For example, after *fork*(2), the child inherits file descriptors referring to the same eBPF objects. In addition, file descriptors referring to eBPF objects can be transferred over UNIX domain sockets. File descriptors referring to eBPF objects can be duplicated in the usual way, using *dup*(2) and similar calls. An eBPF object is deallocated only after all file descriptors referring to the object have been closed.

eBPF programs can be written in a restricted C that is compiled (using the **clang** compiler) into eBPF bytecode. Various features are omitted from this restricted C, such as loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments. Some examples can be found in the *samples/bpf/*_kern.c* files in the kernel source tree.

The kernel contains a just-in-time (JIT) compiler that translates eBPF bytecode into native machine code for better performance. Before Linux 4.15, the JIT compiler is disabled by default, but its operation can be controlled by writing one of the following integer strings to the file */proc/sys/net/core/bpf_jit_enable*:

0 Disable JIT compilation (default).

1 Normal compilation.

2 Debugging mode. The generated opcodes are dumped in hexadecimal into the kernel log. These opcodes can then be disassembled using the program *tools/net/bpf_jit_disasm.c* provided in the kernel source tree.

Since Linux 4.15, the kernel may be configured with the **CONFIG_BPF_JIT_ALWAYS_ON** option. In this case, the JIT compiler is always enabled, and the *bpf_jit_enable* is initialized to 1 and is immutable. (This kernel configuration option was provided as a mitigation for one of the Spectre attacks against the BPF interpreter.)

The JIT compiler for eBPF is currently available for the following architectures:

- x86-64 (since Linux 3.18; cBPF since Linux 3.0);
- ARM32 (since Linux 3.18; cBPF since Linux 3.4);
- SPARC 32 (since Linux 3.18; cBPF since Linux 3.5);
- ARM-64 (since Linux 3.18);
- s390 (since Linux 4.1; cBPF since Linux 3.7);

- PowerPC 64 (since Linux 4.8; cBPF since Linux 3.1);
- SPARC 64 (since Linux 4.12);
- x86-32 (since Linux 4.18);
- MIPS 64 (since Linux 4.18; cBPF since Linux 3.16);
- riscv (since Linux 5.1).

EXAMPLES

```
/* bpf+sockets example:
 * 1. create array map of 256 elements
 * 2. load program that counts number of packets received
 *    r0 = skb->data[ETH_HLEN + offsetof(struct iphdr, protocol)]
 *    map[r0]++
 * 3. attach prog_fd to raw socket via setsockopt()
 * 4. print number of received TCP/UDP packets every second
 */
int
main(int argc, char *argv[])
{
    int sock, map_fd, prog_fd, key;
    long long value = 0, tcp_cnt, udp_cnt;

    map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key),
                            sizeof(value), 256);
    if (map_fd < 0) {
        printf("failed to create map '%s'\n", strerror(errno));
        /* likely not run as root */
        return 1;
    }

    struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),        /* r6 = r1 */
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol)),
                                /* r0 = ip->proto */
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4),
                                /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),       /* r2 = fp */
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),      /* r2 = r2 - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),           /* r1 = map_fd */
        BPF_CALL_FUNC(BPF_FUNC_map_lookup_elem),
                                /* r0 = map_lookup(r1, r2) */
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
                                /* if (r0 == 0) goto pc+2 */
        BPF_MOV64_IMM(BPF_REG_1, 1),                /* r1 = 1 */
        BPF_XADD(BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0),
                                /* lock *(u64 *) r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0),                /* r0 = 0 */
        BPF_EXIT_INSN(),                            /* return r0 */
    };

    prog_fd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER, prog,
                            sizeof(prog) / sizeof(prog[0]), "GPL");

    sock = open_raw_sock("lo");

    assert(setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
                      sizeof(prog_fd)) == 0);

    for (;;) {
        key = IPPROTO_TCP;
```

```
            assert(bpf_lookup_elem(map_fd, &key, &tcp_cnt) == 0);
            key = IPPROTO_UDP;
            assert(bpf_lookup_elem(map_fd, &key, &udp_cnt) == 0);
            printf("TCP %lld UDP %lld packets\n", tcp_cnt, udp_cnt);
            sleep(1);
        }

        return 0;
    }
```

Some complete working code can be found in the *samples/bpf* directory in the kernel source tree.

SEE ALSO

seccomp(2), *bpf-helpers*(7), *socket*(7), *tc*(8), *tc-bpf* (8)

Both classic and extended BPF are explained in the kernel source file *Documentation/networking/filter.txt*.

NAME

brk, sbrk − change data segment size

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int brk(void *addr);
void *sbrk(intptr_t increment);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

brk(), **sbrk**():
- Since glibc 2.19:
  - _DEFAULT_SOURCE
    - || ((_XOPEN_SOURCE >= 500) &&
      - ! (_POSIX_C_SOURCE >= 200112L))
- From glibc 2.12 to glibc 2.19:
  - _BSD_SOURCE || _SVID_SOURCE
    - || ((_XOPEN_SOURCE >= 500) &&
      - ! (_POSIX_C_SOURCE >= 200112L))
- Before glibc 2.12:
  - _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

brk() and **sbrk**() change the location of the *program break*, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by *addr*, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see *setrlimit*(2)).

sbrk() increments the program's data space by *increment* bytes. Calling **sbrk**() with an *increment* of 0 can be used to find the current location of the program break.

RETURN VALUE

On success, **brk**() returns zero. On error, −1 is returned, and *errno* is set to **ENOMEM**.

On success, **sbrk**() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, *(void *) −1* is returned, and *errno* is set to **ENOMEM**.

STANDARDS

None.

HISTORY

4.3BSD; SUSv1, marked LEGACY in SUSv2, removed in POSIX.1-2001.

NOTES

Avoid using **brk**() and **sbrk**(): the *malloc*(3) memory allocation package is the portable and comfortable way of allocating memory.

Various systems use various types for the argument of **sbrk**(). Common are *int*, *ssize_t*, *ptrdiff_t*, *intptr_t*.

C library/kernel differences

The return value described above for **brk**() is the behavior provided by the glibc wrapper function for the Linux **brk**() system call. (On most other implementations, the return value from **brk**() is the same; this return value was also specified in SUSv2.) However, the actual Linux system call returns the new program break on success. On failure, the system call returns the current break. The glibc wrapper function does some work (i.e., checks whether the new break is less than *addr*) to provide the 0 and −1 return values described above.

On Linux, **sbrk**() is implemented as a library function that uses the **brk**() system call, and does some internal bookkeeping so that it can return the old break value.

SEE ALSO

execve(2), *getrlimit*(2), *end*(3), *malloc*(3)

NAME

cacheflush − flush contents of instruction and/or data cache

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/cachectl.h>

int cacheflush(void *addr***[.***nbytes***], int** *nbytes***, int** *cache***);**

Note: On some architectures, there is no glibc wrapper for this system call; see NOTES.

DESCRIPTION

cacheflush() flushes the contents of the indicated cache(s) for the user addresses in the range *addr* to *(addr+nbytes−1)*. *cache* may be one of:

ICACHE
: Flush the instruction cache.

DCACHE
: Write back to memory and invalidate the affected valid cache lines.

BCACHE
: Same as **(ICACHE|DCACHE)**.

RETURN VALUE

cacheflush() returns 0 on success. On error, it returns −1 and sets *errno* to indicate the error.

ERRORS

EFAULT
: Some or all of the address range *addr* to *(addr+nbytes−1)* is not accessible.

EINVAL
: *cache* is not one of **ICACHE**, **DCACHE**, or **BCACHE** (but see BUGS).

VERSIONS

cacheflush() should not be used in programs intended to be portable. On Linux, this call first appeared on the MIPS architecture, but nowadays, Linux provides a **cacheflush**() system call on some other architectures, but with different arguments.

Architecture-specific variants

glibc provides a wrapper for this system call, with the prototype shown in SYNOPSIS, for the following architectures: ARC, CSKY, MIPS, and NIOS2.

On some other architectures, Linux provides this system call, with different arguments:

M68K:

int cacheflush(unsigned long *addr***, int** *scope***, int** *cache***,**
unsigned long *len***);**

SH:

int cacheflush(unsigned long *addr***, unsigned long** *len***, int** *op***);**

NDS32:

int cacheflush(unsigned int *start***, unsigned int** *end***, int** *cache***);**

On the above architectures, glibc does not provide a wrapper for this system call; call it using *syscall*(2).

GCC alternative

Unless you need the finer grained control that this system call provides, you probably want to use the GCC built-in function **__builtin___clear_cache**(), which provides a portable interface across platforms supported by GCC and compatible compilers:

```
void __builtin___clear_cache(void *begin, void *end);
```

On platforms that don't require instruction cache flushes, **__builtin___clear_cache**() has no effect.

Note: On some GCC-compatible compilers, the prototype for this built-in function uses *char* * instead of *void* * for the parameters.

STANDARDS

Historically, this system call was available on all MIPS UNIX variants including RISC/os, IRIX, Ultrix, NetBSD, OpenBSD, and FreeBSD (and also on some non-UNIX MIPS operating systems), so that the existence of this call in MIPS operating systems is a de-facto standard.

BUGS

Linux kernels older than Linux 2.6.11 ignore the *addr* and *nbytes* arguments, making this function fairly expensive. Therefore, the whole cache is always flushed.

This function always behaves as if **BCACHE** has been passed for the *cache* argument and does not do any error checking on the *cache* argument.

NAME

capget, capset − set/get capabilities of thread(s)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/capability.h> /* Definition of **CAP_*** and
_LINUX_CAPABILITY_* constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_capget, cap_user_header_t *hdrp***,**
cap_user_data_t *datap***);**
int syscall(SYS_capset, cap_user_header_t *hdrp***,**
const cap_user_data_t *datap***);**

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

These two system calls are the raw kernel interface for getting and setting thread capabilities. Not only are these system calls specific to Linux, but the kernel API is likely to change and use of these system calls (in particular the format of the *cap_user_*_t* types) is subject to extension with each kernel revision, but old programs will keep working.

The portable interfaces are *cap_set_proc*(3) and *cap_get_proc*(3); if possible, you should use those interfaces in applications; see NOTES.

Current details

Now that you have been warned, some current kernel details. The structures are defined as follows.

```
#define _LINUX_CAPABILITY_VERSION_1  0x19980330
#define _LINUX_CAPABILITY_U32S_1     1

        /* V2 added in Linux 2.6.25; deprecated */
#define _LINUX_CAPABILITY_VERSION_2  0x20071026
#define _LINUX_CAPABILITY_U32S_2     2

        /* V3 added in Linux 2.6.26 */
#define _LINUX_CAPABILITY_VERSION_3  0x20080522
#define _LINUX_CAPABILITY_U32S_3     2

typedef struct __user_cap_header_struct {
   __u32 version;
   int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
   __u32 effective;
   __u32 permitted;
   __u32 inheritable;
} *cap_user_data_t;
```

The *effective*, *permitted*, and *inheritable* fields are bit masks of the capabilities defined in *capabilities*(7). Note that the **CAP_*** values are bit indexes and need to be bit-shifted before ORing into the bit fields. To define the structures for passing to the system call, you have to use the *struct __user_cap_header_struct* and *struct __user_cap_data_struct* names because the typedefs are only pointers.

Kernels prior to Linux 2.6.25 prefer 32-bit capabilities with version **_LINUX_CAPABILITY_VERSION_1**. Linux 2.6.25 added 64-bit capability sets, with version **_LINUX_CAPABILITY_VERSION_2**. There was, however, an API glitch, and Linux 2.6.26 added **_LINUX_CAPABILITY_VERSION_3** to fix the problem.

Note that 64-bit capabilities use *datap[0]* and *datap[1]*, whereas 32-bit capabilities use only *datap[0]*.

On kernels that support file capabilities (VFS capabilities support), these system calls behave slightly differently. This support was added as an option in Linux 2.6.24, and became fixed (nonoptional) in Linux 2.6.33.

For **capget**() calls, one can probe the capabilities of any process by specifying its process ID with the *hdrp->pid* field value.

For details on the data, see *capabilities*(7).

With VFS capabilities support

VFS capabilities employ a file extended attribute (see *xattr*(7)) to allow capabilities to be attached to executables. This privilege model obsoletes kernel support for one process asynchronously setting the capabilities of another. That is, on kernels that have VFS capabilities support, when calling **capset**(), the only permitted values for *hdrp->pid* are 0 or, equivalently, the value returned by *gettid*(2).

Without VFS capabilities support

On older kernels that do not provide VFS capabilities support **capset**() can, if the caller has the **CAP_SETPCAP** capability, be used to change not only the caller's own capabilities, but also the capabilities of other threads. The call operates on the capabilities of the thread specified by the *pid* field of *hdrp* when that is nonzero, or on the capabilities of the calling thread if *pid* is 0. If *pid* refers to a single-threaded process, then *pid* can be specified as a traditional process ID; operating on a thread of a multithreaded process requires a thread ID of the type returned by *gettid*(2). For **capset**(), *pid* can also be: −1, meaning perform the change on all threads except the caller and *init*(1); or a value less than −1, in which case the change is applied to all members of the process group whose ID is −*pid*.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

The calls fail with the error **EINVAL**, and set the *version* field of *hdrp* to the kernel preferred value of **_LINUX_CAPABILITY_VERSION_?** when an unsupported *version* value is specified. In this way, one can probe what the current preferred capability revision is.

ERRORS

EFAULT

Bad memory address. *hdrp* must not be NULL. *datap* may be NULL only when the user is trying to determine the preferred capability version format supported by the kernel.

EINVAL

One of the arguments was invalid.

EPERM

An attempt was made to add a capability to the permitted set, or to set a capability in the effective set that is not in the permitted set.

EPERM

An attempt was made to add a capability to the inheritable set, and either:

- that capability was not in the caller's bounding set; or
- the capability was not in the caller's permitted set and the caller lacked the **CAP_SETPCAP** capability in its effective set.

EPERM

The caller attempted to use **capset**() to modify the capabilities of a thread other than itself, but lacked sufficient privilege. For kernels supporting VFS capabilities, this is never permitted. For kernels lacking VFS support, the **CAP_SETPCAP** capability is required. (A bug in kernels before Linux 2.6.11 meant that this error could also occur if a thread without this capability tried to change its own capabilities by specifying the *pid* field as a nonzero value (i.e., the value returned by *getpid*(2)) instead of 0.)

ESRCH

No such thread.

STANDARDS

Linux.

NOTES

The portable interface to the capability querying and setting functions is provided by the *libcap* library and is available here:

SEE ALSO

clone(2), *gettid*(2), *capabilities*(7)

NAME

chdir, fchdir − change working directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fchdir():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc up to and including 2.19: */ _BSD_SOURCE

DESCRIPTION

chdir() changes the current working directory of the calling process to the directory specified in *path*.

fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Depending on the filesystem, other errors can be returned. The more general errors for **chdir**() are listed below:

EACCES
Search permission is denied for one of the components of *path*. (See also *path_resolution*(7).)

EFAULT
path points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
path is too long.

ENOENT
The directory specified in *path* does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of *path* is not a directory.

The general errors for **fchdir**() are listed below:

EACCES
Search permission was denied on the directory open on *fd*.

EBADF
fd is not a valid file descriptor.

ENOTDIR
fd does not refer to a directory.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD.

NOTES

The current working directory is the starting point for interpreting relative pathnames (those not starting with '/').

A child process created via *fork*(2) inherits its parent's current working directory. The current working directory is left unchanged by *execve*(2).

SEE ALSO

chroot(2), *getcwd*(3), *path_resolution*(7)

NAME

chmod, fchmod, fchmodat − change permissions of a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fchmodat(int dirfd, const char *pathname, mode_t mode, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fchmod():
- Since glibc 2.24:
  - _POSIX_C_SOURCE >= 199309L
- glibc 2.19 to glibc 2.23
  - _POSIX_C_SOURCE
- glibc 2.16 to glibc 2.19:
  - _BSD_SOURCE || _POSIX_C_SOURCE
- glibc 2.12 to glibc 2.16:
  - _BSD_SOURCE || _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
- glibc 2.11 and earlier:
  - _BSD_SOURCE || _XOPEN_SOURCE >= 500

fchmodat():
- Since glibc 2.10:
  - _POSIX_C_SOURCE >= 200809L
- Before glibc 2.10:
  - _ATFILE_SOURCE

DESCRIPTION

The **chmod**() and **fchmod**() system calls change a file's mode bits. (The file mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits.) These system calls differ only in how the file is specified:

- **chmod**() changes the mode of the file specified whose pathname is given in *pathname*, which is dereferenced if it is a symbolic link.
- **fchmod**() changes the mode of the file referred to by the open file descriptor *fd*.

The new file mode is specified in *mode*, which is a bit mask created by ORing together zero or more of the following:

S_ISUID	(04000)	set-user-ID (set process effective user ID on *execve*(2))
S_ISGID	(02000)	set-group-ID (set process effective group ID on *execve*(2); mandatory locking, as described in *fcntl*(2); take a new file's group from parent directory, as described in *chown*(2) and *mkdir*(2))
S_ISVTX	(01000)	sticky bit (restricted deletion flag, as described in *unlink*(2))
S_IRUSR	(00400)	read by owner
S_IWUSR	(00200)	write by owner
S_IXUSR	(00100)	execute/search by owner ("search" applies for directories, and means that entries within the directory can be accessed)
S_IRGRP	(00040)	read by group
S_IWGRP	(00020)	write by group

S_IXGRP (00010) execute/search by group

S_IROTH (00004) read by others

S_IWOTH (00002) write by others

S_IXOTH (00001) execute/search by others

The effective UID of the calling process must match the owner of the file, or the process must be privileged (Linux: it must have the **CAP_FOWNER** capability).

If the calling process is not privileged (Linux: does not have the **CAP_FSETID** capability), and the group of the file does not match the effective group ID of the process or one of its supplementary group IDs, the **S_ISGID** bit will be turned off, but this will not cause an error to be returned.

As a security measure, depending on the filesystem, the set-user-ID and set-group-ID execution bits may be turned off if a file is written. (On Linux, this occurs if the writing process does not have the **CAP_FSETID** capability.) On some filesystems, only the superuser can set the sticky bit, which may have a special meaning. For the sticky bit, and for set-user-ID and set-group-ID bits on directories, see *inode*(7).

On NFS filesystems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client. Widening the permissions may be delayed for other clients if attribute caching is enabled on them.

fchmodat()

The **fchmodat**() system call operates in exactly the same way as **chmod**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **chmod**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *chmod*())

If *pathname* is absolute, then *dirfd* is ignored.

flags can either be 0, or include the following flag:

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, do not dereference it: instead operate on the link itself. This flag is not currently implemented.

See *openat*(2) for an explanation of the need for **fchmodat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Depending on the filesystem, errors other than those listed below can be returned.

The more general errors for **chmod**() are listed below:

EACCES

Search permission is denied on a component of the path prefix. (See also *path_resolution*(7).)

EBADF

(**fchmod**()) The file descriptor *fd* is not valid.

EBADF

(**fchmodat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT

pathname points outside your accessible address space.

EINVAL

(**fchmodat**()) Invalid flag specified in *flags*.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

ENOTDIR
(**fchmodat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

ENOTSUP
(**fchmodat**()) *flags* specified **AT_SYMLINK_NOFOLLOW**, which is not supported.

EPERM
The effective UID does not match the owner of the file, and the process is not privileged (Linux: it does not have the **CAP_FOWNER** capability).

EPERM
The file is marked immutable or append-only. (See *ioctl_iflags*(2).)

EROFS
The named file resides on a read-only filesystem.

VERSIONS

C library/kernel differences

The GNU C library **fchmodat**() wrapper function implements the POSIX-specified interface described in this page. This interface differs from the underlying Linux system call, which does *not* have a *flags* argument.

glibc notes

On older kernels where **fchmodat**() is unavailable, the glibc wrapper function falls back to the use of **chmod**(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

STANDARDS

POSIX.1-2008.

HISTORY

chmod()
fchmod()
4.4BSD, SVr4, POSIX.1-2001.

fchmodat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

SEE ALSO

chmod(1), *chown*(2), *execve*(2), *open*(2), *stat*(2), *inode*(7), *path_resolution*(7), *symlink*(7)

NAME

chown, fchown, lchown, fchownat − change ownership of a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int fchownat(int dirfd, const char *pathname,
             uid_t owner, gid_t group, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fchown(), **lchown**():

```
/* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
    || _XOPEN_SOURCE >= 500
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

fchownat():

```
Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
    _ATFILE_SOURCE
```

DESCRIPTION

These system calls change the owner and group of a file. The **chown**(), **fchown**(), and **lchown**() system calls differ only in how the file is specified:

- **chown**() changes the ownership of the file specified by *pathname*, which is dereferenced if it is a symbolic link.
- **fchown**() changes the ownership of the file referred to by the open file descriptor *fd*.
- **lchown**() is like **chown**(), but does not dereference symbolic links.

Only a privileged process (Linux: one with the **CAP_CHOWN** capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. A privileged process (Linux: with **CAP_CHOWN**) may change the group arbitrarily.

If the *owner* or *group* is specified as −1, then that ID is not changed.

When the owner or group of an executable file is changed by an unprivileged user, the **S_ISUID** and **S_ISGID** mode bits are cleared. POSIX does not specify whether this also should happen when root does the **chown**(); the Linux behavior depends on the kernel version, and since Linux 2.2.13, root is treated like other users. In case of a non-group-executable file (i.e., one for which the **S_IXGRP** bit is not set) the **S_ISGID** bit indicates mandatory locking, and is not cleared by a **chown**().

When the owner or group of an executable file is changed (by any user), all capability sets for the file are cleared.

fchownat()

The **fchownat**() system call operates in exactly the same way as **chown**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **chown**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *chown*())

If *pathname* is absolute, then *dirfd* is ignored.

The *flags* argument is a bit mask created by ORing together 0 or more of the following values;

AT_EMPTY_PATH (since Linux 2.6.39)
If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory. If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_NOFOLLOW
If *pathname* is a symbolic link, do not dereference it: instead operate on the link itself, like **lchown**(). (By default, **fchownat**() dereferences symbolic links, like *chown*().)

See *openat*(2) for an explanation of the need for **fchownat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Depending on the filesystem, errors other than those listed below can be returned.

The more general errors for **chown**() are listed below.

EACCES
Search permission is denied on a component of the path prefix. (See also *path_resolution*(7).)

EBADF
(**fchown**()) *fd* is not a valid open file descriptor.

EBADF
(**fchownat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
pathname points outside your accessible address space.

EINVAL
(**fchownat**()) Invalid flag specified in *flags*.

EIO (**fchown**()) A low-level I/O error occurred while modifying the inode.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

ENOTDIR
(**fchownat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM
The calling process did not have the required permissions (see above) to change owner and/or group.

EPERM
The file is marked immutable or append-only. (See *ioctl_iflags*(2).)

EROFS
The named file resides on a read-only filesystem.

VERSIONS

The 4.4BSD version can be used only by the superuser (that is, ordinary users cannot give away files).

STANDARDS

POSIX.1-2008.

HISTORY

chown()
fchown()
lchown()
4.4BSD, SVr4, POSIX.1-2001.

fchownat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES

Ownership of new files

When a new file is created (by, for example, *open*(2) or *mkdir*(2)), its owner is made the same as the filesystem user ID of the creating process. The group of the file depends on a range of factors, including the type of filesystem, the options used to mount the filesystem, and whether or not the set-group-ID mode bit is enabled on the parent directory. If the filesystem supports the **−o grpid** (or, synonymously **-o bsdgroups**) and **−o nogrpid** (or, synonymously **-o sysvgroups**) *mount*(8) options, then the rules are as follows:

- If the filesystem is mounted with **-o grpid**, then the group of a new file is made the same as that of the parent directory.
- If the filesystem is mounted with **−o nogrpid** and the set-group-ID bit is disabled on the parent directory, then the group of a new file is made the same as the process's filesystem GID.
- If the filesystem is mounted with **−o nogrpid** and the set-group-ID bit is enabled on the parent directory, then the group of a new file is made the same as that of the parent directory.

As at Linux 4.12, the **−o grpid** and **−o nogrpid** mount options are supported by ext2, ext3, ext4, and XFS. Filesystems that don't support these mount options follow the **−o nogrpid** rules.

glibc notes

On older kernels where **fchownat**() is unavailable, the glibc wrapper function falls back to the use of **chown**() and **lchown**(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

NFS

The **chown**() semantics are deliberately violated on NFS filesystems which have UID mapping enabled. Additionally, the semantics of all system calls which access the file contents are violated, because **chown**() may cause immediate access revocation on already open files. Client side caching may lead to a delay between the time where ownership have been changed to allow access for a user and the time where the file can actually be accessed by the user on other clients.

Historical details

The original Linux **chown**(), **fchown**(), and **lchown**() system calls supported only 16-bit user and group IDs. Subsequently, Linux 2.4 added **chown32**(), **fchown32**(), and **lchown32**(), supporting 32-bit IDs. The glibc **chown**(), **fchown**(), and **lchown**() wrapper functions transparently deal with the variations across kernel versions.

Before Linux 2.1.81 (except 2.1.46), **chown**() did not follow symbolic links. Since Linux 2.1.81, **chown**() does follow symbolic links, and there is a new system call **lchown**() that does not follow symbolic links. Since Linux 2.1.86, this new call (that has the same semantics as the old *chown*()) has got the same syscall number, and **chown**() got the newly introduced number.

EXAMPLES

The following program changes the ownership of the file named in its second command-line argument to the value specified in its first command-line argument. The new owner can be specified either as a numeric user ID, or as a username (which is converted to a user ID by using *getpwnam*(3) to perform a lookup in the system password file).

Program source

```
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>

int
main(int argc, char *argv[])
{
    char           *endptr;
    uid_t          uid;
    struct passwd  *pwd;

    if (argc != 3 || argv[1][0] == '\0') {
        fprintf(stderr, "%s <owner> <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    uid = strtol(argv[1], &endptr, 10);  /* Allow a numeric string */

    if (*endptr != '\0') {         /* Was not pure numeric string */
        pwd = getpwnam(argv[1]);   /* Try getting UID for username */
        if (pwd == NULL) {
            perror("getpwnam");
            exit(EXIT_FAILURE);
        }

        uid = pwd->pw_uid;
    }

    if (chown(argv[2], uid, -1) == -1) {
        perror("chown");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

chgrp(1), *chown*(1), *chmod*(2), *flock*(2), *path_resolution*(7), *symlink*(7)

NAME

chroot − change root directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int chroot(const char ****path*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

chroot():
Since glibc 2.2.2:
_XOPEN_SOURCE && ! (_POSIX_C_SOURCE >= 200112L)
|| /* Since glibc 2.20: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE
Before glibc 2.2.2:
none

DESCRIPTION

chroot() changes the root directory of the calling process to that specified in *path*. This directory will be used for pathnames beginning with /. The root directory is inherited by all children of the calling process.

Only a privileged process (Linux: one with the **CAP_SYS_CHROOT** capability in its user namespace) may call **chroot**().

This call changes an ingredient in the pathname resolution process and does nothing else. In particular, it is not intended to be used for any kind of security purpose, neither to fully sandbox a process nor to restrict filesystem system calls. In the past, **chroot**() has been used by daemons to restrict themselves prior to passing paths supplied by untrusted users to system calls such as *open*(2). However, if a folder is moved out of the chroot directory, an attacker can exploit that to get out of the chroot directory as well. The easiest way to do that is to *chdir*(2) to the to-be-moved directory, wait for it to be moved out, then open a path like ../../../etc/passwd.

A slightly trickier variation also works under some circumstances if *chdir*(2) is not permitted. If a daemon allows a "chroot directory" to be specified, that usually means that if you want to prevent remote users from accessing files outside the chroot directory, you must ensure that folders are never moved out of it.

This call does not change the current working directory, so that after the call '.' can be outside the tree rooted at '/'. In particular, the superuser can escape from a "chroot jail" by doing:

```
mkdir foo; chroot foo; cd ..
```

This call does not close open file descriptors, and such file descriptors may allow access to files outside the chroot tree.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Depending on the filesystem, other errors can be returned. The more general errors are listed below:

EACCES
Search permission is denied on a component of the path prefix. (See also *path_resolution*(7).)

EFAULT
path points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
path is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of *path* is not a directory.

EPERM
The caller has insufficient privilege.

STANDARDS

None.

HISTORY

SVr4, 4.4BSD, SUSv2 (marked LEGACY). This function is not part of POSIX.1-2001.

NOTES

A child process created via *fork*(2) inherits its parent's root directory. The root directory is left unchanged by *execve*(2).

The magic symbolic link, */proc/*pid*/root*, can be used to discover a process's root directory; see *proc*(5) for details.

FreeBSD has a stronger **jail**() system call.

SEE ALSO

chroot(1), *chdir*(2), *pivot_root*(2), *path_resolution*(7), *switch_root*(8)

NAME

clock_getres, clock_gettime, clock_settime − clock and time functions

LIBRARY

Standard C library (*libc*, *−lc*), since glibc 2.17

Before glibc 2.17, Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <time.h>

int clock_getres(clockid_t clockid, struct timespec *_Nullable res);

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_settime(clockid_t clockid, const struct timespec *tp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

clock_getres(), **clock_gettime**(), **clock_settime**():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

The function **clock_getres**() finds the resolution (precision) of the specified clock *clockid*, and, if *res* is non-NULL, stores it in the *struct timespec* pointed to by *res*. The resolution of clocks depends on the implementation and cannot be configured by a particular process. If the time value pointed to by the argument *tp* of **clock_settime**() is not a multiple of *res*, then it is truncated to a multiple of *res*.

The functions **clock_gettime**() and **clock_settime**() retrieve and set the time of the specified clock *clockid*.

The *res* and *tp* arguments are *timespec*(3) structures.

The *clockid* argument is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process.

All implementations support the system-wide real-time clock, which is identified by **CLOCK_REALTIME**. Its time represents seconds and nanoseconds since the Epoch. When its time is changed, timers for a relative interval are unaffected, but timers for an absolute point in time are affected.

More clocks may be implemented. The interpretation of the corresponding time values and the effect on timers is unspecified.

Sufficiently recent versions of glibc and the Linux kernel support the following clocks:

CLOCK_REALTIME
A settable system-wide clock that measures real (i.e., wall-clock) time. Setting this clock requires appropriate privileges. This clock is affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), and by frequency adjustments performed by NTP and similar applications via *adjtime*(3), *adjtimex*(2), *clock_adjtime*(2), and *ntp_adjtime*(3). This clock normally counts the number of seconds since 1970-01-01 00:00:00 Coordinated Universal Time (UTC) except that it ignores leap seconds; near a leap second it is typically adjusted by NTP to stay roughly in sync with UTC.

CLOCK_REALTIME_ALARM (since Linux 3.0; Linux-specific)
Like **CLOCK_REALTIME**, but not settable. See *timer_create*(2) for further details.

CLOCK_REALTIME_COARSE (since Linux 2.6.32; Linux-specific)
A faster but less precise version of **CLOCK_REALTIME**. This clock is not settable. Use when you need very fast, but not fine-grained timestamps. Requires per-architecture support, and probably also architecture support for this flag in the *vdso*(7).

CLOCK_TAI (since Linux 3.10; Linux-specific)
A nonsettable system-wide clock derived from wall-clock time but counting leap seconds. This clock does not experience discontinuities or frequency adjustments caused by inserting leap seconds as **CLOCK_REALTIME** does.

The acronym TAI refers to International Atomic Time.

CLOCK_MONOTONIC
A nonsettable system-wide clock that represents monotonic time since—as described by POSIX—"some unspecified point in the past". On Linux, that point corresponds to the

number of seconds that the system has been running since it was booted.

The **CLOCK_MONOTONIC** clock is not affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), but is affected by frequency adjustments. This clock does not count time that the system is suspended. All **CLOCK_MONOTONIC** variants guarantee that the time returned by consecutive calls will not go backwards, but successive calls may—depending on the architecture—return identical (not-increased) time values.

CLOCK_MONOTONIC_COARSE (since Linux 2.6.32; Linux-specific)
A faster but less precise version of **CLOCK_MONOTONIC**. Use when you need very fast, but not fine-grained timestamps. Requires per-architecture support, and probably also architecture support for this flag in the *vdso*(7).

CLOCK_MONOTONIC_RAW (since Linux 2.6.28; Linux-specific)
Similar to **CLOCK_MONOTONIC**, but provides access to a raw hardware-based time that is not subject to frequency adjustments. This clock does not count time that the system is suspended.

CLOCK_BOOTTIME (since Linux 2.6.39; Linux-specific)
A nonsettable system-wide clock that is identical to **CLOCK_MONOTONIC**, except that it also includes any time that the system is suspended. This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of **CLOCK_REALTIME**, which may have discontinuities if the time is changed using *settimeofday*(2) or similar.

CLOCK_BOOTTIME_ALARM (since Linux 3.0; Linux-specific)
Like **CLOCK_BOOTTIME**. See *timer_create*(2) for further details.

CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
This is a clock that measures CPU time consumed by this process (i.e., CPU time consumed by all threads in the process). On Linux, this clock is not settable.

CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
This is a clock that measures CPU time consumed by this thread. On Linux, this clock is not settable.

Linux also implements dynamic clock instances as described below.

Dynamic clocks

In addition to the hard-coded System-V style clock IDs described above, Linux also supports POSIX clock operations on certain character devices. Such devices are called "dynamic" clocks, and are supported since Linux 2.6.39.

Using the appropriate macros, open file descriptors may be converted into clock IDs and passed to **clock_gettime**(), **clock_settime**(), and *clock_adjtime*(2). The following example shows how to convert a file descriptor into a dynamic clock ID.

```
#define CLOCKFD 3
#define FD_TO_CLOCKID(fd)   ((~(clockid_t) (fd) << 3) | CLOCKFD)
#define CLOCKID_TO_FD(clk)  ((unsigned int) ~((clk) >> 3))

struct timespec ts;
clockid_t clkid;
int fd;

fd = open("/dev/ptp0", O_RDWR);
clkid = FD_TO_CLOCKID(fd);
clock_gettime(clkid, &ts);
```

RETURN VALUE

clock_gettime(), **clock_settime**(), and **clock_getres**() return 0 for success. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
: **clock_settime**() does not have write permission for the dynamic POSIX clock device indicated.

EFAULT
: *tp* points outside the accessible address space.

EINVAL
: The *clockid* specified is invalid for one of two reasons. Either the System-V style hard coded positive value is out of range, or the dynamic clock ID does not refer to a valid instance of a clock object.

EINVAL
: (**clock_settime**()): *tp.tv_sec* is negative or *tp.tv_nsec* is outside the range [0, 999,999,999].

EINVAL
: The *clockid* specified in a call to **clock_settime**() is not a settable clock.

EINVAL (since Linux 4.3)
: A call to **clock_settime**() with a *clockid* of **CLOCK_REALTIME** attempted to set the time to a value less than the current value of the **CLOCK_MONOTONIC** clock.

ENODEV
: The hot-pluggable device (like USB for example) represented by a dynamic *clk_id* has disappeared after its character device was opened.

ENOTSUP
: The operation is not supported by the dynamic POSIX clock device specified.

EOVERFLOW
: The timestamp would not fit in *time_t* range. This can happen if an executable with 32-bit *time_t* is run on a 64-bit kernel when the time is 2038-01-19 03:14:08 UTC or later. However, when the system time is out of *time_t* range in other situations, the behavior is undefined.

EPERM
: **clock_settime**() does not have permission to set the clock indicated.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clock_getres(), **clock_gettime**(), **clock_settime**()	Thread safety	MT-Safe

VERSIONS

POSIX.1 specifies the following:

> Setting the value of the **CLOCK_REALTIME** clock via **clock_settime**() shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the **nanosleep**() function; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

According to POSIX.1-2001, a process with "appropriate privileges" may set the **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID** clocks using **clock_settime**(). On Linux, these clocks are not settable (i.e., no process has "appropriate privileges").

C library/kernel differences

On some architectures, an implementation of **clock_gettime**() is provided in the *vdso*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SUSv2. Linux 2.6.

On POSIX systems on which these functions are available, the symbol **_POSIX_TIMERS** is defined in <*unistd.h*> to a value greater than 0. POSIX.1-2008 makes these functions mandatory.

The symbols **_POSIX_MONOTONIC_CLOCK**, **_POSIX_CPUTIME**,

_POSIX_THREAD_CPUTIME indicate that **CLOCK_MONOTONIC**, **CLOCK_PROCESS_CPUTIME_ID**, **CLOCK_THREAD_CPUTIME_ID** are available. (See also *sysconf*(3).)

Historical note for SMP systems

Before Linux added kernel support for **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID**, glibc implemented these clocks on many platforms using timer registers from the CPUs (TSC on i386, AR.ITC on Itanium). These registers may differ between CPUs and as a consequence these clocks may return **bogus results** if a process is migrated to another CPU.

If the CPUs in an SMP system have different clock sources, then there is no way to maintain a correlation between the timer registers since each CPU will run at a slightly different frequency. If that is the case, then *clock_getcpuclockid(0)* will return **ENOENT** to signify this condition. The two clocks will then be useful only if it can be ensured that a process stays on a certain CPU.

The processors in an SMP system do not start all at exactly the same time and therefore the timer registers are typically running at an offset. Some architectures include code that attempts to limit these offsets on bootup. However, the code cannot guarantee to accurately tune the offsets. glibc contains no provisions to deal with these offsets (unlike the Linux Kernel). Typically these offsets are small and therefore the effects may be negligible in most cases.

Since glibc 2.4, the wrapper functions for the system calls described in this page avoid the abovementioned problems by employing the kernel implementation of **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID**, on systems that provide such an implementation (i.e., Linux 2.6.12 and later).

EXAMPLES

The program below demonstrates the use of **clock_gettime**() and **clock_getres**() with various clocks. This is an example of what we might see when running the program:

```
$ ./clock_times x
CLOCK_REALTIME : 1585985459.446 (18356 days +  7h 30m 59s)
     resolution:          0.000000001
CLOCK_TAI      : 1585985496.447 (18356 days +  7h 31m 36s)
     resolution:          0.000000001
CLOCK_MONOTONIC:      52395.722 (14h 33m 15s)
     resolution:          0.000000001
CLOCK_BOOTTIME :      72691.019 (20h 11m 31s)
     resolution:          0.000000001
```

Program source

```
/* clock_times.c

   Licensed under GNU General Public License v2 or later.
*/
#define _XOPEN_SOURCE 600
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SECS_IN_DAY (24 * 60 * 60)

static void
displayClock(clockid_t clock, const char *name, bool showRes)
{
    long             days;
    struct timespec  ts;

    if (clock_gettime(clock, &ts) == -1) {
        perror("clock_gettime");
```

```
            exit(EXIT_FAILURE);
        }

        printf("%-15s: %10jd.%03ld (", name,
               (intmax_t) ts.tv_sec, ts.tv_nsec / 1000000);

        days = ts.tv_sec / SECS_IN_DAY;
        if (days > 0)
            printf("%ld days + ", days);

        printf("%2dh %2dm %2ds",
               (int) (ts.tv_sec % SECS_IN_DAY) / 3600,
               (int) (ts.tv_sec % 3600) / 60,
               (int) ts.tv_sec % 60);
        printf(")\n");

        if (clock_getres(clock, &ts) == -1) {
            perror("clock_getres");
            exit(EXIT_FAILURE);
        }

        if (showRes)
            printf("     resolution: %10jd.%09ld\n",
                   (intmax_t) ts.tv_sec, ts.tv_nsec);
    }

    int
    main(int argc, char *argv[])
    {
        bool showRes = argc > 1;

        displayClock(CLOCK_REALTIME, "CLOCK_REALTIME", showRes);
    #ifdef CLOCK_TAI
        displayClock(CLOCK_TAI, "CLOCK_TAI", showRes);
    #endif
        displayClock(CLOCK_MONOTONIC, "CLOCK_MONOTONIC", showRes);
    #ifdef CLOCK_BOOTTIME
        displayClock(CLOCK_BOOTTIME, "CLOCK_BOOTTIME", showRes);
    #endif
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

date(1), *gettimeofday*(2), *settimeofday*(2), *time*(2), *adjtime*(3), *clock_getcpuclockid*(3), *ctime*(3), *ftime*(3), *pthread_getcpuclockid*(3), *sysconf*(3), *timespec*(3), *time*(7), *time_namespaces*(7), *vdso*(7), *hwclock*(8)

NAME

clock_nanosleep − high-resolution sleep with specifiable clock

LIBRARY

Standard C library (*libc*, *−lc*), since glibc 2.17

Before glibc 2.17, Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <time.h>

int clock_nanosleep(clockid_t clockid, int flags,
                    const struct timespec *t,
                    struct timespec *_Nullable remain);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

clock_nanosleep():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

Like *nanosleep*(2), **clock_nanosleep**() allows the calling thread to sleep for an interval specified with nanosecond precision. It differs in allowing the caller to select the clock against which the sleep interval is to be measured, and in allowing the sleep interval to be specified as either an absolute or a relative value.

The time values passed to and returned by this call are specified using *timespec*(3) structures.

The *clockid* argument specifies the clock against which the sleep interval is to be measured. This argument can have one of the following values:

CLOCK_REALTIME
A settable system-wide real-time clock.

CLOCK_TAI (since Linux 3.10)
A system-wide clock derived from wall-clock time but counting leap seconds.

CLOCK_MONOTONIC
A nonsettable, monotonically increasing clock that measures time since some unspecified point in the past that does not change after system startup.

CLOCK_BOOTTIME (since Linux 2.6.39)
Identical to **CLOCK_MONOTONIC**, except that it also includes any time that the system is suspended.

CLOCK_PROCESS_CPUTIME_ID
A settable per-process clock that measures CPU time consumed by all threads in the process.

See *clock_getres*(2) for further details on these clocks. In addition, the CPU clock IDs returned by *clock_getcpuclockid*(3) and *pthread_getcpuclockid*(3) can also be passed in *clockid*.

If *flags* is 0, then the value specified in *t* is interpreted as an interval relative to the current value of the clock specified by *clockid*.

If *flags* is **TIMER_ABSTIME**, then *t* is interpreted as an absolute time as measured by the clock, *clockid*. If *t* is less than or equal to the current value of the clock, then **clock_nanosleep**() returns immediately without suspending the calling thread.

clock_nanosleep() suspends the execution of the calling thread until either at least the time specified by *t* has elapsed, or a signal is delivered that causes a signal handler to be called or that terminates the process.

If the call is interrupted by a signal handler, **clock_nanosleep**() fails with the error **EINTR**. In addition, if *remain* is not NULL, and *flags* was not **TIMER_ABSTIME**, it returns the remaining unslept time in *remain*. This value can then be used to call **clock_nanosleep**() again and complete a (relative) sleep.

RETURN VALUE

On successfully sleeping for the requested interval, **clock_nanosleep**() returns 0. If the call is interrupted by a signal handler or encounters an error, then it returns one of the positive error number listed in ERRORS.

ERRORS

EFAULT
: *t* or *remain* specified an invalid address.

EINTR
: The sleep was interrupted by a signal handler; see *signal*(7).

EINVAL
: The value in the *tv_nsec* field was not in the range [0, 999999999] or *tv_sec* was negative.

EINVAL
: *clockid* was invalid. (**CLOCK_THREAD_CPUTIME_ID** is not a permitted value for *clockid*.)

ENOTSUP
: The kernel does not support sleeping against this *clockid*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. Linux 2.6, glibc 2.1.

NOTES

If the interval specified in *t* is not an exact multiple of the granularity underlying clock (see *time*(7)), then the interval will be rounded up to the next multiple. Furthermore, after the sleep completes, there may still be a delay before the CPU becomes free to once again execute the calling thread.

Using an absolute timer is useful for preventing timer drift problems of the type described in *nanosleep*(2). (Such problems are exacerbated in programs that try to restart a relative sleep that is repeatedly interrupted by signals.) To perform a relative sleep that avoids these problems, call *clock_gettime*(2) for the desired clock, add the desired interval to the returned time value, and then call **clock_nanosleep**() with the **TIMER_ABSTIME** flag.

clock_nanosleep() is never restarted after being interrupted by a signal handler, regardless of the use of the *sigaction*(2) **SA_RESTART** flag.

The *remain* argument is unused, and unnecessary, when *flags* is **TIMER_ABSTIME**. (An absolute sleep can be restarted using the same *t* argument.)

POSIX.1 specifies that **clock_nanosleep**() has no effect on signals dispositions or the signal mask.

POSIX.1 specifies that after changing the value of the **CLOCK_REALTIME** clock via *clock_settime*(2), the new clock value shall be used to determine the time at which a thread blocked on an absolute **clock_nanosleep**() will wake up; if the new clock value falls past the end of the sleep interval, then the **clock_nanosleep**() call will return immediately.

POSIX.1 specifies that changing the value of the **CLOCK_REALTIME** clock via *clock_settime*(2) shall have no effect on a thread that is blocked on a relative **clock_nanosleep**().

SEE ALSO

clock_getres(2), *nanosleep*(2), *restart_syscall*(2), *timer_create*(2), *sleep*(3), *timespec*(3), *usleep*(3), *time*(7)

NAME

clone, __clone2, clone3 − create a child process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

/* Prototype for the glibc wrapper function */

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *_Nullable), void *stack, int flags,
          void *_Nullable arg, ...  /* pid_t *_Nullable parent_tid,
                                       void *_Nullable tls,
                                       pid_t *_Nullable child_tid */ );
```

/* For the prototype of the raw clone() system call, see NOTES */

```
#include <linux/sched.h>    /* Definition of struct clone_args */
#include <sched.h>          /* Definition of CLONE_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_clone3, struct clone_args *cl_args, size_t size);
```

Note: glibc provides no wrapper for **clone3**(), necessitating the use of *syscall*(2).

DESCRIPTION

These system calls create a new ("child") process, in a manner similar to *fork*(2).

By contrast with *fork*(2), these system calls provide more precise control over what pieces of execution context are shared between the calling process and the child process. For example, using these system calls, the caller can control whether or not the two processes share the virtual address space, the table of file descriptors, and the table of signal handlers. These system calls also allow the new child process to be placed in separate *namespaces*(7).

Note that in this manual page, "calling process" normally corresponds to "parent process". But see the descriptions of **CLONE_PARENT** and **CLONE_THREAD** below.

This page describes the following interfaces:

- The glibc **clone**() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.
- The newer **clone3**() system call.

In the remainder of this page, the terminology "the clone call" is used when noting details that apply to all of these interfaces.

The clone() wrapper function

When the child process is created with the **clone**() wrapper function, it commences execution by calling the function pointed to by the argument *fn*. (This differs from *fork*(2), where execution continues in the child from the point of the *fork*(2) call.) The *arg* argument is passed as the argument of the function *fn*.

When the *fn*(*arg*) function returns, the child process terminates. The integer returned by *fn* is the exit status for the child process. The child process may also terminate explicitly by calling *exit*(2) or after receiving a fatal signal.

The *stack* argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to **clone**(). Stacks grow downward on all processors that run Linux (except the HP PA processors), so *stack* usually points to the topmost address of the memory space set up for the child stack. Note that **clone**() does not provide a means whereby the caller can inform the kernel of the size of the stack area.

The remaining arguments to **clone**() are discussed below.

clone3()

The **clone3**() system call provides a superset of the functionality of the older **clone**() interface. It also provides a number of API improvements, including: space for additional flags bits; cleaner separation in the use of various arguments; and the ability to specify the size of the child's stack area.

As with *fork*(2), **clone3**() returns in both the parent and the child. It returns 0 in the child process and returns the PID of the child in the parent.

The *cl_args* argument of **clone3**() is a structure of the following form:

```
struct clone_args {
    u64 flags;        /* Flags bit mask */
    u64 pidfd;        /* Where to store PID file descriptor
                         (int *) */
    u64 child_tid;    /* Where to store child TID,
                         in child's memory (pid_t *) */
    u64 parent_tid;   /* Where to store child TID,
                         in parent's memory (pid_t *) */
    u64 exit_signal;  /* Signal to deliver to parent on
                         child termination */
    u64 stack;        /* Pointer to lowest byte of stack */
    u64 stack_size;   /* Size of stack */
    u64 tls;          /* Location of new TLS */
    u64 set_tid;      /* Pointer to a pid_t array
                         (since Linux 5.5) */
    u64 set_tid_size; /* Number of elements in set_tid
                         (since Linux 5.5) */
    u64 cgroup;       /* File descriptor for target cgroup
                         of child (since Linux 5.7) */
};
```

The *size* argument that is supplied to **clone3**() should be initialized to the size of this structure. (The existence of the *size* argument permits future extensions to the *clone_args* structure.)

The stack for the child process is specified via *cl_args.stack*, which points to the lowest byte of the stack area, and *cl_args.stack_size*, which specifies the size of the stack in bytes. In the case where the **CLONE_VM** flag (see below) is specified, a stack must be explicitly allocated and specified. Otherwise, these two fields can be specified as NULL and 0, which causes the child to use the same stack area as the parent (in the child's own virtual address space).

The remaining fields in the *cl_args* argument are discussed below.

Equivalence between clone() and clone3() arguments

Unlike the older **clone**() interface, where arguments are passed individually, in the newer **clone3**() interface the arguments are packaged into the *clone_args* structure shown above. This structure allows for a superset of the information passed via the **clone**() arguments.

The following table shows the equivalence between the arguments of **clone**() and the fields in the *clone_args* argument supplied to **clone3**():

clone()	**clone3()**	**Notes**
	cl_args field	
flags & ~0xff	*flags*	For most flags; details below
parent_tid	*pidfd*	See CLONE_PIDFD
child_tid	*child_tid*	See CLONE_CHILD_SETTID
parent_tid	*parent_tid*	See CLONE_PARENT_SETTID
flags & 0xff	*exit_signal*	
stack	*stack*	
---	*stack_size*	
tls	*tls*	See CLONE_SETTLS
---	*set_tid*	See below for details
---	*set_tid_size*	
---	*cgroup*	See CLONE_INTO_CGROUP

The child termination signal

When the child process terminates, a signal may be sent to the parent. The termination signal is specified in the low byte of *flags* (**clone**()) or in *cl_args.exit_signal* (**clone3**()). If this signal is specified as anything other than **SIGCHLD**, then the parent process must specify the **__WALL** or **__WCLONE** options when waiting for the child with *wait*(2). If no signal (i.e., zero) is specified, then the parent process is not signaled when the child terminates.

The set_tid array

By default, the kernel chooses the next sequential PID for the new process in each of the PID namespaces where it is present. When creating a process with **clone3**(), the *set_tid* array (available since Linux 5.5) can be used to select specific PIDs for the process in some or all of the PID namespaces where it is present. If the PID of the newly created process should be set only for the current PID namespace or in the newly created PID namespace (if *flags* contains **CLONE_NEWPID**) then the first element in the *set_tid* array has to be the desired PID and *set_tid_size* needs to be 1.

If the PID of the newly created process should have a certain value in multiple PID namespaces, then the *set_tid* array can have multiple entries. The first entry defines the PID in the most deeply nested PID namespace and each of the following entries contains the PID in the corresponding ancestor PID namespace. The number of PID namespaces in which a PID should be set is defined by *set_tid_size* which cannot be larger than the number of currently nested PID namespaces.

To create a process with the following PIDs in a PID namespace hierarchy:

PID NS level	Requested PID	Notes
0	31496	Outermost PID namespace
1	42	
2	7	Innermost PID namespace

Set the array to:

```
set_tid[0] = 7;
set_tid[1] = 42;
set_tid[2] = 31496;
set_tid_size = 3;
```

If only the PIDs in the two innermost PID namespaces need to be specified, set the array to:

```
set_tid[0] = 7;
set_tid[1] = 42;
set_tid_size = 2;
```

The PID in the PID namespaces outside the two innermost PID namespaces is selected the same way as any other PID is selected.

The *set_tid* feature requires **CAP_SYS_ADMIN** or (since Linux 5.9) **CAP_CHECKPOINT_RESTORE** in all owning user namespaces of the target PID namespaces.

Callers may only choose a PID greater than 1 in a given PID namespace if an **init** process (i.e., a process with PID 1) already exists in that namespace. Otherwise the PID entry for this PID namespace must be 1.

The flags mask

Both **clone**() and **clone3**() allow a flags bit mask that modifies their behavior and allows the caller to specify what is shared between the calling process and the child process. This bit mask—the *flags* argument of **clone**() or the *cl_args.flags* field passed to **clone3**()—is referred to as the *flags* mask in the remainder of this page.

The *flags* mask is specified as a bitwise OR of zero or more of the constants listed below. Except as noted below, these flags are available (and have the same effect) in both **clone**() and **clone3**().

CLONE_CHILD_CLEARTID (since Linux 2.5.49)

Clear (zero) the child thread ID at the location pointed to by *child_tid* (**clone**()) or *cl_args.child_tid* (**clone3**()) in child memory when the child exits, and do a wakeup on the futex at that address. The address involved may be changed by the *set_tid_address*(2) system call. This is used by threading libraries.

CLONE_CHILD_SETTID (since Linux 2.5.49)
Store the child thread ID at the location pointed to by *child_tid* (**clone**()) or *cl_args.child_tid* (**clone3**()) in the child's memory. The store operation completes before the clone call returns control to user space in the child process. (Note that the store operation may not have completed before the clone call returns in the parent process, which is relevant if the **CLONE_VM** flag is also employed.)

CLONE_CLEAR_SIGHAND (since Linux 5.5)
By default, signal dispositions in the child thread are the same as in the parent. If this flag is specified, then all signals that are handled in the parent (and not set to **SIG_IGN**) are reset to their default dispositions (**SIG_DFL**) in the child.

Specifying this flag together with **CLONE_SIGHAND** is nonsensical and disallowed.

CLONE_DETACHED (historical)
For a while (during the Linux 2.5 development series) there was a **CLONE_DETACHED** flag, which caused the parent not to receive a signal when the child terminated. Ultimately, the effect of this flag was subsumed under the **CLONE_THREAD** flag and by the time Linux 2.6.0 was released, this flag had no effect. Starting in Linux 2.6.2, the need to give this flag together with **CLONE_THREAD** disappeared.

This flag is still defined, but it is usually ignored when calling **clone**(). However, see the description of **CLONE_PIDFD** for some exceptions.

CLONE_FILES (since Linux 2.0)
If **CLONE_FILES** is set, the calling process and the child process share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags (using the *fcntl*(2) **F_SETFD** operation), the other process is also affected. If a process sharing a file descriptor table calls *execve*(2), its file descriptor table is duplicated (unshared).

If **CLONE_FILES** is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of the clone call. Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process. Note, however, that the duplicated file descriptors in the child refer to the same open file descriptions as the corresponding file descriptors in the calling process, and thus share file offsets and file status flags (see *open*(2)).

CLONE_FS (since Linux 2.0)
If **CLONE_FS** is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. Any call to *chroot*(2), *chdir*(2), or *umask*(2) performed by the calling process or the child process also affects the other process.

If **CLONE_FS** is not set, the child process works on a copy of the filesystem information of the calling process at the time of the clone call. Calls to *chroot*(2), *chdir*(2), or *umask*(2) performed later by one of the processes do not affect the other process.

CLONE_INTO_CGROUP (since Linux 5.7)
By default, a child process is placed in the same version 2 cgroup as its parent. The **CLONE_INTO_CGROUP** flag allows the child process to be created in a different version 2 cgroup. (Note that **CLONE_INTO_CGROUP** has effect only for version 2 cgroups.)

In order to place the child process in a different cgroup, the caller specifies **CLONE_INTO_CGROUP** in *cl_args.flags* and passes a file descriptor that refers to a version 2 cgroup in the *cl_args.cgroup* field. (This file descriptor can be obtained by opening a cgroup v2 directory using either the **O_RDONLY** or the **O_PATH** flag.) Note that all of the usual restrictions (described in *cgroups*(7)) on placing a process into a version 2 cgroup apply.

Among the possible use cases for **CLONE_INTO_CGROUP** are the following:

- Spawning a process into a cgroup different from the parent's cgroup makes it possible for a service manager to directly spawn new services into dedicated cgroups. This eliminates the accounting jitter that would be caused if the child process was first created in the same cgroup as the parent and then moved into the target cgroup. Furthermore, spawning the

child process directly into a target cgroup is significantly cheaper than moving the child process into the target cgroup after it has been created.

- The **CLONE_INTO_CGROUP** flag also allows the creation of frozen child processes by spawning them into a frozen cgroup. (See *cgroups*(7) for a description of the freezer controller.)
- For threaded applications (or even thread implementations which make use of cgroups to limit individual threads), it is possible to establish a fixed cgroup layout before spawning each thread directly into its target cgroup.

CLONE_IO (since Linux 2.6.25)
If **CLONE_IO** is set, then the new process shares an I/O context with the calling process. If this flag is not set, then (as with *fork*(2)) the new process has its own I/O context.

The I/O context is the I/O scope of the disk scheduler (i.e., what the I/O scheduler uses to model scheduling of a process's I/O). If processes share the same I/O context, they are treated as one by the I/O scheduler. As a consequence, they get to share disk time. For some I/O schedulers, if two processes share an I/O context, they will be allowed to interleave their disk access. If several threads are doing I/O on behalf of the same process (**aio_read**(3), for instance), they should employ **CLONE_IO** to get better I/O performance.

If the kernel is not configured with the **CONFIG_BLOCK** option, this flag is a no-op.

CLONE_NEWCGROUP (since Linux 4.6)
Create the process in a new cgroup namespace. If this flag is not set, then (as with *fork*(2)) the process is created in the same cgroup namespaces as the calling process.

For further information on cgroup namespaces, see *cgroup_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWCGROUP**.

CLONE_NEWIPC (since Linux 2.6.19)
If **CLONE_NEWIPC** is set, then create the process in a new IPC namespace. If this flag is not set, then (as with *fork*(2)), the process is created in the same IPC namespace as the calling process.

For further information on IPC namespaces, see *ipc_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWIPC**. This flag can't be specified in conjunction with **CLONE_SYSVSEM**.

CLONE_NEWNET (since Linux 2.6.24)
(The implementation of this flag was completed only by about Linux 2.6.29.)

If **CLONE_NEWNET** is set, then create the process in a new network namespace. If this flag is not set, then (as with *fork*(2)) the process is created in the same network namespace as the calling process.

For further information on network namespaces, see *network_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWNET**.

CLONE_NEWNS (since Linux 2.4.19)
If **CLONE_NEWNS** is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent. If **CLONE_NEWNS** is not set, the child lives in the same mount namespace as the parent.

For further information on mount namespaces, see *namespaces*(7) and *mount_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWNS**. It is not permitted to specify both **CLONE_NEWNS** and **CLONE_FS** in the same clone call.

CLONE_NEWPID (since Linux 2.6.24)
If **CLONE_NEWPID** is set, then create the process in a new PID namespace. If this flag is not set, then (as with *fork*(2)) the process is created in the same PID namespace as the calling process.

For further information on PID namespaces, see *namespaces*(7) and *pid_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWPID**. This flag can't be specified in conjunction with **CLONE_THREAD**.

CLONE_NEWUSER
(This flag first became meaningful for **clone**() in Linux 2.6.23, the current **clone**() semantics were merged in Linux 3.5, and the final pieces to make the user namespaces completely usable were merged in Linux 3.8.)

If **CLONE_NEWUSER** is set, then create the process in a new user namespace. If this flag is not set, then (as with *fork*(2)) the process is created in the same user namespace as the calling process.

For further information on user namespaces, see *namespaces*(7) and *user_namespaces*(7).

Before Linux 3.8, use of **CLONE_NEWUSER** required that the caller have three capabilities: **CAP_SYS_ADMIN**, **CAP_SETUID**, and **CAP_SETGID**. Starting with Linux 3.8, no privileges are needed to create a user namespace.

This flag can't be specified in conjunction with **CLONE_THREAD** or **CLONE_PARENT**. For security reasons, **CLONE_NEWUSER** cannot be specified in conjunction with **CLONE_FS**.

CLONE_NEWUTS (since Linux 2.6.19)
If **CLONE_NEWUTS** is set, then create the process in a new UTS namespace, whose identifiers are initialized by duplicating the identifiers from the UTS namespace of the calling process. If this flag is not set, then (as with *fork*(2)) the process is created in the same UTS namespace as the calling process.

For further information on UTS namespaces, see *uts_namespaces*(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWUTS**.

CLONE_PARENT (since Linux 2.3.12)
If **CLONE_PARENT** is set, then the parent of the new child (as returned by *getppid*(2)) will be the same as that of the calling process.

If **CLONE_PARENT** is not set, then (as with *fork*(2)) the child's parent is the calling process.

Note that it is the parent process, as returned by *getppid*(2), which is signaled when the child terminates, so that if **CLONE_PARENT** is set, then the parent of the calling process, rather than the calling process itself, is signaled.

The **CLONE_PARENT** flag can't be used in clone calls by the global init process (PID 1 in the initial PID namespace) and init processes in other PID namespaces. This restriction prevents the creation of multi-rooted process trees as well as the creation of unreapable zombies in the initial PID namespace.

CLONE_PARENT_SETTID (since Linux 2.5.49)
Store the child thread ID at the location pointed to by *parent_tid* (**clone**()) or *cl_args.parent_tid* (**clone3**()) in the parent's memory. (In Linux 2.5.32-2.5.48 there was a flag **CLONE_SETTID** that did this.) The store operation completes before the clone call returns control to user space.

CLONE_PID (Linux 2.0 to Linux 2.5.15)
If **CLONE_PID** is set, the child process is created with the same process ID as the calling process. This is good for hacking the system, but otherwise of not much use. From Linux 2.3.21 onward, this flag could be specified only by the system boot process (PID 0). The flag disappeared completely from the kernel sources in Linux 2.5.16. Subsequently, the kernel silently ignored this bit if it was specified in the *flags* mask. Much later, the same bit was recycled for use as the **CLONE_PIDFD** flag.

CLONE_PIDFD (since Linux 5.2)
If this flag is specified, a PID file descriptor referring to the child process is allocated and placed at a specified location in the parent's memory. The close-on-exec flag is set on this new file descriptor. PID file descriptors can be used for the purposes described in *pidfd_open*(2).

- When using **clone3**(), the PID file descriptor is placed at the location pointed to by *cl_args.pidfd*.
- When using **clone**(), the PID file descriptor is placed at the location pointed to by *parent_tid*. Since the *parent_tid* argument is used to return the PID file descriptor, **CLONE_PIDFD** cannot be used with **CLONE_PARENT_SETTID** when calling **clone**().

It is currently not possible to use this flag together with **CLONE_THREAD.** This means that the process identified by the PID file descriptor will always be a thread group leader.

If the obsolete **CLONE_DETACHED** flag is specified alongside **CLONE_PIDFD** when calling **clone**(), an error is returned. An error also results if **CLONE_DETACHED** is specified when calling **clone3**(). This error behavior ensures that the bit corresponding to **CLONE_DETACHED** can be reused for further PID file descriptor features in the future.

CLONE_PTRACE (since Linux 2.2)
If **CLONE_PTRACE** is specified, and the calling process is being traced, then trace the child also (see *ptrace*(2)).

CLONE_SETTLS (since Linux 2.5.32)
The TLS (Thread Local Storage) descriptor is set to *tls*.

The interpretation of *tls* and the resulting effect is architecture dependent. On x86, *tls* is interpreted as a *struct user_desc* * (see *set_thread_area*(2)). On x86-64 it is the new value to be set for the %fs base register (see the **ARCH_SET_FS** argument to *arch_prctl*(2)). On architectures with a dedicated TLS register, it is the new value of that register.

Use of this flag requires detailed knowledge and generally it should not be used except in libraries implementing threading.

CLONE_SIGHAND (since Linux 2.0)
If **CLONE_SIGHAND** is set, the calling process and the child process share the same table of signal handlers. If the calling process or child process calls *sigaction*(2) to change the behavior associated with a signal, the behavior is changed in the other process as well. However, the calling process and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock signals using *sigprocmask*(2) without affecting the other process.

If **CLONE_SIGHAND** is not set, the child process inherits a copy of the signal handlers of the calling process at the time of the clone call. Calls to *sigaction*(2) performed later by one of the processes have no effect on the other process.

Since Linux 2.6.0, the *flags* mask must also include **CLONE_VM** if **CLONE_SIGHAND** is specified.

CLONE_STOPPED (since Linux 2.6.0)
If **CLONE_STOPPED** is set, then the child is initially stopped (as though it was sent a **SIGSTOP** signal), and must be resumed by sending it a **SIGCONT** signal.

This flag was *deprecated* from Linux 2.6.25 onward, and was *removed* altogether in Linux 2.6.38. Since then, the kernel silently ignores it without error. Starting with Linux 4.6, the same bit was reused for the **CLONE_NEWCGROUP** flag.

CLONE_SYSVSEM (since Linux 2.5.10)
If **CLONE_SYSVSEM** is set, then the child and the calling process share a single list of System V semaphore adjustment (*semadj*) values (see *semop*(2)). In this case, the shared list accumulates *semadj* values across all processes sharing the list, and semaphore adjustments are performed only when the last process that is sharing the list terminates (or ceases sharing the list using *unshare*(2)). If this flag is not set, then the child has a separate *semadj* list that is initially empty.

CLONE_THREAD (since Linux 2.4.0)
If **CLONE_THREAD** is set, the child is placed in the same thread group as the calling process. To make the remainder of the discussion of **CLONE_THREAD** more readable, the term "thread" is used to refer to the processes within a thread group.

Thread groups were a feature added in Linux 2.4 to support the POSIX threads notion of a set of threads that share a single PID. Internally, this shared PID is the so-called thread group identifier (TGID) for the thread group. Since Linux 2.4, calls to *getpid*(2) return the TGID of the caller.

The threads within a group can be distinguished by their (system-wide) unique thread IDs (TID). A new thread's TID is available as the function result returned to the caller, and a thread can obtain its own TID using *gettid*(2).

When a clone call is made without specifying **CLONE_THREAD**, then the resulting thread is placed in a new thread group whose TGID is the same as the thread's TID. This thread is the *leader* of the new thread group.

A new thread created with **CLONE_THREAD** has the same parent process as the process that made the clone call (i.e., like **CLONE_PARENT**), so that calls to *getppid*(2) return the same value for all of the threads in a thread group. When a **CLONE_THREAD** thread terminates, the thread that created it is not sent a **SIGCHLD** (or other termination) signal; nor can the status of such a thread be obtained using *wait*(2). (The thread is said to be *detached*.)

After all of the threads in a thread group terminate the parent process of the thread group is sent a **SIGCHLD** (or other termination) signal.

If any of the threads in a thread group performs an *execve*(2), then all threads other than the thread group leader are terminated, and the new program is executed in the thread group leader.

If one of the threads in a thread group creates a child using *fork*(2), then any thread in the group can *wait*(2) for that child.

Since Linux 2.5.35, the *flags* mask must also include **CLONE_SIGHAND** if **CLONE_THREAD** is specified (and note that, since Linux 2.6.0, **CLONE_SIGHAND** also requires **CLONE_VM** to be included).

Signal dispositions and actions are process-wide: if an unhandled signal is delivered to a thread, then it will affect (terminate, stop, continue, be ignored in) all members of the thread group.

Each thread has its own signal mask, as set by *sigprocmask*(2).

A signal may be process-directed or thread-directed. A process-directed signal is targeted at a thread group (i.e., a TGID), and is delivered to an arbitrarily selected thread from among those that are not blocking the signal. A signal may be process-directed because it was generated by the kernel for reasons other than a hardware exception, or because it was sent using *kill*(2) or *sigqueue*(3). A thread-directed signal is targeted at (i.e., delivered to) a specific thread. A signal may be thread directed because it was sent using *tgkill*(2) or *pthread_sigqueue*(3), or because the thread executed a machine language instruction that triggered a hardware exception (e.g., invalid memory access triggering **SIGSEGV** or a floating-point exception triggering **SIGFPE**).

A call to *sigpending*(2) returns a signal set that is the union of the pending process-directed signals and the signals that are pending for the calling thread.

If a process-directed signal is delivered to a thread group, and the thread group has installed a handler for the signal, then the handler is invoked in exactly one, arbitrarily selected member of the thread group that has not blocked the signal. If multiple threads in a group are waiting to accept the same signal using *sigwaitinfo*(2), the kernel will arbitrarily select one of these threads to receive the signal.

CLONE_UNTRACED (since Linux 2.5.46)
If **CLONE_UNTRACED** is specified, then a tracing process cannot force **CLONE_PTRACE** on this child process.

CLONE_VFORK (since Linux 2.2)
If **CLONE_VFORK** is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to *execve*(2) or *_exit*(2) (as with *vfork*(2)).

If **CLONE_VFORK** is not set, then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

CLONE_VM (since Linux 2.0)
If **CLONE_VM** is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with *mmap*(2) or *munmap*(2) by the child or calling process also affects the other process.

If **CLONE_VM** is not set, the child process runs in a separate copy of the memory space of the calling process at the time of the clone call. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with *fork*(2).

If the **CLONE_VM** flag is specified and the **CLONE_VFORK** flag is not specified, then any alternate signal stack that was established by *sigaltstack*(2) is cleared in the child process.

RETURN VALUE

On success, the thread ID of the child process is returned in the caller's thread of execution. On failure, −1 is returned in the caller's context, no child process is created, and *errno* is set to indicate the error.

ERRORS

EACCES (**clone3**() only)
CLONE_INTO_CGROUP was specified in *cl_args.flags*, but the restrictions (described in *cgroups*(7)) on placing the child process into the version 2 cgroup referred to by *cl_args.cgroup* are not met.

EAGAIN
Too many processes are already running; see *fork*(2).

EBUSY (**clone3**() only)
CLONE_INTO_CGROUP was specified in *cl_args.flags*, but the file descriptor specified in *cl_args.cgroup* refers to a version 2 cgroup in which a domain controller is enabled.

EEXIST (**clone3**() only)
One (or more) of the PIDs specified in *set_tid* already exists in the corresponding PID namespace.

EINVAL
Both **CLONE_SIGHAND** and **CLONE_CLEAR_SIGHAND** were specified in the *flags* mask.

EINVAL
CLONE_SIGHAND was specified in the *flags* mask, but **CLONE_VM** was not. (Since Linux 2.6.0.)

EINVAL
CLONE_THREAD was specified in the *flags* mask, but **CLONE_SIGHAND** was not. (Since Linux 2.5.35.)

EINVAL
CLONE_THREAD was specified in the *flags* mask, but the current process previously called *unshare*(2) with the **CLONE_NEWPID** flag or used *setns*(2) to reassociate itself with a PID namespace.

EINVAL
Both **CLONE_FS** and **CLONE_NEWNS** were specified in the *flags* mask.

EINVAL (since Linux 3.9)
Both **CLONE_NEWUSER** and **CLONE_FS** were specified in the *flags* mask.

EINVAL
Both **CLONE_NEWIPC** and **CLONE_SYSVSEM** were specified in the *flags* mask.

EINVAL
CLONE_NEWPID and one (or both) of **CLONE_THREAD** or **CLONE_PARENT** were specified in the *flags* mask.

EINVAL
CLONE_NEWUSER and **CLONE_THREAD** were specified in the *flags* mask.

EINVAL (since Linux 2.6.32)
CLONE_PARENT was specified, and the caller is an init process.

EINVAL
Returned by the glibc **clone**() wrapper function when *fn* or *stack* is specified as NULL.

EINVAL
CLONE_NEWIPC was specified in the *flags* mask, but the kernel was not configured with the **CONFIG_SYSVIPC** and **CONFIG_IPC_NS** options.

EINVAL
CLONE_NEWNET was specified in the *flags* mask, but the kernel was not configured with the **CONFIG_NET_NS** option.

EINVAL
CLONE_NEWPID was specified in the *flags* mask, but the kernel was not configured with the **CONFIG_PID_NS** option.

EINVAL
CLONE_NEWUSER was specified in the *flags* mask, but the kernel was not configured with the **CONFIG_USER_NS** option.

EINVAL
CLONE_NEWUTS was specified in the *flags* mask, but the kernel was not configured with the **CONFIG_UTS_NS** option.

EINVAL
stack is not aligned to a suitable boundary for this architecture. For example, on aarch64, *stack* must be a multiple of 16.

EINVAL (**clone3**() only)
CLONE_DETACHED was specified in the *flags* mask.

EINVAL (**clone**() only)
CLONE_PIDFD was specified together with **CLONE_DETACHED** in the *flags* mask.

EINVAL
CLONE_PIDFD was specified together with **CLONE_THREAD** in the *flags* mask.

EINVAL (**clone**() only)
CLONE_PIDFD was specified together with **CLONE_PARENT_SETTID** in the *flags* mask.

EINVAL (**clone3**() only)
set_tid_size is greater than the number of nested PID namespaces.

EINVAL (**clone3**() only)
One of the PIDs specified in *set_tid* was an invalid.

EINVAL (**clone3**() only)
CLONE_THREAD or **CLONE_PARENT** was specified in the *flags* mask, but a signal was specified in *exit_signal*.

EINVAL (AArch64 only, Linux 4.6 and earlier)
stack was not aligned to a 128-bit boundary.

ENOMEM
Cannot allocate sufficient memory to allocate a task structure for the child, or to copy those parts of the caller's context that need to be copied.

ENOSPC (since Linux 3.7)
CLONE_NEWPID was specified in the *flags* mask, but the limit on the nesting depth of PID namespaces would have been exceeded; see *pid_namespaces*(7).

ENOSPC (since Linux 4.9; beforehand **EUSERS**)
CLONE_NEWUSER was specified in the *flags* mask, and the call would cause the limit on the number of nested user namespaces to be exceeded. See *user_namespaces*(7).

From Linux 3.11 to Linux 4.8, the error diagnosed in this case was **EUSERS**.

ENOSPC (since Linux 4.9)
One of the values in the *flags* mask specified the creation of a new user namespace, but doing so would have caused the limit defined by the corresponding file in */proc/sys/user* to be exceeded. For further details, see *namespaces*(7).

EOPNOTSUPP (**clone3**() only)
CLONE_INTO_CGROUP was specified in *cl_args.flags*, but the file descriptor specified in *cl_args.cgroup* refers to a version 2 cgroup that is in the *domain invalid* state.

EPERM
CLONE_NEWCGROUP, **CLONE_NEWIPC**, **CLONE_NEWNET**, **CLONE_NEWNS**, **CLONE_NEWPID**, or **CLONE_NEWUTS** was specified by an unprivileged process (process without **CAP_SYS_ADMIN**).

EPERM
CLONE_PID was specified by a process other than process 0. (This error occurs only on Linux 2.5.15 and earlier.)

EPERM
CLONE_NEWUSER was specified in the *flags* mask, but either the effective user ID or the effective group ID of the caller does not have a mapping in the parent namespace (see *user_namespaces*(7)).

EPERM (since Linux 3.9)
CLONE_NEWUSER was specified in the *flags* mask and the caller is in a chroot environment (i.e., the caller's root directory does not match the root directory of the mount namespace in which it resides).

EPERM (**clone3**() only)
set_tid_size was greater than zero, and the caller lacks the **CAP_SYS_ADMIN** capability in one or more of the user namespaces that own the corresponding PID namespaces.

ERESTARTNOINTR (since Linux 2.6.17)
System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

EUSERS (Linux 3.11 to Linux 4.8)
CLONE_NEWUSER was specified in the *flags* mask, and the limit on the number of nested user namespaces would be exceeded. See the discussion of the **ENOSPC** error above.

VERSIONS

The glibc **clone**() wrapper function makes some changes in the memory pointed to by *stack* (changes required to set the stack up correctly for the child) *before* invoking the **clone**() system call. So, in cases where **clone**() is used to recursively create children, do not use the buffer employed for the parent's stack as the stack of the child.

On i386, **clone**() should not be called through vsyscall, but directly through *int $0x80*.

C library/kernel differences

The raw **clone**() system call corresponds more closely to *fork*(2) in that execution in the child continues from the point of the call. As such, the *fn* and *arg* arguments of the **clone**() wrapper function are omitted.

In contrast to the glibc wrapper, the raw **clone**() system call accepts NULL as a *stack* argument (and **clone3**() likewise allows *cl_args.stack* to be NULL). In this case, the child uses a duplicate of the parent's stack. (Copy-on-write semantics ensure that the child gets separate copies of stack pages when either process modifies the stack.) In this case, for correct operation, the **CLONE_VM** option should not be specified. (If the child *shares* the parent's memory because of the use of the **CLONE_VM** flag, then no copy-on-write duplication occurs and chaos is likely to result.)

The order of the arguments also differs in the raw system call, and there are variations in the arguments across architectures, as detailed in the following paragraphs.

The raw system call interface on x86-64 and some other architectures (including sh, tile, and alpha) is:

```
long clone(unsigned long flags, void *stack,
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

On x86-32, and several other common architectures (including score, ARM, ARM 64, PA-RISC, arc, Power PC, xtensa, and MIPS), the order of the last two arguments is reversed:

```
long clone(unsigned long flags, void *stack,
           int *parent_tid, unsigned long tls,
           int *child_tid);
```

On the cris and s390 architectures, the order of the first two arguments is reversed:

```
long clone(void *stack, unsigned long flags,
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

On the microblaze architecture, an additional argument is supplied:

```
long clone(unsigned long flags, void *stack,
           int stack_size,         /* Size of stack */
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

blackfin, m68k, and sparc

The argument-passing conventions on blackfin, m68k, and sparc are different from the descriptions above. For details, see the kernel (and glibc) source.

ia64

On ia64, a different interface is used:

```
int __clone2(int (*fn)(void *),
             void *stack_base, size_t stack_size,
             int flags, void *arg, ...
            /* pid_t *parent_tid, struct user_desc *tls,
             pid_t *child_tid */ );
```

The prototype shown above is for the glibc wrapper function; for the system call itself, the prototype can be described as follows (it is identical to the **clone**() prototype on microblaze):

```
long clone2(unsigned long flags, void *stack_base,
            int stack_size,         /* Size of stack */
            int *parent_tid, int *child_tid,
            unsigned long tls);
```

__clone2() operates in the same way as **clone**(), except that *stack_base* points to the lowest address of the child's stack area, and *stack_size* specifies the size of the stack pointed to by *stack_base*.

STANDARDS

Linux.

HISTORY

clone3()

Linux 5.3.

Linux 2.4 and earlier

In the Linux 2.4.x series, **CLONE_THREAD** generally does not make the parent of the new thread the same as the parent of the calling process. However, from Linux 2.4.7 to Linux 2.4.18 the **CLONE_THREAD** flag implied the **CLONE_PARENT** flag (as in Linux 2.6.0 and later).

In Linux 2.4 and earlier, **clone**() does not take arguments *parent_tid*, *tls*, and *child_tid*.

NOTES

One use of these system calls is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.

The *kcmp*(2) system call can be used to test whether two processes share various resources such as a file descriptor table, System V semaphore undo operations, or a virtual address space.

Handlers registered using *pthread_atfork*(3) are not executed during a clone call.

BUGS

GNU C library versions 2.3.4 up to and including 2.24 contained a wrapper function for *getpid*(2) that performed caching of PIDs. This caching relied on support in the glibc wrapper for **clone**(), but

limitations in the implementation meant that the cache was not up to date in some circumstances. In particular, if a signal was delivered to the child immediately after the **clone**() call, then a call to *getpid*(2) in a handler for the signal could return the PID of the calling process ("the parent"), if the clone wrapper had not yet had a chance to update the PID cache in the child. (This discussion ignores the case where the child was created using **CLONE_THREAD**, when *getpid*(2) *should* return the same value in the child and in the process that called **clone**(), since the caller and the child are in the same thread group. The stale-cache problem also does not occur if the *flags* argument includes **CLONE_VM**.) To get the truth, it was sometimes necessary to use code such as the following:

```
#include <syscall.h>

pid_t mypid;

mypid = syscall(SYS_getpid);
```

Because of the stale-cache problem, as well as other problems noted in *getpid*(2), the PID caching feature was removed in glibc 2.25.

EXAMPLES

The following program demonstrates the use of **clone**() to create a child process that executes in a separate UTS namespace. The child changes the hostname in its UTS namespace. Both parent and child then display the system hostname, making it possible to see that the hostname differs in the UTS namespaces of the parent and child. For an example of the use of this program, see *setns*(2).

Within the sample program, we allocate the memory that is to be used for the child's stack using *mmap*(2) rather than *malloc*(3) for the following reasons:

- *mmap*(2) allocates a block of memory that starts on a page boundary and is a multiple of the page size. This is useful if we want to establish a guard page (a page with protection **PROT_NONE**) at the end of the stack using *mprotect*(2).
- We can specify the **MAP_STACK** flag to request a mapping that is suitable for a stack. For the moment, this flag is a no-op on Linux, but it exists and has effect on some other systems, so we should include it for portability.

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <sched.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/utsname.h>
#include <sys/wait.h>
#include <unistd.h>

static int              /* Start function for cloned child */
childFunc(void *arg)
{
    struct utsname uts;

    /* Change hostname in UTS namespace of child. */

    if (sethostname(arg, strlen(arg)) == -1)
        err(EXIT_FAILURE, "sethostname");

    /* Retrieve and display hostname. */

    if (uname(&uts) == -1)
        err(EXIT_FAILURE, "uname");
```

```
    printf("uts.nodename in child:  %s\n", uts.nodename);

    /* Keep the namespace open for a while, by sleeping.
       This allows some experimentation--for example, another
       process might join the namespace. */

    sleep(200);

    return 0;           /* Child terminates now */
}

#define STACK_SIZE (1024 * 1024)    /* Stack size for cloned child */

int
main(int argc, char *argv[])
{
    char            *stack;         /* Start of stack buffer */
    char            *stackTop;      /* End of stack buffer */
    pid_t           pid;
    struct utsname  uts;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    /* Allocate memory to be used for the stack of the child. */

    stack = mmap(NULL, STACK_SIZE, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
    if (stack == MAP_FAILED)
        err(EXIT_FAILURE, "mmap");

    stackTop = stack + STACK_SIZE;  /* Assume stack grows downward */

    /* Create child that has its own UTS namespace;
       child commences execution in childFunc(). */

    pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv[1]);
    if (pid == -1)
        err(EXIT_FAILURE, "clone");
    printf("clone() returned %jd\n", (intmax_t) pid);

    /* Parent falls through to here */

    sleep(1);           /* Give child time to change its hostname */

    /* Display hostname in parent's UTS namespace. This will be
       different from hostname in child's UTS namespace. */

    if (uname(&uts) == -1)
        err(EXIT_FAILURE, "uname");
    printf("uts.nodename in parent: %s\n", uts.nodename);

    if (waitpid(pid, NULL, 0) == -1)    /* Wait for child */
        err(EXIT_FAILURE, "waitpid");
    printf("child has terminated\n");

    exit(EXIT_SUCCESS);
```

```
}
```

SEE ALSO

fork(2), *futex*(2), *getpid*(2), *gettid*(2), *kcmp*(2), *mmap*(2), *pidfd_open*(2), *set_thread_area*(2), *set_tid_address*(2), *setns*(2), *tkill*(2), *unshare*(2), *wait*(2), *capabilities*(7), *namespaces*(7), *pthreads*(7)

NAME

close − close a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see *fcntl*(2)) held on the file it was associated with, and owned by the process, are removed regardless of the file descriptor that was used to obtain the lock. This has some unfortunate consequences and one should be extra careful when using advisory record locking. See *fcntl*(2) for discussion of the risks and consequences as well as for the (probably preferred) open file description locks.

If *fd* is the last file descriptor referring to the underlying open file description (see *open*(2)), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using *unlink*(2), the file is deleted.

RETURN VALUE

close() returns zero on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
: *fd* isn't a valid open file descriptor.

EINTR
: The **close**() call was interrupted by a signal; see *signal*(7).

EIO An I/O error occurred.

ENOSPC
EDQUOT
: On NFS, these errors are not normally reported against the first write which exceeds the available storage space, but instead against a subsequent *write*(2), *fsync*(2), or **close**().

See NOTES for a discussion of why **close**() should not be retried after an error.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use *fsync*(2). (It will depend on the disk hardware at this point.)

The close-on-exec file descriptor flag can be used to ensure that a file descriptor is automatically closed upon a successful *execve*(2); see *fcntl*(2) for details.

Multithreaded processes and close()

It is probably unwise to close file descriptors while they may be in use by system calls in other threads in the same process. Since a file descriptor may be reused, there are some obscure race conditions that may cause unintended side effects.

Furthermore, consider the following scenario where two threads are performing operations on the same file descriptor:

(1) One thread is blocked in an I/O system call on the file descriptor. For example, it is trying to *write*(2) to a pipe that is already full, or trying to *read*(2) from a stream socket which currently has no available data.

(2) Another thread closes the file descriptor.

The behavior in this situation varies across systems. On some systems, when the file descriptor is closed, the blocking system call returns immediately with an error.

On Linux (and possibly some other systems), the behavior is different: the blocking I/O system call holds a reference to the underlying open file description, and this reference keeps the description open until the I/O system call completes. (See *open*(2) for a discussion of open file descriptions.) Thus, the blocking system call in the first thread may successfully complete after the **close**() in the second thread.

Dealing with error returns from close()

A careful programmer will check the return value of **close**(), since it is quite possible that errors on a previous *write*(2) operation are reported only on the final **close**() that releases the open file description. Failing to check the return value when closing a file may lead to *silent* loss of data. This can especially be observed with NFS and with disk quota.

Note, however, that a failure return should be used only for diagnostic purposes (i.e., a warning to the application that there may still be I/O pending or there may have been failed I/O) or remedial purposes (e.g., writing the file once more or creating a backup).

Retrying the **close**() after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to be closed. This can occur because the Linux kernel *always* releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Many other implementations similarly always close the file descriptor (except in the case of **EBADF**, meaning that the file descriptor was invalid) even if they subsequently report an error on return from **close**(). POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

A careful programmer who wants to know about I/O errors may precede **close**() with a call to *fsync*(2).

The **EINTR** error is a somewhat special case. Regarding the **EINTR** error, POSIX.1-2008 says:

> If **close**() is interrupted by a signal that is to be caught, it shall return −1 with *errno* set to **EINTR** and the state of *fildes* is unspecified.

This permits the behavior that occurs on Linux and many other implementations, where, as with other errors that may be reported by **close**(), the file descriptor is guaranteed to be closed. However, it also permits another possibility: that the implementation returns an **EINTR** error and keeps the file descriptor open. (According to its documentation, HP-UX's **close**() does this.) The caller must then once more use **close**() to close the file descriptor, to avoid file descriptor leaks. This divergence in implementation behaviors provides a difficult hurdle for portable applications, since on many implementations, **close**() must not be called again after an **EINTR** error, and on at least one, **close**() must be called again. There are plans to address this conundrum for the next major release of the POSIX.1 standard.

SEE ALSO

close_range(2), *fcntl*(2), *fsync*(2), *open*(2), *shutdown*(2), *unlink*(2), *fclose*(3)

NAME

close_range − close all file descriptors in a given range

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>

#include <linux/close_range.h> /* Definition of CLOSE_RANGE_* constants */

int close_range(unsigned int first, unsigned int last, int flags);
```

DESCRIPTION

The **close_range**() system call closes all open file descriptors from *first* to *last* (included).

Errors closing a given file descriptor are currently ignored.

flags is a bit mask containing 0 or more of the following:

CLOSE_RANGE_CLOEXEC (since Linux 5.11)
: Set the close-on-exec flag on the specified file descriptors, rather than immediately closing them.

CLOSE_RANGE_UNSHARE
: Unshare the specified file descriptors from any other processes before closing them, avoiding races with other threads sharing the file descriptor table.

RETURN VALUE

On success, **close_range**() returns 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
: *flags* is not valid, or *first* is greater than *last*.

The following can occur with **CLOSE_RANGE_UNSHARE** (when constructing the new descriptor table):

EMFILE
: The number of open file descriptors exceeds the limit specified in */proc/sys/fs/nr_open* (see *proc*(5)). This error can occur in situations where that limit was lowered before a call to **close_range**() where the **CLOSE_RANGE_UNSHARE** flag is specified.

ENOMEM
: Insufficient kernel memory was available.

STANDARDS

None.

HISTORY

FreeBSD. Linux 5.9, glibc 2.34.

NOTES

Closing all open file descriptors

To avoid blindly closing file descriptors in the range of possible file descriptors, this is sometimes implemented (on Linux) by listing open file descriptors in */proc/self/fd/* and calling *close*(2) on each one. **close_range**() can take care of this without requiring */proc* and within a single system call, which provides significant performance benefits.

Closing file descriptors before exec

File descriptors can be closed safely using

```
/* we don't want anything past stderr here */
close_range(3, ~0U, CLOSE_RANGE_UNSHARE);
execve(....);
```

CLOSE_RANGE_UNSHARE is conceptually equivalent to

```
unshare(CLONE_FILES);
```

```
close_range(first, last, 0);
```

but can be more efficient: if the unshared range extends past the current maximum number of file descriptors allocated in the caller's file descriptor table (the common case when *last* is ~0U), the kernel will unshare a new file descriptor table for the caller up to *first*, copying as few file descriptors as possible. This avoids subsequent *close*(2) calls entirely; the whole operation is complete once the table is unshared.

Closing files on exec

This is particularly useful in cases where multiple pre-**exec** setup steps risk conflicting with each other. For example, setting up a *seccomp*(2) profile can conflict with a **close_range**() call: if the file descriptors are closed before the *seccomp*(2) profile is set up, the profile setup can't use them itself, or control their closure; if the file descriptors are closed afterwards, the seccomp profile can't block the **close_range**() call or any fallbacks. Using **CLOSE_RANGE_CLOEXEC** avoids this: the descriptors can be marked before the *seccomp*(2) profile is set up, and the profile can control access to **close_range**() without affecting the calling process.

EXAMPLES

The program shown below opens the files named in its command-line arguments, displays the list of files that it has opened (by iterating through the entries in */proc/PID/fd*), uses **close_range**() to close all file descriptors greater than or equal to 3, and then once more displays the process's list of open files. The following example demonstrates the use of the program:

```
$ touch /tmp/a /tmp/b /tmp/c
$ ./a.out /tmp/a /tmp/b /tmp/c
/tmp/a opened as FD 3
/tmp/b opened as FD 4
/tmp/c opened as FD 5
/proc/self/fd/0 ==> /dev/pts/1
/proc/self/fd/1 ==> /dev/pts/1
/proc/self/fd/2 ==> /dev/pts/1
/proc/self/fd/3 ==> /tmp/a
/proc/self/fd/4 ==> /tmp/b
/proc/self/fd/5 ==> /tmp/b
/proc/self/fd/6 ==> /proc/9005/fd
========= About to call close_range() =======
/proc/self/fd/0 ==> /dev/pts/1
/proc/self/fd/1 ==> /dev/pts/1
/proc/self/fd/2 ==> /dev/pts/1
/proc/self/fd/3 ==> /proc/9005/fd
```

Note that the lines showing the pathname */proc/9005/fd* result from the calls to *opendir*(3).

Program source

```
#define _GNU_SOURCE
#include <dirent.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* Show the contents of the symbolic links in /proc/self/fd */

static void
show_fds(void)
{
    DIR           *dirp;
    char          path[PATH_MAX], target[PATH_MAX];
    ssize_t       len;
    struct dirent  *dp;
```

```
        dirp = opendir("/proc/self/fd");
        if (dirp  == NULL) {
            perror("opendir");
            exit(EXIT_FAILURE);
        }

        for (;;) {
            dp = readdir(dirp);
            if (dp == NULL)
                break;

            if (dp->d_type == DT_LNK) {
                snprintf(path, sizeof(path), "/proc/self/fd/%s",
                         dp->d_name);

                len = readlink(path, target, sizeof(target));
                printf("%s ==> %.*s\n", path, (int) len, target);
            }
        }

        closedir(dirp);
    }

    int
    main(int argc, char *argv[])
    {
        int  fd;

        for (size_t j = 1; j < argc; j++) {
            fd = open(argv[j], O_RDONLY);
            if (fd == -1) {
                perror(argv[j]);
                exit(EXIT_FAILURE);
            }
            printf("%s opened as FD %d\n", argv[j], fd);
        }

        show_fds();

        printf("========= About to call close_range() =======\n");

        if (close_range(3, ~0U, 0) == -1) {
            perror("close_range");
            exit(EXIT_FAILURE);
        }

        show_fds();
        exit(EXIT_FAILURE);
    }
```

SEE ALSO

close(2)

NAME

connect − initiate a connection on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

DESCRIPTION

The **connect**() system call connects the socket referred to by the file descriptor *sockfd* to the address specified by *addr*. The *addrlen* argument specifies the size of *addr*. The format of the address in *addr* is determined by the address space of the socket *sockfd*; see *socket*(2) for further details.

If the socket *sockfd* is of type **SOCK_DGRAM**, then *addr* is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK_STREAM** or **SOCK_SEQPACKET**, this call attempts to make a connection to the socket that is bound to the address specified by *addr*.

Some protocol sockets (e.g., UNIX domain stream sockets) may successfully **connect**() only once.

Some protocol sockets (e.g., datagram sockets in the UNIX and Internet domains) may use **connect**() multiple times to change their association.

Some protocol sockets (e.g., TCP sockets as well as datagram sockets in the UNIX and Internet domains) may dissolve the association by connecting to an address with the *sa_family* member of *sockaddr* set to **AF_UNSPEC**; thereafter, the socket can be connected to another address. (**AF_UNSPEC** is supported since Linux 2.2.)

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The following are general socket errors only. There may be other domain-specific error codes.

EACCES
: For UNIX domain sockets, which are identified by pathname: Write permission is denied on the socket file, or search permission is denied for one of the directories in the path prefix. (See also *path_resolution*(7).)

EACCES
EPERM
: The user tried to connect to a broadcast address without having the socket broadcast flag enabled or the connection request failed because of a local firewall rule.

EACCES
: It can also be returned if an SELinux policy denied a connection (for example, if there is a policy saying that an HTTP proxy can only connect to ports associated with HTTP servers, and the proxy tries to connect to a different port).

EADDRINUSE
: Local address is already in use.

EADDRNOTAVAIL
: (Internet domain sockets) The socket referred to by *sockfd* had not previously been bound to an address and, upon attempting to bind it to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of */proc/sys/net/ipv4/ip_local_port_range* in *ip*(7).

EAFNOSUPPORT
: The passed address didn't have the correct address family in its *sa_family* field.

EAGAIN
: For nonblocking UNIX domain sockets, the socket is nonblocking, and the connection cannot be completed immediately. For other socket families, there are insufficient entries in the

routing cache.

EALREADY
The socket is nonblocking and a previous connection attempt has not yet been completed.

EBADF
sockfd is not a valid open file descriptor.

ECONNREFUSED
A **connect**() on a stream socket found no one listening on the remote address.

EFAULT
The socket structure address is outside the user's address space.

EINPROGRESS
The socket is nonblocking and the connection cannot be completed immediately. (UNIX domain sockets failed with **EAGAIN** instead.) It is possible to *select*(2) or *poll*(2) for completion by selecting the socket for writing. After *select*(2) indicates writability, use *getsockopt*(2) to read the **SO_ERROR** option at level **SOL_SOCKET** to determine whether **connect**() completed successfully (**SO_ERROR** is zero) or unsuccessfully (**SO_ERROR** is one of the usual error codes listed here, explaining the reason for the failure).

EINTR
The system call was interrupted by a signal that was caught; see *signal*(7).

EISCONN
The socket is already connected.

ENETUNREACH
Network is unreachable.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

EPROTOTYPE
The socket type does not support the requested communications protocol. This error can occur, for example, on an attempt to connect a UNIX domain datagram socket to a stream socket.

ETIMEDOUT
Timeout while attempting connection. The server may be too busy to accept new connections. Note that for IP sockets the timeout may be very long when syncookies are enabled on the server.

STANDARDS
POSIX.1-2008.

HISTORY
POSIX.1-2001, SVr4, 4.4BSD, (**connect**() first appeared in 4.2BSD).

NOTES
If **connect**() fails, consider the state of the socket as unspecified. Portable applications should close the socket and create a new one for reconnecting.

EXAMPLES
An example of the use of **connect**() is shown in *getaddrinfo*(3).

SEE ALSO
accept(2), *bind*(2), *getsockname*(2), *listen*(2), *socket*(2), *path_resolution*(7), *selinux*(8)

NAME

copy_file_range − Copy a range of data from one file to another

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <unistd.h>

ssize_t copy_file_range(int fd_in, off_t *_Nullable off_in,
                        int fd_out, off_t *_Nullable off_out,
                        size_t len, unsigned int flags);
```

DESCRIPTION

The **copy_file_range**() system call performs an in-kernel copy between two file descriptors without the additional cost of transferring data from the kernel to user space and then back into the kernel. It copies up to *len* bytes of data from the source file descriptor *fd_in* to the target file descriptor *fd_out*, overwriting any data that exists within the requested range of the target file.

The following semantics apply for *off_in*, and similar statements apply to *off_out*:

- If *off_in* is NULL, then bytes are read from *fd_in* starting from the file offset, and the file offset is adjusted by the number of bytes copied.
- If *off_in* is not NULL, then *off_in* must point to a buffer that specifies the starting offset where bytes from *fd_in* will be read. The file offset of *fd_in* is not changed, but *off_in* is adjusted appropriately.

fd_in and *fd_out* can refer to the same file. If they refer to the same file, then the source and target ranges are not allowed to overlap.

The *flags* argument is provided to allow for future extensions and currently must be set to 0.

RETURN VALUE

Upon successful completion, **copy_file_range**() will return the number of bytes copied between files. This could be less than the length originally requested. If the file offset of *fd_in* is at or past the end of file, no bytes are copied, and **copy_file_range**() returns zero.

On error, **copy_file_range**() returns −1 and *errno* is set to indicate the error.

ERRORS

EBADF
One or more file descriptors are not valid.

EBADF
fd_in is not open for reading; or *fd_out* is not open for writing.

EBADF
The **O_APPEND** flag is set for the open file description (see *open*(2)) referred to by the file descriptor *fd_out*.

EFBIG
An attempt was made to write at a position past the maximum file offset the kernel supports.

EFBIG
An attempt was made to write a range that exceeds the allowed maximum file size. The maximum file size differs between filesystem implementations and can be different from the maximum allowed file offset.

EFBIG
An attempt was made to write beyond the process's file size resource limit. This may also result in the process receiving a **SIGXFSZ** signal.

EINVAL
The *flags* argument is not 0.

EINVAL
fd_in and *fd_out* refer to the same file and the source and target ranges overlap.

EINVAL
Either *fd_in* or *fd_out* is not a regular file.

EIO A low-level I/O error occurred while copying.

EISDIR
Either *fd_in* or *fd_out* refers to a directory.

ENOMEM
Out of memory.

ENOSPC
There is not enough space on the target filesystem to complete the copy.

EOPNOTSUPP (since Linux 5.19)
The filesystem does not support this operation.

EOVERFLOW
The requested source or destination range is too large to represent in the specified data types.

EPERM
fd_out refers to an immutable file.

ETXTBSY
Either *fd_in* or *fd_out* refers to an active swap file.

EXDEV (before Linux 5.3)
The files referred to by *fd_in* and *fd_out* are not on the same filesystem.

EXDEV (since Linux 5.19)
The files referred to by *fd_in* and *fd_out* are not on the same filesystem, and the source and target filesystems are not of the same type, or do not support cross-filesystem copy.

VERSIONS

A major rework of the kernel implementation occurred in Linux 5.3. Areas of the API that weren't clearly defined were clarified and the API bounds are much more strictly checked than on earlier kernels.

Since Linux 5.19, cross-filesystem copies can be achieved when both filesystems are of the same type, and that filesystem implements support for it. See BUGS for behavior prior to Linux 5.19.

Applications should target the behaviour and requirements of Linux 5.19, that was also backported to earlier stable kernels.

STANDARDS

Linux, GNU.

HISTORY

Linux 4.5, but glibc 2.27 provides a user-space emulation when it is not available.

NOTES

If *fd_in* is a sparse file, then **copy_file_range**() may expand any holes existing in the requested range. Users may benefit from calling **copy_file_range**() in a loop, and using the *lseek*(2) **SEEK_DATA** and **SEEK_HOLE** operations to find the locations of data segments.

copy_file_range() gives filesystems an opportunity to implement "copy acceleration" techniques, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks) or server-side-copy (in the case of NFS).

_FILE_OFFSET_BITS should be defined to be 64 in code that uses non-null *off_in* or *off_out* or that takes the address of **copy_file_range**, if the code is intended to be portable to traditional 32-bit x86 and ARM platforms where **off_t**'s width defaults to 32 bits.

BUGS

In Linux 5.3 to Linux 5.18, cross-filesystem copies were implemented by the kernel, if the operation was not supported by individual filesystems. However, on some virtual filesystems, the call failed to copy, while still reporting success.

EXAMPLES

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int          fd_in, fd_out;
    off_t        len, ret;
    struct stat  stat;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd_in = open(argv[1], O_RDONLY);
    if (fd_in == -1) {
        perror("open (argv[1])");
        exit(EXIT_FAILURE);
    }

    if (fstat(fd_in, &stat) == -1) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }

    len = stat.st_size;

    fd_out = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd_out == -1) {
        perror("open (argv[2])");
        exit(EXIT_FAILURE);
    }

    do {
        ret = copy_file_range(fd_in, NULL, fd_out, NULL, len, 0);
        if (ret == -1) {
            perror("copy_file_range");
            exit(EXIT_FAILURE);
        }

        len -= ret;
    } while (len > 0 && ret > 0);

    close(fd_in);
    close(fd_out);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

lseek(2), *sendfile*(2), *splice*(2)

NAME

create_module − create a loadable module entry

SYNOPSIS

#include <linux/module.h>

[[deprecated]] caddr_t create_module(const char ****name*, size_t** *size***);**

DESCRIPTION

Note: This system call is present only before Linux 2.6.

create_module() attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module. This system call requires privilege.

RETURN VALUE

On success, returns the kernel address at which the module will reside. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EEXIST
A module by that name already exists.

EFAULT
name is outside the program's accessible address space.

EINVAL
The requested size is too small even for the module header information.

ENOMEM
The kernel could not allocate a contiguous block of memory large enough for the module.

ENOSYS
create_module() is not supported in this version of the kernel (e.g., Linux 2.6 or later).

EPERM
The caller was not privileged (did not have the **CAP_SYS_MODULE** capability).

STANDARDS

Linux.

HISTORY

Removed in Linux 2.6.

This obsolete system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc versions before glibc 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it was sufficient to manually declare the interface in your code; alternatively, you could invoke the system call using *syscall*(2).

SEE ALSO

delete_module(2), *init_module*(2), *query_module*(2)

NAME

delete_module − unload a kernel module

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>          /* Definition of O_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_delete_module, const char *name, unsigned int flags);
```

Note: glibc provides no wrapper for **delete_module**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **delete_module**() system call attempts to remove the unused loadable module entry identified by *name*. If the module has an *exit* function, then that function is executed before unloading the module. The *flags* argument is used to modify the behavior of the system call, as described below. This system call requires privilege.

Module removal is attempted according to the following rules:

(1) If there are other loaded modules that depend on (i.e., refer to symbols defined in) this module, then the call fails.

(2) Otherwise, if the reference count for the module (i.e., the number of processes currently using the module) is zero, then the module is immediately unloaded.

(3) If a module has a nonzero reference count, then the behavior depends on the bits set in *flags*. In normal usage (see NOTES), the **O_NONBLOCK** flag is always specified, and the **O_TRUNC** flag may additionally be specified.

The various combinations for *flags* have the following effect:

flags == O_NONBLOCK
The call returns immediately, with an error.

flags == (O_NONBLOCK | O_TRUNC)
The module is unloaded immediately, regardless of whether it has a nonzero reference count.

(flags & O_NONBLOCK) == 0
If *flags* does not specify **O_NONBLOCK**, the following steps occur:

- The module is marked so that no new references are permitted.
- If the module's reference count is nonzero, the caller is placed in an uninterruptible sleep state (**TASK_UNINTERRUPTIBLE**) until the reference count is zero, at which point the call unblocks.
- The module is unloaded in the usual way.

The **O_TRUNC** flag has one further effect on the rules described above. By default, if a module has an *init* function but no *exit* function, then an attempt to remove the module fails. However, if **O_TRUNC** was specified, this requirement is bypassed.

Using the **O_TRUNC** flag is dangerous! If the kernel was not built with **CONFIG_MODULE_FORCE_UNLOAD**, this flag is silently ignored. (Normally, **CONFIG_MODULE_FORCE_UNLOAD** is enabled.) Using this flag taints the kernel (TAINT_FORCED_RMMOD).

RETURN VALUE

On success, zero is returned. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBUSY
The module is not "live" (i.e., it is still being initialized or is already marked for removal); or, the module has an *init* function but has no *exit* function, and **O_TRUNC** was not specified in *flags*.

EFAULT
name refers to a location outside the process's accessible address space.

ENOENT
No module by that name exists.

EPERM
The caller was not privileged (did not have the **CAP_SYS_MODULE** capability), or module unloading is disabled (see */proc/sys/kernel/modules_disabled* in *proc*(5)).

EWOULDBLOCK
Other modules depend on this module; or, **O_NONBLOCK** was specified in *flags*, but the reference count of this module is nonzero and **O_TRUNC** was not specified in *flags*.

STANDARDS
Linux.

HISTORY
The **delete_module**() system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc versions before glibc 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it is (before glibc 2.23) sufficient to manually declare the interface in your code; alternatively, you can invoke the system call using *syscall*(2).

Linux 2.4 and earlier
In Linux 2.4 and earlier, the system call took only one argument:

int delete_module(const char *_name_);

If *name* is NULL, all unused modules marked auto-clean are removed.

Some further details of differences in the behavior of **delete_module**() in Linux 2.4 and earlier are *not* currently explained in this manual page.

NOTES
The uninterruptible sleep that may occur if **O_NONBLOCK** is omitted from *flags* is considered undesirable, because the sleeping process is left in an unkillable state. As at Linux 3.7, specifying **O_NONBLOCK** is optional, but in future kernels it is likely to become mandatory.

SEE ALSO
create_module(2), *init_module*(2), *query_module*(2), *lsmod*(8), *modprobe*(8), *rmmod*(8)

NAME

dup, dup2, dup3 − duplicate a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>          /* Definition of O_* constants */
#include <unistd.h>

int dup3(int oldfd, int newfd, int  flags);
```

DESCRIPTION

The **dup**() system call allocates a new file descriptor that refers to the same open file description as the descriptor *oldfd*. (For an explanation of open file descriptions, see *open*(2).) The new file descriptor number is guaranteed to be the lowest-numbered file descriptor that was unused in the calling process.

After a successful return, the old and new file descriptors may be used interchangeably. Since the two file descriptors refer to the same open file description, they share file offset and file status flags; for example, if the file offset is modified by using *lseek*(2) on one of the file descriptors, the offset is also changed for the other file descriptor.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (**FD_CLOEXEC**; see *fcntl*(2)) for the duplicate descriptor is off.

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*. In other words, the file descriptor *newfd* is adjusted so that it now refers to the same open file description as *oldfd*.

If the file descriptor *newfd* was previously open, it is closed before being reused; the close is performed silently (i.e., any errors during the close are not reported by *dup2*())

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*. This is important, because trying to implement equivalent functionality using *close*(2) and **dup**() would be subject to race conditions, whereby *newfd* might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

- If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
- If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.

dup3()

dup3() is the same as **dup2**(), except that:

- The caller can force the close-on-exec flag to be set for the new file descriptor by specifying **O_CLOEXEC** in *flags*. See the description of the same flag in *open*(2) for reasons why this may be useful.
- If *oldfd* equals *newfd*, then **dup3**() fails with the error **EINVAL**.

RETURN VALUE

On success, these system calls return the new file descriptor. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF

oldfd isn't an open file descriptor.

EBADF
newfd is out of the allowed range for file descriptors (see the discussion of **RLIMIT_NOFILE** in *getrlimit*(2)).

EBUSY
(Linux only) This may be returned by **dup2**() or **dup3**() during a race condition with *open*(2) and **dup**().

EINTR
The **dup2**() or **dup3**() call was interrupted by a signal; see *signal*(7).

EINVAL
(**dup3**()) *flags* contain an invalid value.

EINVAL
(**dup3**()) *oldfd* was equal to *newfd*.

EMFILE
The per-process limit on the number of open file descriptors has been reached (see the discussion of **RLIMIT_NOFILE** in *getrlimit*(2)).

STANDARDS

dup()
dup2() POSIX.1-2008.

dup3() Linux.

HISTORY

dup()
dup2() POSIX.1-2001, SVr4, 4.3BSD.

dup3() Linux 2.6.27, glibc 2.9.

NOTES

The error returned by **dup2**() is different from that returned by **fcntl(**..., **F_DUPFD**, ...**)** when *newfd* is out of range. On some systems, **dup2**() also sometimes returns **EINVAL** like **F_DUPFD**.

If *newfd* was open, any errors that would have been reported at *close*(2) time are lost. If this is of concern, then—unless the program is single-threaded and does not allocate file descriptors in signal handlers—the correct approach is *not* to close *newfd* before calling **dup2**(), because of the race condition described above. Instead, code something like the following could be used:

```
/* Obtain a duplicate of 'newfd' that can subsequently
   be used to check for close() errors; an EBADF error
   means that 'newfd' was not open. */

tmpfd = dup(newfd);
if (tmpfd == -1 && errno != EBADF) {
    /* Handle unexpected dup() error. */
}

/* Atomically duplicate 'oldfd' on 'newfd'. */

if (dup2(oldfd, newfd) == -1) {
    /* Handle dup2() error. */
}

/* Now check for close() errors on the file originally
   referred to by 'newfd'. */

if (tmpfd != -1) {
    if (close(tmpfd) == -1) {
        /* Handle errors from close. */
    }
}
```

SEE ALSO

close(2), *fcntl*(2), *open*(2), *pidfd_getfd*(2)

NAME

epoll_create, epoll_create1 − open an epoll file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/epoll.h>

int epoll_create(int *size***);**
int epoll_create1(int *flags***);**

DESCRIPTION

epoll_create() creates a new *epoll*(7) instance. Since Linux 2.6.8, the *size* argument is ignored, but must be greater than zero; see HISTORY.

epoll_create() returns a file descriptor referring to the new epoll instance. This file descriptor is used for all the subsequent calls to the **epoll** interface. When no longer required, the file descriptor returned by **epoll_create**() should be closed by using *close*(2). When all file descriptors referring to an epoll instance have been closed, the kernel destroys the instance and releases the associated resources for reuse.

epoll_create1()

If *flags* is 0, then, other than the fact that the obsolete *size* argument is dropped, **epoll_create1**() is the same as **epoll_create**(). The following value can be included in *flags* to obtain different behavior:

EPOLL_CLOEXEC
Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

RETURN VALUE

On success, these system calls return a file descriptor (a nonnegative integer). On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
size is not positive.

EINVAL
(**epoll_create1**()) Invalid value specified in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
There was insufficient memory to create the kernel object.

STANDARDS

Linux.

HISTORY

epoll_create()
Linux 2.6, glibc 2.3.2.

epoll_create1()
Linux 2.6.27, glibc 2.9.

In the initial **epoll_create**() implementation, the *size* argument informed the kernel of the number of file descriptors that the caller expected to add to the **epoll** instance. The kernel used this information as a hint for the amount of space to initially allocate in internal data structures describing events. (If necessary, the kernel would allocate more space if the caller's usage exceeded the hint given in *size*.) Nowadays, this hint is no longer required (the kernel dynamically sizes the required data structures without needing the hint), but *size* must still be greater than zero, in order to ensure backward compatibility when new **epoll** applications are run on older kernels.

Prior to Linux 2.6.29, a */proc/sys/fs/epoll/max_user_instances* kernel parameter limited live epolls for

each real user ID, and caused **epoll_create**() to fail with **EMFILE** on overrun.

SEE ALSO

close(2), *epoll_ctl*(2), *epoll_wait*(2), *epoll*(7)

NAME

epoll_ctl − control interface for an epoll file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *_Nullable event);
```

DESCRIPTION

This system call is used to add, modify, or remove entries in the interest list of the *epoll*(7) instance referred to by the file descriptor *epfd*. It requests that the operation *op* be performed for the target file descriptor, *fd*.

Valid values for the *op* argument are:

EPOLL_CTL_ADD
: Add an entry to the interest list of the epoll file descriptor, *epfd*. The entry includes the file descriptor, *fd*, a reference to the corresponding open file description (see *epoll*(7) and *open*(2)), and the settings specified in *event*.

EPOLL_CTL_MOD
: Change the settings associated with *fd* in the interest list to the new settings specified in *event*.

EPOLL_CTL_DEL
: Remove (deregister) the target file descriptor *fd* from the interest list. The *event* argument is ignored and can be NULL (but see BUGS below).

The *event* argument describes the object linked to the file descriptor *fd*. The *struct epoll_event* is described in *epoll_event*(3type).

The *data* member of the *epoll_event* structure specifies data that the kernel should save and then return (via *epoll_wait*(2)) when this file descriptor becomes ready.

The *events* member of the *epoll_event* structure is a bit mask composed by ORing together zero or more event types, returned by *epoll_wait*(2), and input flags, which affect its behaviour, but aren't returned. The available event types are:

EPOLLIN
: The associated file is available for *read*(2) operations.

EPOLLOUT
: The associated file is available for *write*(2) operations.

EPOLLRDHUP (since Linux 2.6.17)
: Stream socket peer closed connection, or shut down writing half of connection. (This flag is especially useful for writing simple code to detect peer shutdown when using edge-triggered monitoring.)

EPOLLPRI
: There is an exceptional condition on the file descriptor. See the discussion of **POLLPRI** in *poll*(2).

EPOLLERR
: Error condition happened on the associated file descriptor. This event is also reported for the write end of a pipe when the read end has been closed.

  epoll_wait(2) will always report for this event; it is not necessary to set it in *events* when calling **epoll_ctl**().

EPOLLHUP
: Hang up happened on the associated file descriptor.

  epoll_wait(2) will always wait for this event; it is not necessary to set it in *events* when calling **epoll_ctl**().

  Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer closed its end of the channel. Subsequent reads from the channel will

return 0 (end of file) only after all outstanding data in the channel has been consumed.

And the available input flags are:

EPOLLET
Requests edge-triggered notification for the associated file descriptor. The default behavior for **epoll** is level-triggered. See *epoll*(7) for more detailed information about edge-triggered and level-triggered notification.

EPOLLONESHOT (since Linux 2.6.2)
Requests one-shot notification for the associated file descriptor. This means that after an event notified for the file descriptor by *epoll_wait*(2), the file descriptor is disabled in the interest list and no other events will be reported by the **epoll** interface. The user must call **epoll_ctl**() with **EPOLL_CTL_MOD** to rearm the file descriptor with a new event mask.

EPOLLWAKEUP (since Linux 3.5)
If **EPOLLONESHOT** and **EPOLLET** are clear and the process has the **CAP_BLOCK_SUSPEND** capability, ensure that the system does not enter "suspend" or "hibernate" while this event is pending or being processed. The event is considered as being "processed" from the time when it is returned by a call to *epoll_wait*(2) until the next call to *epoll_wait*(2) on the same *epoll*(7) file descriptor, the closure of that file descriptor, the removal of the event file descriptor with **EPOLL_CTL_DEL**, or the clearing of **EPOLLWAKEUP** for the event file descriptor with **EPOLL_CTL_MOD**. See also BUGS.

EPOLLEXCLUSIVE (since Linux 4.5)
Sets an exclusive wakeup mode for the epoll file descriptor that is being attached to the target file descriptor, *fd*. When a wakeup event occurs and multiple epoll file descriptors are attached to the same target file using **EPOLLEXCLUSIVE**, one or more of the epoll file descriptors will receive an event with *epoll_wait*(2). The default in this scenario (when **EPOLLEXCLUSIVE** is not set) is for all epoll file descriptors to receive an event. **EPOLLEXCLUSIVE** is thus useful for avoiding thundering herd problems in certain scenarios.

If the same file descriptor is in multiple epoll instances, some with the **EPOLLEXCLUSIVE** flag, and others without, then events will be provided to all epoll instances that did not specify **EPOLLEXCLUSIVE**, and at least one of the epoll instances that did specify **EPOLLEXCLUSIVE**.

The following values may be specified in conjunction with **EPOLLEXCLUSIVE**: **EPOLLIN**, **EPOLLOUT**, **EPOLLWAKEUP**, and **EPOLLET**. **EPOLLHUP** and **EPOLLERR** can also be specified, but this is not required: as usual, these events are always reported if they occur, regardless of whether they are specified in *events*. Attempts to specify other values in *events* yield the error **EINVAL**.

EPOLLEXCLUSIVE may be used only in an **EPOLL_CTL_ADD** operation; attempts to employ it with **EPOLL_CTL_MOD** yield an error. If **EPOLLEXCLUSIVE** has been set using **epoll_ctl**(), then a subsequent **EPOLL_CTL_MOD** on the same *epfd*, *fd* pair yields an error. A call to **epoll_ctl**() that specifies **EPOLLEXCLUSIVE** in *events* and specifies the target file descriptor *fd* as an epoll instance will likewise fail. The error in all of these cases is **EINVAL**.

RETURN VALUE
When successful, **epoll_ctl**() returns zero. When an error occurs, **epoll_ctl**() returns −1 and *errno* is set to indicate the error.

ERRORS
EBADF
epfd or *fd* is not a valid file descriptor.

EEXIST
op was **EPOLL_CTL_ADD**, and the supplied file descriptor *fd* is already registered with this epoll instance.

EINVAL
epfd is not an **epoll** file descriptor, or *fd* is the same as *epfd*, or the requested operation *op* is not supported by this interface.

EINVAL
An invalid event type was specified along with **EPOLLEXCLUSIVE** in *events*.

EINVAL
op was **EPOLL_CTL_MOD** and *events* included **EPOLLEXCLUSIVE**.

EINVAL
op was **EPOLL_CTL_MOD** and the **EPOLLEXCLUSIVE** flag has previously been applied to this *epfd*, *fd* pair.

EINVAL
EPOLLEXCLUSIVE was specified in *event* and *fd* refers to an epoll instance.

ELOOP
fd refers to an epoll instance and this **EPOLL_CTL_ADD** operation would result in a circular loop of epoll instances monitoring one another or a nesting depth of epoll instances greater than 5.

ENOENT
op was **EPOLL_CTL_MOD** or **EPOLL_CTL_DEL**, and *fd* is not registered with this epoll instance.

ENOMEM
There was insufficient memory to handle the requested *op* control operation.

ENOSPC
The limit imposed by */proc/sys/fs/epoll/max_user_watches* was encountered while trying to register (**EPOLL_CTL_ADD**) a new file descriptor on an epoll instance. See *epoll*(7) for further details.

EPERM
The target file *fd* does not support **epoll**. This error can occur if *fd* refers to, for example, a regular file or a directory.

STANDARDS
Linux.

HISTORY
Linux 2.6, glibc 2.3.2.

NOTES
The **epoll** interface supports all file descriptors that support *poll*(2).

BUGS
Before Linux 2.6.9, the **EPOLL_CTL_DEL** operation required a non-null pointer in *event*, even though this argument is ignored. Since Linux 2.6.9, *event* can be specified as NULL when using **EPOLL_CTL_DEL**. Applications that need to be portable to kernels before Linux 2.6.9 should specify a non-null pointer in *event*.

If **EPOLLWAKEUP** is specified in *flags*, but the caller does not have the **CAP_BLOCK_SUSPEND** capability, then the **EPOLLWAKEUP** flag is *silently ignored*. This unfortunate behavior is necessary because no validity checks were performed on the *flags* argument in the original implementation, and the addition of the **EPOLLWAKEUP** with a check that caused the call to fail if the caller did not have the **CAP_BLOCK_SUSPEND** capability caused a breakage in at least one existing user-space application that happened to randomly (and uselessly) specify this bit. A robust application should therefore double check that it has the **CAP_BLOCK_SUSPEND** capability if attempting to use the **EPOLLWAKEUP** flag.

SEE ALSO
epoll_create(2), *epoll_wait*(2), *poll*(2), *epoll*(7)

NAME

epoll_wait, epoll_pwait, epoll_pwait2 − wait for an I/O event on an epoll file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
int epoll_pwait(int epfd, struct epoll_event *events,
               int maxevents, int timeout,
               const sigset_t *_Nullable sigmask);
int epoll_pwait2(int epfd, struct epoll_event *events,
               int maxevents, const struct timespec *_Nullable timeout,
               const sigset_t *_Nullable sigmask);
```

DESCRIPTION

The **epoll_wait**() system call waits for events on the *epoll*(7) instance referred to by the file descriptor *epfd*. The buffer pointed to by *events* is used to return information from the ready list about file descriptors in the interest list that have some events available. Up to *maxevents* are returned by **epoll_wait**(). The *maxevents* argument must be greater than zero.

The *timeout* argument specifies the number of milliseconds that **epoll_wait**() will block. Time is measured against the **CLOCK_MONOTONIC** clock.

A call to **epoll_wait**() will block until either:

- a file descriptor delivers an event;
- the call is interrupted by a signal handler; or
- the timeout expires.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. Specifying a *timeout* of −1 causes **epoll_wait**() to block indefinitely, while specifying a *timeout* equal to zero causes **epoll_wait**() to return immediately, even if no events are available.

The *struct epoll_event* is described in *epoll_event*(3type).

The *data* field of each returned *epoll_event* structure contains the same data as was specified in the most recent call to *epoll_ctl*(2) (**EPOLL_CTL_ADD**, **EPOLL_CTL_MOD**) for the corresponding open file descriptor.

The *events* field is a bit mask that indicates the events that have occurred for the corresponding open file description. See *epoll_ctl*(2) for a list of the bits that may appear in this mask.

epoll_pwait()

The relationship between **epoll_wait**() and **epoll_pwait**() is analogous to the relationship between *select*(2) and *pselect*(2): like *pselect*(2), **epoll_pwait**() allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

The following **epoll_pwait**() call:

```
ready = epoll_pwait(epfd, &events, maxevents, timeout, &sigmask);
```

is equivalent to *atomically* executing the following calls:

```
sigset_t origmask;

pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = epoll_wait(epfd, &events, maxevents, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The *sigmask* argument may be specified as NULL, in which case **epoll_pwait**() is equivalent to **epoll_wait**().

epoll_pwait2()
The **epoll_pwait2**() system call is equivalent to **epoll_pwait**() except for the *timeout* argument. It takes an argument of type *timespec* to be able to specify nanosecond resolution timeout. This argument functions the same as in *pselect*(2) and *ppoll*(2). If *timeout* is NULL, then **epoll_pwait2**() can block indefinitely.

RETURN VALUE
On success, **epoll_wait**() returns the number of file descriptors ready for the requested I/O operation, or zero if no file descriptor became ready during the requested *timeout* milliseconds. On failure, **epoll_wait**() returns −1 and *errno* is set to indicate the error.

ERRORS
EBADF
epfd is not a valid file descriptor.

EFAULT
The memory area pointed to by *events* is not accessible with write permissions.

EINTR
The call was interrupted by a signal handler before either (1) any of the requested events occurred or (2) the *timeout* expired; see *signal*(7).

EINVAL
epfd is not an **epoll** file descriptor, or *maxevents* is less than or equal to zero.

STANDARDS
Linux.

HISTORY
epoll_wait()
Linux 2.6, glibc 2.3.2.

epoll_pwait()
Linux 2.6.19, glibc 2.6.

epoll_pwait2()
Linux 5.11.

NOTES
While one thread is blocked in a call to **epoll_wait**(), it is possible for another thread to add a file descriptor to the waited-upon **epoll** instance. If the new file descriptor becomes ready, it will cause the **epoll_wait**() call to unblock.

If more than *maxevents* file descriptors are ready when **epoll_wait**() is called, then successive **epoll_wait**() calls will round robin through the set of ready file descriptors. This behavior helps avoid starvation scenarios, where a process fails to notice that additional file descriptors are ready because it focuses on a set of file descriptors that are already known to be ready.

Note that it is possible to call **epoll_wait**() on an **epoll** instance whose interest list is currently empty (or whose interest list becomes empty because file descriptors are closed or removed from the interest in another thread). The call will block until some file descriptor is later added to the interest list (in another thread) and that file descriptor becomes ready.

C library/kernel differences
The raw **epoll_pwait**() and **epoll_pwait2**() system calls have a sixth argument, *size_t sigsetsize*, which specifies the size in bytes of the *sigmask* argument. The glibc **epoll_pwait**() wrapper function specifies this argument as a fixed value (equal to *sizeof(sigset_t)*).

BUGS
Before Linux 2.6.37, a *timeout* value larger than approximately *LONG_MAX / HZ* milliseconds is treated as −1 (i.e., infinity). Thus, for example, on a system where *sizeof(long)* is 4 and the kernel *HZ* value is 1000, this means that timeouts greater than 35.79 minutes are treated as infinity.

SEE ALSO
epoll_create(2), *epoll_ctl*(2), *epoll*(7)

NAME

eventfd − create a file descriptor for event notification

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/eventfd.h>

int eventfd(unsigned int *initval***, int** *flags***);**

DESCRIPTION

eventfd() creates an "eventfd object" that can be used as an event wait/notify mechanism by user-space applications, and by the kernel to notify user-space applications of events. The object contains an unsigned 64-bit integer (*uint64_t*) counter that is maintained by the kernel. This counter is initialized with the value specified in the argument *initval*.

As its return value, **eventfd**() returns a new file descriptor that can be used to refer to the eventfd object.

The following values may be bitwise ORed in *flags* to change the behavior of **eventfd**():

EFD_CLOEXEC (since Linux 2.6.27)
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

EFD_NONBLOCK (since Linux 2.6.27)
: Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

EFD_SEMAPHORE (since Linux 2.6.30)
: Provide semaphore-like semantics for reads from the new file descriptor. See below.

Up to Linux 2.6.26, the *flags* argument is unused, and must be specified as zero.

The following operations can be performed on the file descriptor returned by **eventfd**():

read(2) Each successful *read*(2) returns an 8-byte integer. A *read*(2) fails with the error **EINVAL** if the size of the supplied buffer is less than 8 bytes.

The value returned by *read*(2) is in host byte order—that is, the native byte order for integers on the host machine.

The semantics of *read*(2) depend on whether the eventfd counter currently has a nonzero value and whether the **EFD_SEMAPHORE** flag was specified when creating the eventfd file descriptor:

- If **EFD_SEMAPHORE** was not specified and the eventfd counter has a nonzero value, then a *read*(2) returns 8 bytes containing that value, and the counter's value is reset to zero.
- If **EFD_SEMAPHORE** was specified and the eventfd counter has a nonzero value, then a *read*(2) returns 8 bytes containing the value 1, and the counter's value is decremented by 1.
- If the eventfd counter is zero at the time of the call to *read*(2), then the call either blocks until the counter becomes nonzero (at which time, the *read*(2) proceeds as described above) or fails with the error **EAGAIN** if the file descriptor has been made nonblocking.

write(2)
: A *write*(2) call adds the 8-byte integer value supplied in its buffer to the counter. The maximum value that may be stored in the counter is the largest unsigned 64-bit value minus 1 (i.e., 0xfffffffffffffffe). If the addition would cause the counter's value to exceed the maximum, then the *write*(2) either blocks until a *read*(2) is performed on the file descriptor, or fails with the error **EAGAIN** if the file descriptor has been made nonblocking.

A *write*(2) fails with the error **EINVAL** if the size of the supplied buffer is less than 8 bytes, or if an attempt is made to write the value 0xffffffffffffffff.

poll(2)

select(2)
(and similar)
The returned file descriptor supports *poll*(2) (and analogously *epoll*(7)) and *select*(2), as follows:

- The file descriptor is readable (the *select*(2) *readfds* argument; the *poll*(2) **POLLIN** flag) if the counter has a value greater than 0.
- The file descriptor is writable (the *select*(2) *writefds* argument; the *poll*(2) **POLLOUT** flag) if it is possible to write a value of at least "1" without blocking.
- If an overflow of the counter value was detected, then *select*(2) indicates the file descriptor as being both readable and writable, and *poll*(2) returns a **POLLERR** event. As noted above, *write*(2) can never overflow the counter. However an overflow can occur if 2^64 eventfd "signal posts" were performed by the KAIO subsystem (theoretically possible, but practically unlikely). If an overflow has occurred, then *read*(2) will return that maximum *uint64_t* value (i.e., 0xffffffffffffffff).

The eventfd file descriptor also supports the other file-descriptor multiplexing APIs: *pselect*(2) and *ppoll*(2).

close(2)
When the file descriptor is no longer required it should be closed. When all file descriptors associated with the same eventfd object have been closed, the resources for object are freed by the kernel.

A copy of the file descriptor created by **eventfd**() is inherited by the child produced by *fork*(2). The duplicate file descriptor is associated with the same eventfd object. File descriptors created by **eventfd**() are preserved across *execve*(2), unless the close-on-exec flag has been set.

RETURN VALUE

On success, **eventfd**() returns a new eventfd file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
An unsupported value was specified in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
Could not mount (internal) anonymous inode device.

ENOMEM
There was insufficient memory to create a new eventfd file descriptor.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
eventfd()	Thread safety	MT-Safe

VERSIONS

C library/kernel differences

There are two underlying Linux system calls: **eventfd**() and the more recent **eventfd2**(). The former system call does not implement a *flags* argument. The latter system call implements the *flags* values described above. The glibc wrapper function will use **eventfd2**() where it is available.

Additional glibc features

The GNU C library defines an additional type, and two functions that attempt to abstract some of the details of reading and writing on an eventfd file descriptor:

```
typedef uint64_t eventfd_t;

int eventfd_read(int fd, eventfd_t *value);
```

```
int eventfd_write(int fd, eventfd_t value);
```

The functions perform the read and write operations on an eventfd file descriptor, returning 0 if the correct number of bytes was transferred, or −1 otherwise.

STANDARDS

Linux, GNU.

HISTORY

eventfd()
: Linux 2.6.22, glibc 2.8.

eventfd2()
: Linux 2.6.27 (see VERSIONS). Since glibc 2.9, the **eventfd**() wrapper will employ the **eventfd2**() system call, if it is supported by the kernel.

NOTES

Applications can use an eventfd file descriptor instead of a pipe (see *pipe*(2)) in all cases where a pipe is used simply to signal events. The kernel overhead of an eventfd file descriptor is much lower than that of a pipe, and only one file descriptor is required (versus the two required for a pipe).

When used in the kernel, an eventfd file descriptor can provide a bridge from kernel to user space, allowing, for example, functionalities like KAIO (kernel AIO) to signal to a file descriptor that some operation is complete.

A key point about an eventfd file descriptor is that it can be monitored just like any other file descriptor using *select*(2), *poll*(2), or *epoll*(7). This means that an application can simultaneously monitor the readiness of "traditional" files and the readiness of other kernel mechanisms that support the eventfd interface. (Without the **eventfd**() interface, these mechanisms could not be multiplexed via *select*(2), *poll*(2), or *epoll*(7).)

The current value of an eventfd counter can be viewed via the entry for the corresponding file descriptor in the process's */proc/*pid*/fdinfo* directory. See *proc*(5) for further details.

EXAMPLES

The following program creates an eventfd file descriptor and then forks to create a child process. While the parent briefly sleeps, the child writes each of the integers supplied in the program's command-line arguments to the eventfd file descriptor. When the parent has finished sleeping, it reads from the eventfd file descriptor.

The following shell session shows a sample run of the program:

```
$ ./a.out 1 2 4 7 14
Child writing 1 to efd
Child writing 2 to efd
Child writing 4 to efd
Child writing 7 to efd
Child writing 14 to efd
Child completed write loop
Parent about to read
Parent read 28 (0x1c) from efd
```

Program source

```
#include <err.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/eventfd.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int       efd;
    uint64_t  u;
```

```
        ssize_t   s;

        if (argc < 2) {
            fprintf(stderr, "Usage: %s <num>...\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        efd = eventfd(0, 0);
        if (efd == -1)
            err(EXIT_FAILURE, "eventfd");

        switch (fork()) {
        case 0:
            for (size_t j = 1; j < argc; j++) {
                printf("Child writing %s to efd\n", argv[j]);
                u = strtoull(argv[j], NULL, 0);
                        /* strtoull() allows various bases */
                s = write(efd, &u, sizeof(uint64_t));
                if (s != sizeof(uint64_t))
                    err(EXIT_FAILURE, "write");
            }
            printf("Child completed write loop\n");

            exit(EXIT_SUCCESS);

        default:
            sleep(2);

            printf("Parent about to read\n");
            s = read(efd, &u, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
                err(EXIT_FAILURE, "read");
            printf("Parent read %"PRIu64" (%#"PRIx64") from efd\n", u, u);
            exit(EXIT_SUCCESS);

        case -1:
            err(EXIT_FAILURE, "fork");
        }
    }
```

SEE ALSO

futex(2), *pipe*(2), *poll*(2), *read*(2), *select*(2), *signalfd*(2), *timerfd_create*(2), *write*(2), *epoll*(7), *sem_overview*(7)

NAME

execve − execute program

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int execve(const char ****pathname*, char *const _Nullable** *argv***[],**
char *const _Nullable *envp***[]);**

DESCRIPTION

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a line of the form:

```
#!interpreter [optional-arg]
```

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., *argv[0]*) should contain the filename associated with the file being executed. The *argv* array must be terminated by a null pointer. (Thus, in the new program, *argv[argc]* will be a null pointer.)

envp is an array of pointers to strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The *envp* array must be terminated by a null pointer.

This manual page describes the Linux system call in detail; for an overview of the nomenclature and the many, often preferable, standardised variants of this function provided by libc, including ones that search the **PATH** environment variable, see *exec*(3).

The argument vector and environment can be accessed by the new program's main function, when it is defined as:

```
int main(int argc, char *argv[], char *envp[])
```

Note, however, that the use of a third argument to the main function is not specified in POSIX.1; according to POSIX.1, the environment should be accessed via the external variable *environ*(7).

execve() does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

If the current program is being ptraced, a **SIGTRAP** signal is sent to it after a successful **execve**().

If the set-user-ID bit is set on the program file referred to by *pathname*, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, if the set-group-ID bit is set on the program file, then the effective group ID of the calling process is set to the group of the program file.

The aforementioned transformations of the effective IDs are *not* performed (i.e., the set-user-ID and set-group-ID bits are ignored) if any of the following is true:

- the *no_new_privs* attribute is set for the calling thread (see *prctl*(2));
- the underlying filesystem is mounted *nosuid* (the **MS_NOSUID** flag for *mount*(2)); or
- the calling process is being ptraced.

The capabilities of the program file (see *capabilities*(7)) are also ignored if any of the above are true.

The effective user ID of the process is copied to the saved set-user-ID; similarly, the effective group ID is copied to the saved set-group-ID. This copying takes place after any effective ID changes that occur because of the set-user-ID and set-group-ID mode bits.

The process's real UID and real GID, as well as its supplementary group IDs, are unchanged by a call to **execve**().

If the executable is an a.out dynamically linked binary executable containing shared-library stubs, the Linux dynamic linker *ld.so*(8) is called at the start of execution to bring needed shared objects into

memory and link the executable with them.

If the executable is a dynamically linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared objects. This interpreter is typically */lib/ld−linux.so.2* for binaries linked with glibc (see *ld-linux.so*(8)).

Effect on process attributes

All process attributes are preserved during an **execve**(), except the following:

- The dispositions of any signals that are being caught are reset to the default (**signal**(7)).
- Any alternate signal stack is not preserved (**sigaltstack**(2)).
- Memory mappings are not preserved (**mmap**(2)).
- Attached System V shared memory segments are detached (**shmat**(2)).
- POSIX shared memory regions are unmapped (**shm_open**(3)).
- Open POSIX message queue descriptors are closed (**mq_overview**(7)).
- Any open POSIX named semaphores are closed (**sem_overview**(7)).
- POSIX timers are not preserved (**timer_create**(2)).
- Any open directory streams are closed (**opendir**(3)).
- Memory locks are not preserved (**mlock**(2), *mlockall*(2)).
- Exit handlers are not preserved (**atexit**(3), *on_exit*(3)).
- The floating-point environment is reset to the default (see *fenv*(3)).

The process attributes in the preceding list are all specified in POSIX.1. The following Linux-specific process attributes are also not preserved during an **execve**():

- The process's "dumpable" attribute is set to the value 1, unless a set-user-ID program, a set-group-ID program, or a program with capabilities is being executed, in which case the dumpable flag may instead be reset to the value in */proc/sys/fs/suid_dumpable*, in the circumstances described under **PR_SET_DUMPABLE** in *prctl*(2). Note that changes to the "dumpable" attribute may cause ownership of files in the process's */proc/*pid directory to change to *root:root*, as described in *proc*(5).
- The *prctl*(2) **PR_SET_KEEPCAPS** flag is cleared.
- (Since Linux 2.4.36 / 2.6.23) If a set-user-ID or set-group-ID program is being executed, then the parent death signal set by *prctl*(2) **PR_SET_PDEATHSIG** flag is cleared.
- The process name, as set by *prctl*(2) **PR_SET_NAME** (and displayed by *ps −o comm*), is reset to the name of the new executable file.
- The **SECBIT_KEEP_CAPS** *securebits* flag is cleared. See *capabilities*(7).
- The termination signal is reset to **SIGCHLD** (see *clone*(2)).
- The file descriptor table is unshared, undoing the effect of the **CLONE_FILES** flag of *clone*(2).

Note the following further points:

- All threads other than the calling thread are destroyed during an **execve**(). Mutexes, condition variables, and other pthreads objects are not preserved.
- The equivalent of *setlocale(LC_ALL, "C")* is executed at program start-up.
- POSIX.1 specifies that the dispositions of any signals that are ignored or set to the default are left unchanged. POSIX.1 specifies one exception: if **SIGCHLD** is being ignored, then an implementation may leave the disposition unchanged or reset it to the default; Linux does the former.
- Any outstanding asynchronous I/O operations are canceled (**aio_read**(3), *aio_write*(3)).
- For the handling of capabilities during **execve**(), see *capabilities*(7).
- By default, file descriptors remain open across an **execve**(). File descriptors that are marked close-on-exec are closed; see the description of **FD_CLOEXEC** in *fcntl*(2). (If a file descriptor is closed, this will cause the release of all record locks obtained on the underlying file by this process. See *fcntl*(2) for details.) POSIX.1 says that if file descriptors 0, 1, and 2 would otherwise be closed after a successful **execve**(), and the process would gain privilege because the set-user-ID or set-group-ID

mode bit was set on the executed file, then the system may open an unspecified file for each of these file descriptors. As a general principle, no portable program, whether privileged or not, can assume that these three file descriptors will remain closed across an **execve**().

Interpreter scripts

An interpreter script is a text file that has execute permission enabled and whose first line is of the form:

```
#!interpreter [optional-arg]
```

The *interpreter* must be a valid pathname for an executable file.

If the *pathname* argument of **execve**() specifies an interpreter script, then *interpreter* will be invoked with the following arguments:

```
interpreter [optional-arg] pathname arg...
```

where *pathname* is the pathname of the file specified as the first argument of **execve**(), and *arg...* is the series of words pointed to by the *argv* argument of **execve**(), starting at *argv[1]*. Note that there is no way to get the *argv[0]* that was passed to the **execve**() call.

For portable use, *optional-arg* should either be absent, or be specified as a single word (i.e., it should not contain white space); see NOTES below.

Since Linux 2.6.28, the kernel permits the interpreter of a script to itself be a script. This permission is recursive, up to a limit of four recursions, so that the interpreter may be a script which is interpreted by a script, and so on.

Limits on size of arguments and environment

Most UNIX implementations impose some limit on the total size of the command-line argument (*argv*) and environment (*envp*) strings that may be passed to a new program. POSIX.1 allows an implementation to advertise this limit using the **ARG_MAX** constant (either defined in <*limits.h*> or available at run time using the call *sysconf(_SC_ARG_MAX)*).

Before Linux 2.6.23, the memory used to store the environment and argument strings was limited to 32 pages (defined by the kernel constant **MAX_ARG_PAGES**). On architectures with a 4-kB page size, this yields a maximum size of 128 kB.

On Linux 2.6.23 and later, most architectures support a size limit derived from the soft **RLIMIT_STACK** resource limit (see *getrlimit*(2)) that is in force at the time of the **execve**() call. (Architectures with no memory management unit are excepted: they maintain the limit that was in effect before Linux 2.6.23.) This change allows programs to have a much larger argument and/or environment list. For these architectures, the total size is limited to 1/4 of the allowed stack size. (Imposing the 1/4-limit ensures that the new program always has some stack space.) Additionally, the total size is limited to 3/4 of the value of the kernel constant **_STK_LIM** (8 MiB). Since Linux 2.6.25, the kernel also places a floor of 32 pages on this size limit, so that, even when **RLIMIT_STACK** is set very low, applications are guaranteed to have at least as much argument and environment space as was provided by Linux 2.6.22 and earlier. (This guarantee was not provided in Linux 2.6.23 and 2.6.24.) Additionally, the limit per string is 32 pages (the kernel constant **MAX_ARG_STRLEN**), and the maximum number of strings is 0x7FFFFFFF.

RETURN VALUE

On success, **execve**() does not return, on error −1 is returned, and *errno* is set to indicate the error.

ERRORS

E2BIG The total number of bytes in the environment (*envp*) and argument list (*argv*) is too large, an argument or environment string is too long, or the full *pathname* of the executable is too long. The terminating null byte is counted as part of the string length.

EACCES

Search permission is denied on a component of the path prefix of *pathname* or the name of a script interpreter. (See also *path_resolution*(7).)

EACCES

The file or a script interpreter is not a regular file.

EACCES
Execute permission is denied for the file or a script or ELF interpreter.

EACCES
The filesystem is mounted *noexec*.

EAGAIN (since Linux 3.1)
Having changed its real UID using one of the **set*uid**() calls, the caller was—and is now still—above its **RLIMIT_NPROC** resource limit (see *setrlimit*(2)). For a more detailed explanation of this error, see NOTES.

EFAULT
pathname or one of the pointers in the vectors *argv* or *envp* points outside your accessible address space.

EINVAL
An ELF executable had more than one PT_INTERP segment (i.e., tried to name more than one interpreter).

EIO An I/O error occurred.

EISDIR
An ELF interpreter was a directory.

ELIBBAD
An ELF interpreter was not in a recognized format.

ELOOP
Too many symbolic links were encountered in resolving *pathname* or the name of a script or ELF interpreter.

ELOOP
The maximum recursion limit was reached during recursive script interpretation (see "Interpreter scripts", above). Before Linux 3.8, the error produced for this case was **ENOEXEC**.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG
pathname is too long.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
The file *pathname* or a script or ELF interpreter does not exist.

ENOEXEC
An executable is not in a recognized format, is for the wrong architecture, or has some other format error that means it cannot be executed.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix of *pathname* or a script or ELF interpreter is not a directory.

EPERM
The filesystem is mounted *nosuid*, the user is not the superuser, and the file has the set-user-ID or set-group-ID bit set.

EPERM
The process is being traced, the user is not the superuser and the file has the set-user-ID or set-group-ID bit set.

EPERM
A "capability-dumb" applications would not obtain the full set of permitted capabilities granted by the executable file. See *capabilities*(7).

ETXTBSY
The specified executable was open for writing by one or more processes.

VERSIONS

POSIX does not document the #! behavior, but it exists (with some variations) on other UNIX systems.

On Linux, *argv* and *envp* can be specified as NULL. In both cases, this has the same effect as specifying the argument as a pointer to a list containing a single null pointer. **Do not take advantage of this nonstandard and nonportable misfeature!** On many other UNIX systems, specifying *argv* as NULL will result in an error (**EFAULT**). *Some* other UNIX systems treat the *envp==NULL* case the same as Linux.

POSIX.1 says that values returned by *sysconf*(3) should be invariant over the lifetime of a process. However, since Linux 2.6.23, if the **RLIMIT_STACK** resource limit changes, then the value reported by **_SC_ARG_MAX** will also change, to reflect the fact that the limit on space for holding command-line arguments and environment variables has changed.

Interpreter scripts

The kernel imposes a maximum length on the text that follows the "#!" characters at the start of a script; characters beyond the limit are ignored. Before Linux 5.1, the limit is 127 characters. Since Linux 5.1, the limit is 255 characters.

The semantics of the *optional-arg* argument of an interpreter script vary across implementations. On Linux, the entire string following the *interpreter* name is passed as a single argument to the interpreter, and this string can include white space. However, behavior differs on some other systems. Some systems use the first white space to terminate *optional-arg*. On some systems, an interpreter script can have multiple arguments, and white spaces in *optional-arg* are used to delimit the arguments.

Linux (like most other modern UNIX systems) ignores the set-user-ID and set-group-ID bits on scripts.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

With UNIX V6, the argument list of an **exec**() call was ended by 0, while the argument list of *main* was ended by −1. Thus, this argument list was not directly usable in a further **exec**() call. Since UNIX V7, both are NULL.

NOTES

One sometimes sees **execve**() (and the related functions described in *exec*(3)) described as "executing a *new* process" (or similar). This is a highly misleading description: there is no new process; many attributes of the calling process remain unchanged (in particular, its PID). All that **execve**() does is arrange for an existing process (the calling process) to execute a new program.

Set-user-ID and set-group-ID processes can not be *ptrace*(2)d.

The result of mounting a filesystem *nosuid* varies across Linux kernel versions: some will refuse execution of set-user-ID and set-group-ID executables when this would give the user powers they did not have already (and return **EPERM**), some will just ignore the set-user-ID and set-group-ID bits and **exec**() successfully.

In most cases where **execve**() fails, control returns to the original executable image, and the caller of **execve**() can then handle the error. However, in (rare) cases (typically caused by resource exhaustion), failure may occur past the point of no return: the original executable image has been torn down, but the new image could not be completely built. In such cases, the kernel kills the process with a **SIGSEGV** (**SIGKILL** until Linux 3.17) signal.

execve() and EAGAIN

A more detailed explanation of the **EAGAIN** error that can occur (since Linux 3.1) when calling **execve**() is as follows.

The **EAGAIN** error can occur when a *preceding* call to *setuid*(2), *setreuid*(2), or *setresuid*(2) caused the real user ID of the process to change, and that change caused the process to exceed its **RLIMIT_NPROC** resource limit (i.e., the number of processes belonging to the new real UID exceeds the resource limit). From Linux 2.6.0 to Linux 3.0, this caused the **set*uid**() call to fail. (Before Linux 2.6, the resource limit was not imposed on processes that changed their user IDs.)

Since Linux 3.1, the scenario just described no longer causes the **set*uid**() call to fail, because it too often led to security holes where buggy applications didn't check the return status and assumed that—if the caller had root privileges—the call would always succeed. Instead, the **set*uid**() calls now successfully change the real UID, but the kernel sets an internal flag, named **PF_NPROC_EXCEEDED**, to note that the **RLIMIT_NPROC** resource limit has been exceeded. If the **PF_NPROC_EXCEEDED** flag is set and the resource limit is still exceeded at the time of a subsequent **execve**() call, that call fails with the error **EAGAIN**. This kernel logic ensures that the **RLIMIT_NPROC** resource limit is still enforced for the common privileged daemon workflow—namely, *fork*(2) + **set*uid**() + **execve**().

If the resource limit was not still exceeded at the time of the **execve**() call (because other processes belonging to this real UID terminated between the **set*uid**() call and the **execve**() call), then the **execve**() call succeeds and the kernel clears the **PF_NPROC_EXCEEDED** process flag. The flag is also cleared if a subsequent call to *fork*(2) by this process succeeds.

EXAMPLES

The following program is designed to be execed by the second program below. It just echoes its command-line arguments, one per line.

```
/* myecho.c */

#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    for (size_t j = 0; j < argc; j++)
        printf("argv[%zu]: %s\n", j, argv[j]);

    exit(EXIT_SUCCESS);
}
```

This program can be used to exec the program named in its command-line argument:

```
/* execve.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    static char *newargv[] = { NULL, "hello", "world", NULL };
    static char *newenviron[] = { NULL };

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file-to-exec>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    newargv[0] = argv[1];

    execve(argv[1], newargv, newenviron);
    perror("execve");   /* execve() returns only on error */
    exit(EXIT_FAILURE);
}
```

We can use the second program to exec the first as follows:

```
$ cc myecho.c -o myecho
$ cc execve.c -o execve
$ ./execve ./myecho
```

```
argv[0]: ./myecho
argv[1]: hello
argv[2]: world
```

We can also use these programs to demonstrate the use of a script interpreter. To do this we create a script whose "interpreter" is our *myecho* program:

```
$ cat > script
#!./myecho script-arg
^D
$ chmod +x script
```

We can then use our program to exec the script:

```
$ ./execve ./script
argv[0]: ./myecho
argv[1]: script-arg
argv[2]: ./script
argv[3]: hello
argv[4]: world
```

SEE ALSO

chmod(2), *execveat*(2), *fork*(2), *get_robust_list*(2), *ptrace*(2), *exec*(3), *fexecve*(3), *getauxval*(3), *getopt*(3), *system*(3), *capabilities*(7), *credentials*(7), *environ*(7), *path_resolution*(7), *ld.so*(8)

NAME

execveat − execute program relative to a directory file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/fcntl.h>      /* Definition of AT_* constants */
#include <unistd.h>

int execveat(int dirfd, const char *pathname,
             char *const _Nullable argv[],
             char *const _Nullable envp[],
             int flags);
```

DESCRIPTION

The **execveat**() system call executes the program referred to by the combination of *dirfd* and *pathname*. It operates in exactly the same way as *execve*(2), except for the differences described in this manual page.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by *execve*(2) for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *execve*(2)).

If *pathname* is absolute, then *dirfd* is ignored.

If *pathname* is an empty string and the **AT_EMPTY_PATH** flag is specified, then the file descriptor *dirfd* specifies the file to be executed (i.e., *dirfd* refers to an executable file, rather than a directory).

The *flags* argument is a bit mask that can include zero or more of the following flags:

AT_EMPTY_PATH
: If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag).

AT_SYMLINK_NOFOLLOW
: If the file identified by *dirfd* and a non-NULL *pathname* is a symbolic link, then the call fails with the error **ELOOP**.

RETURN VALUE

On success, **execveat**() does not return. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The same errors that occur for *execve*(2) can also occur for **execveat**(). The following additional errors can occur for **execveat**():

pathname
: is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EINVAL
: Invalid flag specified in *flags*.

ELOOP
: *flags* includes **AT_SYMLINK_NOFOLLOW** and the file identified by *dirfd* and a non-NULL *pathname* is a symbolic link.

ENOENT
: The program identified by *dirfd* and *pathname* requires the use of an interpreter program (such as a script starting with "#!"), but the file descriptor *dirfd* was opened with the **O_CLOEXEC** flag, with the result that the program file is inaccessible to the launched interpreter. See BUGS.

ENOTDIR
: *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

STANDARDS

Linux.

HISTORY

Linux 3.19, glibc 2.34.

NOTES

In addition to the reasons explained in *openat*(2), the **execveat**() system call is also needed to allow *fexecve*(3) to be implemented on systems that do not have the */proc* filesystem mounted.

When asked to execute a script file, the *argv[0]* that is passed to the script interpreter is a string of the form */dev/fd/N* or */dev/fd/N/P*, where *N* is the number of the file descriptor passed via the *dirfd* argument. A string of the first form occurs when **AT_EMPTY_PATH** is employed. A string of the second form occurs when the script is specified via both *dirfd* and *pathname*; in this case, *P* is the value given in *pathname*.

For the same reasons described in *fexecve*(3), the natural idiom when using **execveat**() is to set the close-on-exec flag on *dirfd*. (But see BUGS.)

BUGS

The **ENOENT** error described above means that it is not possible to set the close-on-exec flag on the file descriptor given to a call of the form:

```
execveat(fd, "", argv, envp, AT_EMPTY_PATH);
```

However, the inability to set the close-on-exec flag means that a file descriptor referring to the script leaks through to the script itself. As well as wasting a file descriptor, this leakage can lead to file-descriptor exhaustion in scenarios where scripts recursively employ **execveat**().

SEE ALSO

execve(2), *openat*(2), *fexecve*(3)

NAME

_exit, _Exit − terminate the calling process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

[[noreturn]] void _exit(int status);

#include <stdlib.h>

[[noreturn]] void _Exit(int status);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

_Exit():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

_exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed. Any children of the process are inherited by *init*(1) (or by the nearest "subreaper" process as defined through the use of the *prctl*(2) **PR_SET_CHILD_SUBREAPER** operation). The process's parent is sent a **SIGCHLD** signal.

The value *status & 0xFF* is returned to the parent process as the process's exit status, and can be collected by the parent using one of the *wait*(2) family of calls.

The function **_Exit**() is equivalent to **_exit**().

RETURN VALUE

These functions do not return.

STANDARDS

_exit() POSIX.1-2008.

_Exit() C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

_Exit() was introduced by C99.

NOTES

For a discussion on the effects of an exit, the transmission of exit status, zombie processes, signals sent, and so on, see *exit*(3).

The function **_exit**() is like *exit*(3), but does not call any functions registered with *atexit*(3) or *on_exit*(3). Open *stdio*(3) streams are not flushed. On the other hand, **_exit**() does close open file descriptors, and this may cause an unknown delay, waiting for pending output to finish. If the delay is undesired, it may be useful to call functions like *tcflush*(3) before calling **_exit**(). Whether any pending I/O is canceled, and which pending I/O may be canceled upon **_exit**(), is implementation-dependent.

C library/kernel differences

The text above in DESCRIPTION describes the traditional effect of **_exit**(), which is to terminate a process, and these are the semantics specified by POSIX.1 and implemented by the C library wrapper function. On modern systems, this means termination of all threads in the process.

By contrast with the C library wrapper function, the raw Linux **_exit**() system call terminates only the calling thread, and actions such as reparenting child processes or sending **SIGCHLD** to the parent process are performed only if this is the last thread in the thread group.

Up to glibc 2.3, the **_exit**() wrapper function invoked the kernel system call of the same name. Since glibc 2.3, the wrapper function invokes *exit_group*(2), in order to terminate all of the threads in a process.

SEE ALSO

execve(2), *exit_group*(2), *fork*(2), *kill*(2), *wait*(2), *wait4*(2), *waitpid*(2), *atexit*(3), *exit*(3), *on_exit*(3), *termios*(3)

NAME

exit_group − exit all threads in a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

[[noreturn]] void syscall(SYS_exit_group, int status);
```

Note: glibc provides no wrapper for **exit_group**(), necessitating the use of *syscall*(2).

DESCRIPTION

This system call terminates all threads in the calling process's thread group.

RETURN VALUE

This system call does not return.

STANDARDS

Linux.

HISTORY

Linux 2.5.35.

NOTES

Since glibc 2.3, this is the system call invoked when the *_exit*(2) wrapper function is called.

SEE ALSO

_exit(2)

NAME

fallocate − manipulate file space

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>

int fallocate(int fd, int mode, off_t offset, off_t len);
```

DESCRIPTION

This is a nonportable, Linux-specific system call. For the portable, POSIX.1-specified method of ensuring that space is allocated for a file, see *posix_fallocate*(3).

fallocate() allows the caller to directly manipulate the allocated disk space for the file referred to by *fd* for the byte range starting at *offset* and continuing for *len* bytes.

The *mode* argument determines the operation to be performed on the given range. Details of the supported operations are given in the subsections below.

Allocating disk space

The default operation (i.e., *mode* is zero) of **fallocate**() allocates the disk space within the range specified by *offset* and *len*. The file size (as reported by *stat*(2)) will be changed if *offset*+*len* is greater than the file size. Any subregion within the range specified by *offset* and *len* that did not contain data before the call will be initialized to zero. This default behavior closely resembles the behavior of the *posix_fallocate*(3) library function, and is intended as a method of optimally implementing that function.

After a successful call, subsequent writes into the range specified by *offset* and *len* are guaranteed not to fail because of lack of disk space.

If the **FALLOC_FL_KEEP_SIZE** flag is specified in *mode*, the behavior of the call is similar, but the file size will not be changed even if *offset*+*len* is greater than the file size. Preallocating zeroed blocks beyond the end of the file in this manner is useful for optimizing append workloads.

If the **FALLOC_FL_UNSHARE_RANGE** flag is specified in *mode*, shared file data extents will be made private to the file to guarantee that a subsequent write will not fail due to lack of space. Typically, this will be done by performing a copy-on-write operation on all shared data in the file. This flag may not be supported by all filesystems.

Because allocation is done in block size chunks, **fallocate**() may allocate a larger range of disk space than was specified.

Deallocating file space

Specifying the **FALLOC_FL_PUNCH_HOLE** flag (available since Linux 2.6.38) in *mode* deallocates space (i.e., creates a hole) in the byte range starting at *offset* and continuing for *len* bytes. Within the specified range, partial filesystem blocks are zeroed, and whole filesystem blocks are removed from the file. After a successful call, subsequent reads from this range will return zeros.

The **FALLOC_FL_PUNCH_HOLE** flag must be ORed with **FALLOC_FL_KEEP_SIZE** in *mode*; in other words, even when punching off the end of the file, the file size (as reported by *stat*(2)) does not change.

Not all filesystems support **FALLOC_FL_PUNCH_HOLE**; if a filesystem doesn't support the operation, an error is returned. The operation is supported on at least the following filesystems:

- XFS (since Linux 2.6.38)
- ext4 (since Linux 3.0)
- Btrfs (since Linux 3.7)
- *tmpfs*(5) (since Linux 3.5)
- *gfs2*(5) (since Linux 4.16)

Collapsing file space

Specifying the **FALLOC_FL_COLLAPSE_RANGE** flag (available since Linux 3.15) in *mode* removes a byte range from a file, without leaving a hole. The byte range to be collapsed starts at *offset*

and continues for *len* bytes. At the completion of the operation, the contents of the file starting at the location *offset*+*len* will be appended at the location *offset*, and the file will be *len* bytes smaller.

A filesystem may place limitations on the granularity of the operation, in order to ensure efficient implementation. Typically, *offset* and *len* must be a multiple of the filesystem logical block size, which varies according to the filesystem type and configuration. If a filesystem has such a requirement, **fallocate**() fails with the error **EINVAL** if this requirement is violated.

If the region specified by *offset* plus *len* reaches or passes the end of file, an error is returned; instead, use *ftruncate*(2) to truncate a file.

No other flags may be specified in *mode* in conjunction with **FALLOC_FL_COLLAPSE_RANGE**.

As at Linux 3.15, **FALLOC_FL_COLLAPSE_RANGE** is supported by ext4 (only for extent-based files) and XFS.

Zeroing file space

Specifying the **FALLOC_FL_ZERO_RANGE** flag (available since Linux 3.15) in *mode* zeros space in the byte range starting at *offset* and continuing for *len* bytes. Within the specified range, blocks are preallocated for the regions that span the holes in the file. After a successful call, subsequent reads from this range will return zeros.

Zeroing is done within the filesystem preferably by converting the range into unwritten extents. This approach means that the specified range will not be physically zeroed out on the device (except for partial blocks at the either end of the range), and I/O is (otherwise) required only to update metadata.

If the **FALLOC_FL_KEEP_SIZE** flag is additionally specified in *mode*, the behavior of the call is similar, but the file size will not be changed even if *offset*+*len* is greater than the file size. This behavior is the same as when preallocating space with **FALLOC_FL_KEEP_SIZE** specified.

Not all filesystems support **FALLOC_FL_ZERO_RANGE**; if a filesystem doesn't support the operation, an error is returned. The operation is supported on at least the following filesystems:

- XFS (since Linux 3.15)
- ext4, for extent-based files (since Linux 3.15)
- SMB3 (since Linux 3.17)
- Btrfs (since Linux 4.16)

Increasing file space

Specifying the **FALLOC_FL_INSERT_RANGE** flag (available since Linux 4.1) in *mode* increases the file space by inserting a hole within the file size without overwriting any existing data. The hole will start at *offset* and continue for *len* bytes. When inserting the hole inside file, the contents of the file starting at *offset* will be shifted upward (i.e., to a higher file offset) by *len* bytes. Inserting a hole inside a file increases the file size by *len* bytes.

This mode has the same limitations as **FALLOC_FL_COLLAPSE_RANGE** regarding the granularity of the operation. If the granularity requirements are not met, **fallocate**() fails with the error **EINVAL**. If the *offset* is equal to or greater than the end of file, an error is returned. For such operations (i.e., inserting a hole at the end of file), *ftruncate*(2) should be used.

No other flags may be specified in *mode* in conjunction with **FALLOC_FL_INSERT_RANGE**.

FALLOC_FL_INSERT_RANGE requires filesystem support. Filesystems that support this operation include XFS (since Linux 4.1) and ext4 (since Linux 4.2).

RETURN VALUE

On success, **fallocate**() returns zero. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF

fd is not a valid file descriptor, or is not opened for writing.

EFBIG

offset+*len* exceeds the maximum file size.

EFBIG

mode is **FALLOC_FL_INSERT_RANGE**, and the current file size+*len* exceeds the maximum file size.

EINTR
A signal was caught during execution; see *signal*(7).

EINVAL
offset was less than 0, or *len* was less than or equal to 0.

EINVAL
mode is **FALLOC_FL_COLLAPSE_RANGE** and the range specified by *offset* plus *len* reaches or passes the end of the file.

EINVAL
mode is **FALLOC_FL_INSERT_RANGE** and the range specified by *offset* reaches or passes the end of the file.

EINVAL
mode is **FALLOC_FL_COLLAPSE_RANGE** or **FALLOC_FL_INSERT_RANGE**, but either *offset* or *len* is not a multiple of the filesystem block size.

EINVAL
mode contains one of **FALLOC_FL_COLLAPSE_RANGE** or **FALLOC_FL_INSERT_RANGE** and also other flags; no other flags are permitted with **FALLOC_FL_COLLAPSE_RANGE** or **FALLOC_FL_INSERT_RANGE**.

EINVAL
mode is **FALLOC_FL_COLLAPSE_RANGE**, **FALLOC_FL_ZERO_RANGE**, or **FALLOC_FL_INSERT_RANGE**, but the file referred to by *fd* is not a regular file.

EIO An I/O error occurred while reading from or writing to a filesystem.

ENODEV
fd does not refer to a regular file or a directory. (If *fd* is a pipe or FIFO, a different error results.)

ENOSPC
There is not enough space left on the device containing the file referred to by *fd*.

ENOSYS
This kernel does not implement **fallocate**().

EOPNOTSUPP
The filesystem containing the file referred to by *fd* does not support this operation; or the *mode* is not supported by the filesystem containing the file referred to by *fd*.

EPERM
The file referred to by *fd* is marked immutable (see *chattr*(1)).

EPERM
mode specifies **FALLOC_FL_PUNCH_HOLE**, **FALLOC_FL_COLLAPSE_RANGE**, or **FALLOC_FL_INSERT_RANGE** and the file referred to by *fd* is marked append-only (see *chattr*(1)).

EPERM
The operation was prevented by a file seal; see *fcntl*(2).

ESPIPE
fd refers to a pipe or FIFO.

ETXTBSY
mode specifies **FALLOC_FL_COLLAPSE_RANGE** or **FALLOC_FL_INSERT_RANGE**, but the file referred to by *fd* is currently being executed.

STANDARDS
Linux.

HISTORY
fallocate()
Linux 2.6.23, glibc 2.10.

FALLOC_FL_*
glibc 2.18.

SEE ALSO

fallocate(1), *ftruncate*(2), *posix_fadvise*(3), *posix_fallocate*(3)

NAME

fanotify_init − create and initialize fanotify group

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fcntl.h> /* Definition of **O_*** constants */
#include <sys/fanotify.h>

int fanotify_init(unsigned int *flags***, unsigned int** *event_f_flags***);**

DESCRIPTION

For an overview of the fanotify API, see *fanotify*(7).

fanotify_init() initializes a new fanotify group and returns a file descriptor for the event queue associated with the group.

The file descriptor is used in calls to *fanotify_mark*(2) to specify the files, directories, mounts, or filesystems for which fanotify events shall be created. These events are received by reading from the file descriptor. Some events are only informative, indicating that a file has been accessed. Other events can be used to determine whether another application is permitted to access a file or directory. Permission to access filesystem objects is granted by writing to the file descriptor.

Multiple programs may be using the fanotify interface at the same time to monitor the same files.

The number of fanotify groups per user is limited. See *fanotify*(7) for details about this limit.

The *flags* argument contains a multi-bit field defining the notification class of the listening application and further single bit fields specifying the behavior of the file descriptor.

If multiple listeners for permission events exist, the notification class is used to establish the sequence in which the listeners receive the events.

Only one of the following notification classes may be specified in *flags*:

FAN_CLASS_PRE_CONTENT
: This value allows the receipt of events notifying that a file has been accessed and events for permission decisions if a file may be accessed. It is intended for event listeners that need to access files before they contain their final data. This notification class might be used by hierarchical storage managers, for example. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_CLASS_CONTENT
: This value allows the receipt of events notifying that a file has been accessed and events for permission decisions if a file may be accessed. It is intended for event listeners that need to access files when they already contain their final content. This notification class might be used by malware detection programs, for example. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_CLASS_NOTIF
: This is the default value. It does not need to be specified. This value only allows the receipt of events notifying that a file has been accessed. Permission decisions before the file is accessed are not possible.

Listeners with different notification classes will receive events in the order **FAN_CLASS_PRE_CONTENT**, **FAN_CLASS_CONTENT**, **FAN_CLASS_NOTIF**. The order of notification for listeners in the same notification class is undefined.

The following bits can additionally be set in *flags*:

FAN_CLOEXEC
: Set the close-on-exec flag (**FD_CLOEXEC**) on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2).

FAN_NONBLOCK
: Enable the nonblocking flag (**O_NONBLOCK**) for the file descriptor. Reading from the file descriptor will not block. Instead, if no data is available, *read*(2) fails with the error **EAGAIN**.

FAN_UNLIMITED_QUEUE
Remove the limit on the number of events in the event queue. See *fanotify*(7) for details about this limit. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_UNLIMITED_MARKS
Remove the limit on the number of fanotify marks per user. See *fanotify*(7) for details about this limit. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_REPORT_TID (since Linux 4.20)
Report thread ID (TID) instead of process ID (PID) in the *pid* field of the *struct fanotify_event_metadata* supplied to *read*(2) (see *fanotify*(7)). Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_ENABLE_AUDIT (since Linux 4.15)
Enable generation of audit log records about access mediation performed by permission events. The permission event response has to be marked with the **FAN_AUDIT** flag for an audit log record to be generated. Use of this flag requires the **CAP_AUDIT_WRITE** capability.

FAN_REPORT_FID (since Linux 5.1)
This value allows the receipt of events which contain additional information about the underlying filesystem object correlated to an event. An additional record of type **FAN_EVENT_INFO_TYPE_FID** encapsulates the information about the object and is included alongside the generic event metadata structure. The file descriptor that is used to represent the object correlated to an event is instead substituted with a file handle. It is intended for applications that may find the use of a file handle to identify an object more suitable than a file descriptor. Additionally, it may be used for applications monitoring a directory or a filesystem that are interested in the directory entry modification events **FAN_CREATE**, **FAN_DELETE**, **FAN_MOVE**, and **FAN_RENAME**, or in events such as **FAN_ATTRIB**, **FAN_DELETE_SELF**, and **FAN_MOVE_SELF**. All the events above require an fanotify group that identifies filesystem objects by file handles. Note that without the flag **FAN_REPORT_TARGET_FID**, for the directory entry modification events, there is an information record that identifies the modified directory and not the created/deleted/moved child object. The use of **FAN_CLASS_CONTENT** or **FAN_CLASS_PRE_CONTENT** is not permitted with this flag and will result in the error **EINVAL**. See *fanotify*(7) for additional details.

FAN_REPORT_DIR_FID (since Linux 5.9)
Events for fanotify groups initialized with this flag will contain (see exceptions below) additional information about a directory object correlated to an event. An additional record of type **FAN_EVENT_INFO_TYPE_DFID** encapsulates the information about the directory object and is included alongside the generic event metadata structure. For events that occur on a non-directory object, the additional structure includes a file handle that identifies the parent directory filesystem object. Note that there is no guarantee that the directory filesystem object will be found at the location described by the file handle information at the time the event is received. When combined with the flag **FAN_REPORT_FID**, two records may be reported with events that occur on a non-directory object, one to identify the non-directory object itself and one to identify the parent directory object. Note that in some cases, a filesystem object does not have a parent, for example, when an event occurs on an unlinked but open file. In that case, with the **FAN_REPORT_FID** flag, the event will be reported with only one record to identify the non-directory object itself, because there is no directory associated with the event. Without the **FAN_REPORT_FID** flag, no event will be reported. See *fanotify*(7) for additional details.

FAN_REPORT_NAME (since Linux 5.9)
Events for fanotify groups initialized with this flag will contain additional information about the name of the directory entry correlated to an event. This flag must be provided in conjunction with the flag **FAN_REPORT_DIR_FID**. Providing this flag value without **FAN_REPORT_DIR_FID** will result in the error **EINVAL**. This flag may be combined with the flag **FAN_REPORT_FID**. An additional record of type **FAN_EVENT_INFO_TYPE_DFID_NAME**, which encapsulates the information about the directory entry, is included alongside the generic event metadata structure and substitutes the additional information record of type **FAN_EVENT_INFO_TYPE_DFID**. The additional record includes a file handle that identifies a directory filesystem object followed by a name

that identifies an entry in that directory. For the directory entry modification events **FAN_CREATE**, **FAN_DELETE**, and **FAN_MOVE**, the reported name is that of the created/deleted/moved directory entry. The event **FAN_RENAME** may contain two information records. One of type **FAN_EVENT_INFO_TYPE_OLD_DFID_NAME** identifying the old directory entry, and another of type **FAN_EVENT_INFO_TYPE_NEW_DFID_NAME** identifying the new directory entry. For other events that occur on a directory object, the reported file handle is that of the directory object itself and the reported name is '.'. For other events that occur on a non-directory object, the reported file handle is that of the parent directory object and the reported name is the name of a directory entry where the object was located at the time of the event. The rationale behind this logic is that the reported directory file handle can be passed to *open_by_handle_at*(2) to get an open directory file descriptor and that file descriptor along with the reported name can be used to call *fstatat*(2). The same rule that applies to record type **FAN_EVENT_INFO_TYPE_DFID** also applies to record type **FAN_EVENT_INFO_TYPE_DFID_NAME**: if a non-directory object has no parent, either the event will not be reported or it will be reported without the directory entry information. Note that there is no guarantee that the filesystem object will be found at the location described by the directory entry information at the time the event is received. See *fanotify*(7) for additional details.

FAN_REPORT_DFID_NAME
This is a synonym for (**FAN_REPORT_DIR_FID**|**FAN_REPORT_NAME**).

FAN_REPORT_TARGET_FID (since Linux 5.17)
Events for fanotify groups initialized with this flag will contain additional information about the child correlated with directory entry modification events. This flag must be provided in conjunction with the flags **FAN_REPORT_FID**, **FAN_REPORT_DIR_FID** and **FAN_REPORT_NAME**. or else the error **EINVAL** will be returned. For the directory entry modification events **FAN_CREATE**, **FAN_DELETE**, **FAN_MOVE**, and **FAN_RENAME**, an additional record of type **FAN_EVENT_INFO_TYPE_FID**, is reported in addition to the information records of type **FAN_EVENT_INFO_TYPE_DFID**, **FAN_EVENT_INFO_TYPE_DFID_NAME**, **FAN_EVENT_INFO_TYPE_OLD_DFID_NAME**, and **FAN_EVENT_INFO_TYPE_NEW_DFID_NAME**. The additional record includes a file handle that identifies the filesystem child object that the directory entry is referring to.

FAN_REPORT_DFID_NAME_TARGET
This is a synonym for (**FAN_REPORT_DFID_NAME**|**FAN_REPORT_FID**|**FAN_REPORT_TARGET_FID**).

FAN_REPORT_PIDFD (since Linux 5.15)
Events for fanotify groups initialized with this flag will contain an additional information record alongside the generic *fanotify_event_metadata* structure. This information record will be of type **FAN_EVENT_INFO_TYPE_PIDFD** and will contain a pidfd for the process that was responsible for generating an event. A pidfd returned in this information record object is no different to the pidfd that is returned when calling *pidfd_open*(2). Usage of this information record are for applications that may be interested in reliably determining whether the process responsible for generating an event has been recycled or terminated. The use of the **FAN_REPORT_TID** flag along with **FAN_REPORT_PIDFD** is currently not supported and attempting to do so will result in the error **EINVAL** being returned. This limitation is currently imposed by the pidfd API as it currently only supports the creation of pidfds for thread-group leaders. Creating pidfds for non-thread-group leaders may be supported at some point in the future, so this restriction may eventually be lifted. For more details on information records, see *fanotify*(7).

The *event_f_flags* argument defines the file status flags that will be set on the open file descriptions that are created for fanotify events. For details of these flags, see the description of the *flags* values in *open*(2). *event_f_flags* includes a multi-bit field for the access mode. This field can take the following values:

O_RDONLY
This value allows only read access.

O_WRONLY
This value allows only write access.

O_RDWR
This value allows read and write access.

Additional bits can be set in *event_f_flags*. The most useful values are:

O_LARGEFILE
Enable support for files exceeding 2 GB. Failing to set this flag will result in an **EOVERFLOW** error when trying to open a large file which is monitored by an fanotify group on a 32-bit system.

O_CLOEXEC (since Linux 3.18)
Enable the close-on-exec flag for the file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

The following are also allowable: **O_APPEND**, **O_DSYNC**, **O_NOATIME**, **O_NONBLOCK**, and **O_SYNC**. Specifying any other flag in *event_f_flags* yields the error **EINVAL** (but see BUGS).

RETURN VALUE

On success, **fanotify_init**() returns a new file descriptor. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
An invalid value was passed in *flags* or *event_f_flags*. **FAN_ALL_INIT_FLAGS** (deprecated since Linux 4.20) defines all allowable bits for *flags*.

EMFILE
The number of fanotify groups for this user exceeds the limit. See *fanotify*(7) for details about this limit.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENOMEM
The allocation of memory for the notification group failed.

ENOSYS
This kernel does not implement **fanotify_init**(). The fanotify API is available only if the kernel was configured with **CONFIG_FANOTIFY**.

EPERM
The operation is not permitted because the caller lacks a required capability.

VERSIONS

Prior to Linux 5.13, calling **fanotify_init**() required the **CAP_SYS_ADMIN** capability. Since Linux 5.13, users may call **fanotify_init**() without the **CAP_SYS_ADMIN** capability to create and initialize an fanotify group with limited functionality.

The limitations imposed on an event listener created by a user without the **CAP_SYS_ADMIN** capability are as follows:

- The user cannot request for an unlimited event queue by using **FAN_UNLIMITED_QUEUE**.
- The user cannot request for an unlimited number of marks by using **FAN_UNLIMITED_MARKS**.
- The user cannot request to use either notification classes **FAN_CLASS_CONTENT** or **FAN_CLASS_PRE_CONTENT**. This means that user cannot request permission events.
- The user is required to create a group that identifies filesystem objects by file handles, for example, by providing the **FAN_REPORT_FID** flag.
- The user is limited to only mark inodes. The ability to mark a mount or filesystem via **fanotify_mark**() through the use of **FAN_MARK_MOUNT** or **FAN_MARK_FILESYSTEM** is not permitted.

- The event object in the event queue is limited in terms of the information that is made available to the unprivileged user. A user will also not receive the pid that generated the event, unless the listening process itself generated the event.

STANDARDS

Linux.

HISTORY

Linux 2.6.37.

BUGS

The following bug was present before Linux 3.18:

- The **O_CLOEXEC** is ignored when passed in *event_f_flags*.

The following bug was present before Linux 3.14:

- The *event_f_flags* argument is not checked for invalid flags. Flags that are intended only for internal use, such as **FMODE_EXEC**, can be set, and will consequently be set for the file descriptors returned when reading from the fanotify file descriptor.

SEE ALSO

fanotify_mark(2), *fanotify*(7)

NAME

fanotify_mark − add, remove, or modify an fanotify mark on a filesystem object

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/fanotify.h>

int fanotify_mark(int fanotify_fd, unsigned int flags,
                  uint64_t mask, int dirfd,
                  const char *_Nullable pathname);
```

DESCRIPTION

For an overview of the fanotify API, see *fanotify*(7).

fanotify_mark() adds, removes, or modifies an fanotify mark on a filesystem object. The caller must have read permission on the filesystem object that is to be marked.

The *fanotify_fd* argument is a file descriptor returned by *fanotify_init*(2).

flags is a bit mask describing the modification to perform. It must include exactly one of the following values:

FAN_MARK_ADD
: The events in *mask* will be added to the mark mask (or to the ignore mask). *mask* must be nonempty or the error **EINVAL** will occur.

FAN_MARK_REMOVE
: The events in argument *mask* will be removed from the mark mask (or from the ignore mask). *mask* must be nonempty or the error **EINVAL** will occur.

FAN_MARK_FLUSH
: Remove either all marks for filesystems, all marks for mounts, or all marks for directories and files from the fanotify group. If *flags* contains **FAN_MARK_MOUNT**, all marks for mounts are removed from the group. If *flags* contains **FAN_MARK_FILESYSTEM**, all marks for filesystems are removed from the group. Otherwise, all marks for directories and files are removed. No flag other than, and at most one of, the flags **FAN_MARK_MOUNT** or **FAN_MARK_FILESYSTEM** can be used in conjunction with **FAN_MARK_FLUSH**. *mask* is ignored.

If none of the values above is specified, or more than one is specified, the call fails with the error **EINVAL**.

In addition, zero or more of the following values may be ORed into *flags*:

FAN_MARK_DONT_FOLLOW
: If *pathname* is a symbolic link, mark the link itself, rather than the file to which it refers. (By default, **fanotify_mark**() dereferences *pathname* if it is a symbolic link.)

FAN_MARK_ONLYDIR
: If the filesystem object to be marked is not a directory, the error **ENOTDIR** shall be raised.

FAN_MARK_MOUNT
: Mark the mount specified by *pathname*. If *pathname* is not itself a mount point, the mount containing *pathname* will be marked. All directories, subdirectories, and the contained files of the mount will be monitored. The events which require that filesystem objects are identified by file handles, such as **FAN_CREATE**, **FAN_ATTRIB**, **FAN_MOVE**, and **FAN_DELETE_SELF**, cannot be provided as a *mask* when *flags* contains **FAN_MARK_MOUNT**. Attempting to do so will result in the error **EINVAL** being returned. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_MARK_FILESYSTEM (since Linux 4.20)
: Mark the filesystem specified by *pathname*. The filesystem containing *pathname* will be marked. All the contained files and directories of the filesystem from any mount point will be monitored. Use of this flag requires the **CAP_SYS_ADMIN** capability.

FAN_MARK_IGNORED_MASK
The events in *mask* shall be added to or removed from the ignore mask. Note that the flags **FAN_ONDIR**, and **FAN_EVENT_ON_CHILD** have no effect when provided with this flag. The effect of setting the flags **FAN_ONDIR**, and **FAN_EVENT_ON_CHILD** in the mark mask on the events that are set in the ignore mask is undefined and depends on the Linux kernel version. Specifically, prior to Linux 5.9, setting a mark mask on a file and a mark with ignore mask on its parent directory would not result in ignoring events on the file, regardless of the **FAN_EVENT_ON_CHILD** flag in the parent directory's mark mask. When the ignore mask is updated with the **FAN_MARK_IGNORED_MASK** flag on a mark that was previously updated with the **FAN_MARK_IGNORE** flag, the update fails with **EEXIST** error.

FAN_MARK_IGNORE (since Linux 6.0)
This flag has a similar effect as setting the **FAN_MARK_IGNORED_MASK** flag. The events in *mask* shall be added to or removed from the ignore mask. Unlike the **FAN_MARK_IGNORED_MASK** flag, this flag also has the effect that the **FAN_ONDIR**, and **FAN_EVENT_ON_CHILD** flags take effect on the ignore mask. Specifically, unless the **FAN_ONDIR** flag is set with **FAN_MARK_IGNORE**, events on directories will not be ignored. If the flag **FAN_EVENT_ON_CHILD** is set with **FAN_MARK_IGNORE**, events on children will be ignored. For example, a mark on a directory with combination of a mask with **FAN_CREATE** event and **FAN_ONDIR** flag and an ignore mask with **FAN_CREATE** event and without **FAN_ONDIR** flag, will result in getting only the events for creation of sub-directories. When using the **FAN_MARK_IGNORE** flag to add to an ignore mask of a mount, filesystem, or directory inode mark, the **FAN_MARK_IGNORED_SURV_MODIFY** flag must be specified. Failure to do so will results with **EINVAL** or **EISDIR** error.

FAN_MARK_IGNORED_SURV_MODIFY
The ignore mask shall survive modify events. If this flag is not set, the ignore mask is cleared when a modify event occurs on the marked object. Omitting this flag is typically used to suppress events (e.g., **FAN_OPEN**) for a specific file, until that specific file's content has been modified. It is far less useful to suppress events on an entire filesystem, or mount, or on all files inside a directory, until some file's content has been modified. For this reason, the **FAN_MARK_IGNORE** flag requires the **FAN_MARK_IGNORED_SURV_MODIFY** flag on a mount, filesystem, or directory inode mark. This flag cannot be removed from a mark once set. When the ignore mask is updated without this flag on a mark that was previously updated with the **FAN_MARK_IGNORE** and **FAN_MARK_IGNORED_SURV_MODIFY** flags, the update fails with **EEXIST** error.

FAN_MARK_IGNORE_SURV
This is a synonym for (**FAN_MARK_IGNORE**|**FAN_MARK_IGNORED_SURV_MODIFY**).

FAN_MARK_EVICTABLE (since Linux 5.19)
When an inode mark is created with this flag, the inode object will not be pinned to the inode cache, therefore, allowing the inode object to be evicted from the inode cache when the memory pressure on the system is high. The eviction of the inode object results in the evictable mark also being lost. When the mask of an evictable inode mark is updated without using the **FAN_MARK_EVICATBLE** flag, the marked inode is pinned to inode cache and the mark is no longer evictable. When the mask of a non-evictable inode mark is updated with the **FAN_MARK_EVICTABLE** flag, the inode mark remains non-evictable and the update fails with **EEXIST** error. Mounts and filesystems are not evictable objects, therefore, an attempt to create a mount mark or a filesystem mark with the **FAN_MARK_EVICTABLE** flag, will result in the error **EINVAL**. For example, inode marks can be used in combination with mount marks to reduce the amount of events from noninteresting paths. The event listener reads events, checks if the path reported in the event is of interest, and if it is not, the listener sets a mark with an ignore mask on the directory. Evictable inode marks allow using this method for a large number of directories without the concern of pinning all inodes and exhausting the system's memory.

mask defines which events shall be listened for (or which shall be ignored). It is a bit mask composed of the following values:

FAN_ACCESS
Create an event when a file or directory (but see BUGS) is accessed (read).

FAN_MODIFY
Create an event when a file is modified (write).

FAN_CLOSE_WRITE
Create an event when a writable file is closed.

FAN_CLOSE_NOWRITE
Create an event when a read-only file or directory is closed.

FAN_OPEN
Create an event when a file or directory is opened.

FAN_OPEN_EXEC (since Linux 5.0)
Create an event when a file is opened with the intent to be executed. See NOTES for additional details.

FAN_ATTRIB (since Linux 5.1)
Create an event when the metadata for a file or directory has changed. An fanotify group that identifies filesystem objects by file handles is required.

FAN_CREATE (since Linux 5.1)
Create an event when a file or directory has been created in a marked parent directory. An fanotify group that identifies filesystem objects by file handles is required.

FAN_DELETE (since Linux 5.1)
Create an event when a file or directory has been deleted in a marked parent directory. An fanotify group that identifies filesystem objects by file handles is required.

FAN_DELETE_SELF (since Linux 5.1)
Create an event when a marked file or directory itself is deleted. An fanotify group that identifies filesystem objects by file handles is required.

FAN_FS_ERROR (since Linux 5.16)
Create an event when a filesystem error leading to inconsistent filesystem metadata is detected. An additional information record of type **FAN_EVENT_INFO_TYPE_ERROR** is returned for each event in the read buffer. An fanotify group that identifies filesystem objects by file handles is required.

Events of such type are dependent on support from the underlying filesystem. At the time of writing, only the **ext4** filesystem reports **FAN_FS_ERROR** events.

See *fanotify*(7) for additional details.

FAN_MOVED_FROM (since Linux 5.1)
Create an event when a file or directory has been moved from a marked parent directory. An fanotify group that identifies filesystem objects by file handles is required.

FAN_MOVED_TO (since Linux 5.1)
Create an event when a file or directory has been moved to a marked parent directory. An fanotify group that identifies filesystem objects by file handles is required.

FAN_RENAME (since Linux 5.17)
This event contains the same information provided by events **FAN_MOVED_FROM** and **FAN_MOVED_TO**, however is represented by a single event with up to two information records. An fanotify group that identifies filesystem objects by file handles is required. If the filesystem object to be marked is not a directory, the error **ENOTDIR** shall be raised.

FAN_MOVE_SELF (since Linux 5.1)
Create an event when a marked file or directory itself has been moved. An fanotify group that identifies filesystem objects by file handles is required.

FAN_OPEN_PERM
Create an event when a permission to open a file or directory is requested. An fanotify file descriptor created with **FAN_CLASS_PRE_CONTENT** or **FAN_CLASS_CONTENT** is required.

FAN_OPEN_EXEC_PERM (since Linux 5.0)
Create an event when a permission to open a file for execution is requested. An fanotify file descriptor created with **FAN_CLASS_PRE_CONTENT** or **FAN_CLASS_CONTENT** is required. See NOTES for additional details.

FAN_ACCESS_PERM
Create an event when a permission to read a file or directory is requested. An fanotify file descriptor created with **FAN_CLASS_PRE_CONTENT** or **FAN_CLASS_CONTENT** is required.

FAN_ONDIR
Create events for directories—for example, when *opendir*(3), *readdir*(3) (but see BUGS), and *closedir*(3) are called. Without this flag, events are created only for files. In the context of directory entry events, such as **FAN_CREATE**, **FAN_DELETE**, **FAN_MOVED_FROM**, and **FAN_MOVED_TO**, specifying the flag **FAN_ONDIR** is required in order to create events when subdirectory entries are modified (i.e., *mkdir*(2)/ *rmdir*(2)).

FAN_EVENT_ON_CHILD
Events for the immediate children of marked directories shall be created. The flag has no effect when marking mounts and filesystems. Note that events are not generated for children of the subdirectories of marked directories. More specifically, the directory entry modification events **FAN_CREATE**, **FAN_DELETE**, **FAN_MOVED_FROM**, and **FAN_MOVED_TO** are not generated for any entry modifications performed inside subdirectories of marked directories. Note that the events **FAN_DELETE_SELF** and **FAN_MOVE_SELF** are not generated for children of marked directories. To monitor complete directory trees it is necessary to mark the relevant mount or filesystem.

The following composed values are defined:

FAN_CLOSE
A file is closed (**FAN_CLOSE_WRITE**|**FAN_CLOSE_NOWRITE**).

FAN_MOVE
A file or directory has been moved (**FAN_MOVED_FROM**|**FAN_MOVED_TO**).

The filesystem object to be marked is determined by the file descriptor *dirfd* and the pathname specified in *pathname*:

- If *pathname* is NULL, *dirfd* defines the filesystem object to be marked.
- If *pathname* is NULL, and *dirfd* takes the special value **AT_FDCWD**, the current working directory is to be marked.
- If *pathname* is absolute, it defines the filesystem object to be marked, and *dirfd* is ignored.
- If *pathname* is relative, and *dirfd* does not have the value **AT_FDCWD**, then the filesystem object to be marked is determined by interpreting *pathname* relative the directory referred to by *dirfd*.
- If *pathname* is relative, and *dirfd* has the value **AT_FDCWD**, then the filesystem object to be marked is determined by interpreting *pathname* relative to the current working directory. (See *openat*(2) for an explanation of why the *dirfd* argument is useful.)

RETURN VALUE
On success, **fanotify_mark**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS
EBADF
An invalid file descriptor was passed in *fanotify_fd*.

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EEXIST
The filesystem object indicated by *dirfd* and *pathname* has a mark that was updated without the **FAN_MARK_EVICTABLE** flag, and the user attempted to update the mark with **FAN_MARK_EVICTABLE** flag.

EEXIST
The filesystem object indicated by *dirfd* and *pathname* has a mark that was updated with the **FAN_MARK_IGNORE** flag, and the user attempted to update the mark with **FAN_MARK_IGNORED_MASK** flag.

EEXIST
The filesystem object indicated by *dirfd* and *pathname* has a mark that was updated with the **FAN_MARK_IGNORE** and **FAN_MARK_IGNORED_SURV_MODIFY** flags, and the user attempted to update the mark only with **FAN_MARK_IGNORE** flag.

EINVAL
An invalid value was passed in *flags* or *mask*, or *fanotify_fd* was not an fanotify file descriptor.

EINVAL
The fanotify file descriptor was opened with **FAN_CLASS_NOTIF** or the fanotify group identifies filesystem objects by file handles and mask contains a flag for permission events (**FAN_OPEN_PERM** or **FAN_ACCESS_PERM**).

EINVAL
The group was initialized without **FAN_REPORT_FID** but one or more event types specified in the *mask* require it.

EINVAL
flags contains **FAN_MARK_IGNORE**, and either **FAN_MARK_MOUNT** or **FAN_MARK_FILESYSTEM**, but does not contain **FAN_MARK_IGNORED_SURV_MODIFY**.

EISDIR
flags contains **FAN_MARK_IGNORE**, but does not contain **FAN_MARK_IGNORED_SURV_MODIFY**, and *dirfd* and *pathname* specify a directory.

ENODEV
The filesystem object indicated by *dirfd* and *pathname* is not associated with a filesystem that supports *fsid* (e.g., *fuse*(4)). *tmpfs*(5) did not support *fsid* prior to Linux 5.13. This error can be returned only with an fanotify group that identifies filesystem objects by file handles.

ENOENT
The filesystem object indicated by *dirfd* and *pathname* does not exist. This error also occurs when trying to remove a mark from an object which is not marked.

ENOMEM
The necessary memory could not be allocated.

ENOSPC
The number of marks for this user exceeds the limit and the **FAN_UNLIMITED_MARKS** flag was not specified when the fanotify file descriptor was created with *fanotify_init*(2). See *fanotify*(7) for details about this limit.

ENOSYS
This kernel does not implement **fanotify_mark**(). The fanotify API is available only if the kernel was configured with **CONFIG_FANOTIFY**.

ENOTDIR
flags contains **FAN_MARK_ONLYDIR**, and *dirfd* and *pathname* do not specify a directory.

ENOTDIR
mask contains **FAN_RENAME**, and *dirfd* and *pathname* do not specify a directory.

ENOTDIR
flags contains **FAN_MARK_IGNORE**, or the fanotify group was initialized with flag **FAN_REPORT_TARGET_FID**, and *mask* contains directory entry modification events (e.g., **FAN_CREATE**, **FAN_DELETE**), or directory event flags (e.g., **FAN_ONDIR**, **FAN_EVENT_ON_CHILD**), and *dirfd* and *pathname* do not specify a directory.

EOPNOTSUPP
The object indicated by *pathname* is associated with a filesystem that does not support the encoding of file handles. This error can be returned only with an fanotify group that identifies

filesystem objects by file handles. Calling *name_to_handle_at*(2) with the flag **AT_HANDLE_FID** (since Linux 6.5) can be used as a test to check if a filesystem supports reporting events with file handles.

EPERM
The operation is not permitted because the caller lacks a required capability.

EXDEV
The filesystem object indicated by *pathname* resides within a filesystem subvolume (e.g., *btrfs*(5)) which uses a different *fsid* than its root superblock. This error can be returned only with an fanotify group that identifies filesystem objects by file handles.

STANDARDS
Linux.

HISTORY
Linux 2.6.37.

NOTES
FAN_OPEN_EXEC and FAN_OPEN_EXEC_PERM
When using either **FAN_OPEN_EXEC** or **FAN_OPEN_EXEC_PERM** within the *mask*, events of these types will be returned only when the direct execution of a program occurs. More specifically, this means that events of these types will be generated for files that are opened using *execve*(2), *execveat*(2), or *uselib*(2). Events of these types will not be raised in the situation where an interpreter is passed (or reads) a file for interpretation.

Additionally, if a mark has also been placed on the Linux dynamic linker, a user should also expect to receive an event for it when an ELF object has been successfully opened using *execve*(2) or *execveat*(2).

For example, if the following ELF binary were to be invoked and a **FAN_OPEN_EXEC** mark has been placed on /:

```
$ /bin/echo foo
```

The listening application in this case would receive **FAN_OPEN_EXEC** events for both the ELF binary and interpreter, respectively:

```
/bin/echo
/lib64/ld-linux-x86-64.so.2
```

BUGS
The following bugs were present in before Linux 3.16:

- If *flags* contains **FAN_MARK_FLUSH**, *dirfd*, and *pathname* must specify a valid filesystem object, even though this object is not used.
- *readdir*(2) does not generate a **FAN_ACCESS** event.
- If **fanotify_mark**() is called with **FAN_MARK_FLUSH**, *flags* is not checked for invalid values.

SEE ALSO
fanotify_init(2), *fanotify*(7)

NAME

fcntl − manipulate file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fcntl.h>

int fcntl(int *fd***, int** *op***, ... /*** *arg* ***/);**

DESCRIPTION

fcntl() performs one of the operations described below on the open file descriptor *fd*. The operation is determined by *op*.

fcntl() can take an optional third argument. Whether or not this argument is required is determined by *op*. The required argument type is indicated in parentheses after each *op* name (in most cases, the required type is *int*, and we identify the argument using the name *arg*), or *void* is specified if the argument is not required.

Certain of the operations below are supported only since a particular Linux kernel version. The preferred method of checking whether the host kernel supports a particular operation is to invoke **fcntl**() with the desired *op* value and then test whether the call failed with **EINVAL**, indicating that the kernel does not recognize this value.

Duplicating a file descriptor

F_DUPFD (*int*)

Duplicate the file descriptor *fd* using the lowest-numbered available file descriptor greater than or equal to *arg*. This is different from *dup2*(2), which uses exactly the file descriptor specified.

On success, the new file descriptor is returned.

See *dup*(2) for further details.

F_DUPFD_CLOEXEC (*int*; since Linux 2.6.24)

As for **F_DUPFD**, but additionally set the close-on-exec flag for the duplicate file descriptor. Specifying this flag permits a program to avoid an additional **fcntl**() **F_SETFD** operation to set the **FD_CLOEXEC** flag. For an explanation of why this flag is useful, see the description of **O_CLOEXEC** in *open*(2).

File descriptor flags

The following operations manipulate the flags associated with a file descriptor. Currently, only one such flag is defined: **FD_CLOEXEC**, the close-on-exec flag. If the **FD_CLOEXEC** bit is set, the file descriptor will automatically be closed during a successful *execve*(2). (If the *execve*(2) fails, the file descriptor is left open.) If the **FD_CLOEXEC** bit is not set, the file descriptor will remain open across an *execve*(2).

F_GETFD (*void*)

Return (as the function result) the file descriptor flags; *arg* is ignored.

F_SETFD (*int*)

Set the file descriptor flags to the value specified by *arg*.

In multithreaded programs, using **fcntl**() **F_SETFD** to set the close-on-exec flag at the same time as another thread performs a *fork*(2) plus *execve*(2) is vulnerable to a race condition that may unintentionally leak the file descriptor to the program executed in the child process. See the discussion of the **O_CLOEXEC** flag in *open*(2) for details and a remedy to the problem.

File status flags

Each open file description has certain associated status flags, initialized by *open*(2) and possibly modified by **fcntl**(). Duplicated file descriptors (made with *dup*(2), *fcntl*(F_DUPFD), *fork*(2), etc.) refer to the same open file description, and thus share the same file status flags.

The file status flags and their semantics are described in *open*(2).

F_GETFL (*void*)

Return (as the function result) the file access mode and the file status flags; *arg* is ignored.

F_SETFL (*int*)
Set the file status flags to the value specified by *arg*. File access mode (**O_RDONLY**, **O_WRONLY**, **O_RDWR**) and file creation flags (i.e., **O_CREAT**, **O_EXCL**, **O_NOCTTY**, **O_TRUNC**) in *arg* are ignored. On Linux, this operation can change only the **O_APPEND**, **O_ASYNC**, **O_DIRECT**, **O_NOATIME**, and **O_NONBLOCK** flags. It is not possible to change the **O_DSYNC** and **O_SYNC** flags; see BUGS, below.

Advisory record locking
Linux implements traditional ("process-associated") UNIX record locks, as standardized by POSIX. For a Linux-specific alternative with better semantics, see the discussion of open file description locks below.

F_SETLK, **F_SETLKW**, and **F_GETLK** are used to acquire, release, and test for the existence of record locks (also known as byte-range, file-segment, or file-region locks). The third argument, *lock*, is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

The *l_whence*, *l_start*, and *l_len* fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

l_start is the starting offset for the lock, and is interpreted relative to either: the start of the file (if *l_whence* is **SEEK_SET**); the current file offset (if *l_whence* is **SEEK_CUR**); or the end of the file (if *l_whence* is **SEEK_END**). In the final two cases, *l_start* can be a negative number provided the offset does not lie before the start of the file.

l_len specifies the number of bytes to be locked. If *l_len* is positive, then the range to be locked covers bytes *l_start* up to and including *l_start*+*l_len*−1. Specifying 0 for *l_len* has the special meaning: lock all bytes starting at the location specified by *l_whence* and *l_start* through to the end of file, no matter how large the file grows.

POSIX.1-2001 allows (but does not require) an implementation to support a negative *l_len* value; if *l_len* is negative, the interval described by *lock* covers bytes *l_start*+*l_len* up to and including *l_start*−1. This is supported since Linux 2.4.21 and Linux 2.5.49.

The *l_type* field can be used to place a read (**F_RDLCK**) or a write (**F_WRLCK**) lock on a file. Any number of processes may hold a read lock (shared lock) on a file region, but only one process may hold a write lock (exclusive lock). An exclusive lock excludes all other locks, both shared and exclusive. A single process can hold only one type of lock on a file region; if a new lock is applied to an already-locked region, then the existing lock is converted to the new lock type. (Such conversions may involve splitting, shrinking, or coalescing with an existing lock if the byte range specified by the new lock does not precisely coincide with the range of the existing lock.)

F_SETLK (*struct flock* *)
Acquire a lock (when *l_type* is **F_RDLCK** or **F_WRLCK**) or release a lock (when *l_type* is **F_UNLCK**) on the bytes specified by the *l_whence*, *l_start*, and *l_len* fields of *lock*. If a conflicting lock is held by another process, this call returns −1 and sets *errno* to **EACCES** or **EAGAIN**. (The error returned in this case differs across implementations, so POSIX requires a portable application to check for both errors.)

F_SETLKW (*struct flock* *)
As for **F_SETLK**, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and (after the signal handler has returned) returns immediately (with return value −1 and *errno* set to **EINTR**; see

signal(7)).

F_GETLK (*struct flock* *)
On input to this call, *lock* describes a lock we would like to place on the file. If the lock could be placed, **fcntl**() does not actually place it, but returns **F_UNLCK** in the *l_type* field of *lock* and leaves the other fields of the structure unchanged.

If one or more incompatible locks would prevent this lock being placed, then **fcntl**() returns details about one of those locks in the *l_type*, *l_whence*, *l_start*, and *l_len* fields of *lock*. If the conflicting lock is a traditional (process-associated) record lock, then the *l_pid* field is set to the PID of the process holding that lock. If the conflicting lock is an open file description lock, then *l_pid* is set to −1. Note that the returned information may already be out of date by the time the caller inspects it.

In order to place a read lock, *fd* must be open for reading. In order to place a write lock, *fd* must be open for writing. To place both types of lock, open a file read-write.

When placing locks with **F_SETLKW**, the kernel detects *deadlocks*, whereby two or more processes have their lock requests mutually blocked by locks held by the other processes. For example, suppose process A holds a write lock on byte 100 of a file, and process B holds a write lock on byte 200. If each process then attempts to lock the byte already locked by the other process using **F_SETLKW**, then, without deadlock detection, both processes would remain blocked indefinitely. When the kernel detects such deadlocks, it causes one of the blocking lock requests to immediately fail with the error **EDEADLK**; an application that encounters such an error should release some of its locks to allow other applications to proceed before attempting regain the locks that it requires. Circular deadlocks involving more than two processes are also detected. Note, however, that there are limitations to the kernel's deadlock-detection algorithm; see BUGS.

As well as being removed by an explicit **F_UNLCK**, record locks are automatically released when the process terminates.

Record locks are not inherited by a child created via *fork*(2), but are preserved across an *execve*(2).

Because of the buffering performed by the *stdio*(3) library, the use of record locking with routines in that package should be avoided; use *read*(2) and *write*(2) instead.

The record locks described above are associated with the process (unlike the open file description locks described below). This has some unfortunate consequences:

- If a process closes *any* file descriptor referring to a file, then all of the process's locks on that file are released, regardless of the file descriptor(s) on which the locks were obtained. This is bad: it means that a process can lose its locks on a file such as */etc/passwd* or */etc/mtab* when for some reason a library function decides to open, read, and close the same file.
- The threads in a process share locks. In other words, a multithreaded program can't use record locking to ensure that threads don't simultaneously access the same region of a file.

Open file description locks solve both of these problems.

Open file description locks (non-POSIX)
Open file description locks are advisory byte-range locks whose operation is in most respects identical to the traditional record locks described above. This lock type is Linux-specific, and available since Linux 3.15. (There is a proposal with the Austin Group to include this lock type in the next revision of POSIX.1.) For an explanation of open file descriptions, see *open*(2).

The principal difference between the two lock types is that whereas traditional record locks are associated with a process, open file description locks are associated with the open file description on which they are acquired, much like locks acquired with *flock*(2). Consequently (and unlike traditional advisory record locks), open file description locks are inherited across *fork*(2) (and *clone*(2) with **CLONE_FILES**), and are only automatically released on the last close of the open file description, instead of being released on any close of the file.

Conflicting lock combinations (i.e., a read lock and a write lock or two write locks) where one lock is an open file description lock and the other is a traditional record lock conflict even when they are acquired by the same process on the same file descriptor.

Open file description locks placed via the same open file description (i.e., via the same file descriptor, or via a duplicate of the file descriptor created by *fork*(2), *dup*(2), **fcntl**() **F_DUPFD**, and so on) are

always compatible: if a new lock is placed on an already locked region, then the existing lock is converted to the new lock type. (Such conversions may result in splitting, shrinking, or coalescing with an existing lock as discussed above.)

On the other hand, open file description locks may conflict with each other when they are acquired via different open file descriptions. Thus, the threads in a multithreaded program can use open file description locks to synchronize access to a file region by having each thread perform its own *open*(2) on the file and applying locks via the resulting file descriptor.

As with traditional advisory locks, the third argument to **fcntl**(), *lock*, is a pointer to an *flock* structure. By contrast with traditional record locks, the *l_pid* field of that structure must be set to zero when using the operations described below.

The operations for working with open file description locks are analogous to those used with traditional locks:

F_OFD_SETLK (*struct flock* *)
: Acquire an open file description lock (when *l_type* is **F_RDLCK** or **F_WRLCK**) or release an open file description lock (when *l_type* is **F_UNLCK**) on the bytes specified by the *l_whence*, *l_start*, and *l_len* fields of *lock*. If a conflicting lock is held by another process, this call returns −1 and sets *errno* to **EAGAIN**.

F_OFD_SETLKW (*struct flock* *)
: As for **F_OFD_SETLK**, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and (after the signal handler has returned) returns immediately (with return value −1 and *errno* set to **EINTR**; see *signal*(7)).

F_OFD_GETLK (*struct flock* *)
: On input to this call, *lock* describes an open file description lock we would like to place on the file. If the lock could be placed, **fcntl**() does not actually place it, but returns **F_UNLCK** in the *l_type* field of *lock* and leaves the other fields of the structure unchanged. If one or more incompatible locks would prevent this lock being placed, then details about one of these locks are returned via *lock*, as described above for **F_GETLK**.

In the current implementation, no deadlock detection is performed for open file description locks. (This contrasts with process-associated record locks, for which the kernel does perform deadlock detection.)

Mandatory locking

Warning: the Linux implementation of mandatory locking is unreliable. See BUGS below. Because of these bugs, and the fact that the feature is believed to be little used, since Linux 4.5, mandatory locking has been made an optional feature, governed by a configuration option (**CONFIG_MANDATORY_FILE_LOCKING**). This feature is no longer supported at all in Linux 5.15 and above.

By default, both traditional (process-associated) and open file description record locks are advisory. Advisory locks are not enforced and are useful only between cooperating processes.

Both lock types can also be mandatory. Mandatory locks are enforced for all processes. If a process tries to perform an incompatible access (e.g., *read*(2) or *write*(2)) on a file region that has an incompatible mandatory lock, then the result depends upon whether the **O_NONBLOCK** flag is enabled for its open file description. If the **O_NONBLOCK** flag is not enabled, then the system call is blocked until the lock is removed or converted to a mode that is compatible with the access. If the **O_NONBLOCK** flag is enabled, then the system call fails with the error **EAGAIN**.

To make use of mandatory locks, mandatory locking must be enabled both on the filesystem that contains the file to be locked, and on the file itself. Mandatory locking is enabled on a filesystem using the "−o mand" option to *mount*(8), or the **MS_MANDLOCK** flag for *mount*(2). Mandatory locking is enabled on a file by disabling group execute permission on the file and enabling the set-group-ID permission bit (see *chmod*(1) and *chmod*(2)).

Mandatory locking is not specified by POSIX. Some other systems also support mandatory locking, although the details of how to enable it vary across systems.

Lost locks

When an advisory lock is obtained on a networked filesystem such as NFS it is possible that the lock might get lost. This may happen due to administrative action on the server, or due to a network partition (i.e., loss of network connectivity with the server) which lasts long enough for the server to assume that the client is no longer functioning.

When the filesystem determines that a lock has been lost, future *read*(2) or *write*(2) requests may fail with the error **EIO**. This error will persist until the lock is removed or the file descriptor is closed. Since Linux 3.12, this happens at least for NFSv4 (including all minor versions).

Some versions of UNIX send a signal (**SIGLOST**) in this circumstance. Linux does not define this signal, and does not provide any asynchronous notification of lost locks.

Managing signals

F_GETOWN, **F_SETOWN**, **F_GETOWN_EX**, **F_SETOWN_EX**, **F_GETSIG**, and **F_SETSIG** are used to manage I/O availability signals:

F_GETOWN (*void*)

Return (as the function result) the process ID or process group ID currently receiving **SIGIO** and **SIGURG** signals for events on file descriptor *fd*. Process IDs are returned as positive values; process group IDs are returned as negative values (but see BUGS below). *arg* is ignored.

F_SETOWN (*int*)

Set the process ID or process group ID that will receive **SIGIO** and **SIGURG** signals for events on the file descriptor *fd*. The target process or process group ID is specified in *arg*. A process ID is specified as a positive value; a process group ID is specified as a negative value. Most commonly, the calling process specifies itself as the owner (that is, *arg* is specified as *getpid*(2)).

As well as setting the file descriptor owner, one must also enable generation of signals on the file descriptor. This is done by using the **fcntl**() **F_SETFL** operation to set the **O_ASYNC** file status flag on the file descriptor. Subsequently, a **SIGIO** signal is sent whenever input or output becomes possible on the file descriptor. The **fcntl**() **F_SETSIG** operation can be used to obtain delivery of a signal other than **SIGIO**.

Sending a signal to the owner process (group) specified by **F_SETOWN** is subject to the same permissions checks as are described for *kill*(2), where the sending process is the one that employs **F_SETOWN** (but see BUGS below). If this permission check fails, then the signal is silently discarded. *Note*: The **F_SETOWN** operation records the caller's credentials at the time of the **fcntl**() call, and it is these saved credentials that are used for the permission checks.

If the file descriptor *fd* refers to a socket, **F_SETOWN** also selects the recipient of **SIGURG** signals that are delivered when out-of-band data arrives on that socket. (**SIGURG** is sent in any situation where *select*(2) would report the socket as having an "exceptional condition".)

The following was true in Linux 2.6.x up to and including Linux 2.6.11:

> If a nonzero value is given to **F_SETSIG** in a multithreaded process running with a threading library that supports thread groups (e.g., NPTL), then a positive value given to **F_SETOWN** has a different meaning: instead of being a process ID identifying a whole process, it is a thread ID identifying a specific thread within a process. Consequently, it may be necessary to pass **F_SETOWN** the result of *gettid*(2) instead of *getpid*(2) to get sensible results when **F_SETSIG** is used. (In current Linux threading implementations, a main thread's thread ID is the same as its process ID. This means that a single-threaded program can equally use *gettid*(2) or *getpid*(2) in this scenario.) Note, however, that the statements in this paragraph do not apply to the **SIGURG** signal generated for out-of-band data on a socket: this signal is always sent to either a process or a process group, depending on the value given to **F_SETOWN**.

The above behavior was accidentally dropped in Linux 2.6.12, and won't be restored. From Linux 2.6.32 onward, use **F_SETOWN_EX** to target **SIGIO** and **SIGURG** signals at a particular thread.

F_GETOWN_EX (*struct f_owner_ex* *) (since Linux 2.6.32)
Return the current file descriptor owner settings as defined by a previous **F_SETOWN_EX** operation. The information is returned in the structure pointed to by *arg*, which has the following form:

```
struct f_owner_ex {
    int   type;
    pid_t pid;
};
```

The *type* field will have one of the values **F_OWNER_TID**, **F_OWNER_PID**, or **F_OWNER_PGRP**. The *pid* field is a positive integer representing a thread ID, process ID, or process group ID. See **F_SETOWN_EX** for more details.

F_SETOWN_EX (*struct f_owner_ex* *) (since Linux 2.6.32)
This operation performs a similar task to **F_SETOWN**. It allows the caller to direct I/O availability signals to a specific thread, process, or process group. The caller specifies the target of signals via *arg*, which is a pointer to a *f_owner_ex* structure. The *type* field has one of the following values, which define how *pid* is interpreted:

F_OWNER_TID
Send the signal to the thread whose thread ID (the value returned by a call to *clone*(2) or *gettid*(2)) is specified in *pid*.

F_OWNER_PID
Send the signal to the process whose ID is specified in *pid*.

F_OWNER_PGRP
Send the signal to the process group whose ID is specified in *pid*. (Note that, unlike with **F_SETOWN**, a process group ID is specified as a positive value here.)

F_GETSIG (*void*)
Return (as the function result) the signal sent when input or output becomes possible. A value of zero means **SIGIO** is sent. Any other value (including **SIGIO**) is the signal sent instead, and in this case additional info is available to the signal handler if installed with **SA_SIGINFO**. *arg* is ignored.

F_SETSIG (*int*)
Set the signal sent when input or output becomes possible to the value given in *arg*. A value of zero means to send the default **SIGIO** signal. Any other value (including **SIGIO**) is the signal to send instead, and in this case additional info is available to the signal handler if installed with **SA_SIGINFO**.

By using **F_SETSIG** with a nonzero value, and setting **SA_SIGINFO** for the signal handler (see *sigaction*(2)), extra information about I/O events is passed to the handler in a *siginfo_t* structure. If the *si_code* field indicates the source is **SI_SIGIO**, the *si_fd* field gives the file descriptor associated with the event. Otherwise, there is no indication which file descriptors are pending, and you should use the usual mechanisms (**select**(2), *poll*(2), *read*(2) with **O_NONBLOCK** set etc.) to determine which file descriptors are available for I/O.

Note that the file descriptor provided in *si_fd* is the one that was specified during the **F_SETSIG** operation. This can lead to an unusual corner case. If the file descriptor is duplicated (**dup**(2) or similar), and the original file descriptor is closed, then I/O events will continue to be generated, but the *si_fd* field will contain the number of the now closed file descriptor.

By selecting a real time signal (value >= **SIGRTMIN**), multiple I/O events may be queued using the same signal numbers. (Queuing is dependent on available memory.) Extra information is available if **SA_SIGINFO** is set for the signal handler, as above.

Note that Linux imposes a limit on the number of real-time signals that may be queued to a process (see *getrlimit*(2) and *signal*(7)) and if this limit is reached, then the kernel reverts to delivering **SIGIO**, and this signal is delivered to the entire process rather than to a specific thread.

Using these mechanisms, a program can implement fully asynchronous I/O without using *select*(2) or *poll*(2) most of the time.

The use of **O_ASYNC** is specific to BSD and Linux. The only use of **F_GETOWN** and **F_SETOWN** specified in POSIX.1 is in conjunction with the use of the **SIGURG** signal on sockets. (POSIX does not specify the **SIGIO** signal.) **F_GETOWN_EX**, **F_SETOWN_EX**, **F_GETSIG**, and **F_SETSIG** are Linux-specific. POSIX has asynchronous I/O and the *aio_sigevent* structure to achieve similar things; these are also available in Linux as part of the GNU C Library (glibc).

Leases

F_SETLEASE and **F_GETLEASE** (Linux 2.4 onward) are used to establish a new lease, and retrieve the current lease, on the open file description referred to by the file descriptor *fd*. A file lease provides a mechanism whereby the process holding the lease (the "lease holder") is notified (via delivery of a signal) when a process (the "lease breaker") tries to *open*(2) or *truncate*(2) the file referred to by that file descriptor.

F_SETLEASE (*int*)

Set or remove a file lease according to which of the following values is specified in the integer *arg*:

F_RDLCK

Take out a read lease. This will cause the calling process to be notified when the file is opened for writing or is truncated. A read lease can be placed only on a file descriptor that is opened read-only.

F_WRLCK

Take out a write lease. This will cause the caller to be notified when the file is opened for reading or writing or is truncated. A write lease may be placed on a file only if there are no other open file descriptors for the file.

F_UNLCK

Remove our lease from the file.

Leases are associated with an open file description (see *open*(2)). This means that duplicate file descriptors (created by, for example, *fork*(2) or *dup*(2)) refer to the same lease, and this lease may be modified or released using any of these descriptors. Furthermore, the lease is released by either an explicit **F_UNLCK** operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

Leases may be taken out only on regular files. An unprivileged process may take out a lease only on a file whose UID (owner) matches the filesystem UID of the process. A process with the **CAP_LEASE** capability may take out leases on arbitrary files.

F_GETLEASE (*void*)

Indicates what type of lease is associated with the file descriptor *fd* by returning either **F_RDLCK**, **F_WRLCK**, or **F_UNLCK**, indicating, respectively, a read lease , a write lease, or no lease. *arg* is ignored.

When a process (the "lease breaker") performs an *open*(2) or *truncate*(2) that conflicts with a lease established via **F_SETLEASE**, the system call is blocked by the kernel and the kernel notifies the lease holder by sending it a signal (**SIGIO** by default). The lease holder should respond to receipt of this signal by doing whatever cleanup is required in preparation for the file to be accessed by another process (e.g., flushing cached buffers) and then either remove or downgrade its lease. A lease is removed by performing an **F_SETLEASE** operation specifying *arg* as **F_UNLCK**. If the lease holder currently holds a write lease on the file, and the lease breaker is opening the file for reading, then it is sufficient for the lease holder to downgrade the lease to a read lease. This is done by performing an **F_SETLEASE** operation specifying *arg* as **F_RDLCK**.

If the lease holder fails to downgrade or remove the lease within the number of seconds specified in */proc/sys/fs/lease-break-time*, then the kernel forcibly removes or downgrades the lease holder's lease.

Once a lease break has been initiated, **F_GETLEASE** returns the target lease type (either **F_RDLCK** or **F_UNLCK**, depending on what would be compatible with the lease breaker) until the lease holder voluntarily downgrades or removes the lease or the kernel forcibly does so after the lease break timer expires.

Once the lease has been voluntarily or forcibly removed or downgraded, and assuming the lease breaker has not unblocked its system call, the kernel permits the lease breaker's system call to proceed.

If the lease breaker's blocked *open*(2) or *truncate*(2) is interrupted by a signal handler, then the system call fails with the error **EINTR**, but the other steps still occur as described above. If the lease breaker is killed by a signal while blocked in *open*(2) or *truncate*(2), then the other steps still occur as described above. If the lease breaker specifies the **O_NONBLOCK** flag when calling *open*(2), then the call immediately fails with the error **EWOULDBLOCK**, but the other steps still occur as described above.

The default signal used to notify the lease holder is **SIGIO**, but this can be changed using the **F_SETSIG** operation to **fcntl**(). If a **F_SETSIG** operation is performed (even one specifying **SIGIO**), and the signal handler is established using **SA_SIGINFO**, then the handler will receive a *siginfo_t* structure as its second argument, and the *si_fd* field of this argument will hold the file descriptor of the leased file that has been accessed by another process. (This is useful if the caller holds leases against multiple files.)

File and directory change notification (dnotify)

F_NOTIFY (*int*)

(Linux 2.4 onward) Provide notification when the directory referred to by *fd* or any of the files that it contains is changed. The events to be notified are specified in *arg*, which is a bit mask specified by ORing together zero or more of the following bits:

DN_ACCESS
A file was accessed (**read**(2), *pread*(2), *readv*(2), and similar)

DN_MODIFY
A file was modified (**write**(2), *pwrite*(2), *writev*(2), *truncate*(2), *ftruncate*(2), and similar).

DN_CREATE
A file was created (**open**(2), *creat*(2), *mknod*(2), *mkdir*(2), *link*(2), *symlink*(2), *rename*(2) into this directory).

DN_DELETE
A file was unlinked (**unlink**(2), *rename*(2) to another directory, *rmdir*(2)).

DN_RENAME
A file was renamed within this directory (**rename**(2)).

DN_ATTRIB
The attributes of a file were changed (**chown**(2), *chmod*(2), *utime*(2), *utimensat*(2), and similar).

(In order to obtain these definitions, the **_GNU_SOURCE** feature test macro must be defined before including *any* header files.)

Directory notifications are normally "one-shot", and the application must reregister to receive further notifications. Alternatively, if **DN_MULTISHOT** is included in *arg*, then notification will remain in effect until explicitly removed.

A series of **F_NOTIFY** requests is cumulative, with the events in *arg* being added to the set already monitored. To disable notification of all events, make an **F_NOTIFY** call specifying *arg* as 0.

Notification occurs via delivery of a signal. The default signal is **SIGIO**, but this can be changed using the **F_SETSIG** operation to **fcntl**(). (Note that **SIGIO** is one of the nonqueuing standard signals; switching to the use of a real-time signal means that multiple notifications can be queued to the process.) In the latter case, the signal handler receives a *siginfo_t* structure as its second argument (if the handler was established using **SA_SIGINFO**) and the *si_fd* field of this structure contains the file descriptor which generated the notification (useful when establishing notification on multiple directories).

Especially when using **DN_MULTISHOT**, a real time signal should be used for notification, so that multiple notifications can be queued.

NOTE: New applications should use the *inotify* interface (available since Linux 2.6.13), which provides a much superior interface for obtaining notifications of filesystem events. See *inotify*(7).

Changing the capacity of a pipe

F_SETPIPE_SZ (*int*; since Linux 2.6.35)

Change the capacity of the pipe referred to by *fd* to be at least *arg* bytes. An unprivileged process can adjust the pipe capacity to any value between the system page size and the limit

defined in */proc/sys/fs/pipe-max-size* (see *proc*(5)). Attempts to set the pipe capacity below the page size are silently rounded up to the page size. Attempts by an unprivileged process to set the pipe capacity above the limit in */proc/sys/fs/pipe-max-size* yield the error **EPERM**; a privileged process (**CAP_SYS_RESOURCE**) can override the limit.

When allocating the buffer for the pipe, the kernel may use a capacity larger than *arg*, if that is convenient for the implementation. (In the current implementation, the allocation is the next higher power-of-two page-size multiple of the requested size.) The actual capacity (in bytes) that is set is returned as the function result.

Attempting to set the pipe capacity smaller than the amount of buffer space currently used to store data produces the error **EBUSY**.

Note that because of the way the pages of the pipe buffer are employed when data is written to the pipe, the number of bytes that can be written may be less than the nominal size, depending on the size of the writes.

F_GETPIPE_SZ (*void*; since Linux 2.6.35)
Return (as the function result) the capacity of the pipe referred to by *fd*.

File Sealing

File seals limit the set of allowed operations on a given file. For each seal that is set on a file, a specific set of operations will fail with **EPERM** on this file from now on. The file is said to be sealed. The default set of seals depends on the type of the underlying file and filesystem. For an overview of file sealing, a discussion of its purpose, and some code examples, see *memfd_create*(2).

Currently, file seals can be applied only to a file descriptor returned by *memfd_create*(2) (if the **MFD_ALLOW_SEALING** was employed). On other filesystems, all **fcntl**() operations that operate on seals will return **EINVAL**.

Seals are a property of an inode. Thus, all open file descriptors referring to the same inode share the same set of seals. Furthermore, seals can never be removed, only added.

F_ADD_SEALS (*int*; since Linux 3.17)
Add the seals given in the bit-mask argument *arg* to the set of seals of the inode referred to by the file descriptor *fd*. Seals cannot be removed again. Once this call succeeds, the seals are enforced by the kernel immediately. If the current set of seals includes **F_SEAL_SEAL** (see below), then this call will be rejected with **EPERM**. Adding a seal that is already set is a no-op, in case **F_SEAL_SEAL** is not set already. In order to place a seal, the file descriptor *fd* must be writable.

F_GET_SEALS (*void*; since Linux 3.17)
Return (as the function result) the current set of seals of the inode referred to by *fd*. If no seals are set, 0 is returned. If the file does not support sealing, −1 is returned and *errno* is set to **EINVAL**.

The following seals are available:

F_SEAL_SEAL
If this seal is set, any further call to **fcntl**() with **F_ADD_SEALS** fails with the error **EPERM**. Therefore, this seal prevents any modifications to the set of seals itself. If the initial set of seals of a file includes **F_SEAL_SEAL**, then this effectively causes the set of seals to be constant and locked.

F_SEAL_SHRINK
If this seal is set, the file in question cannot be reduced in size. This affects *open*(2) with the **O_TRUNC** flag as well as *truncate*(2) and *ftruncate*(2). Those calls fail with **EPERM** if you try to shrink the file in question. Increasing the file size is still possible.

F_SEAL_GROW
If this seal is set, the size of the file in question cannot be increased. This affects *write*(2) beyond the end of the file, *truncate*(2), *ftruncate*(2), and *fallocate*(2). These calls fail with **EPERM** if you use them to increase the file size. If you keep the size or shrink it, those calls still work as expected.

F_SEAL_WRITE
If this seal is set, you cannot modify the contents of the file. Note that shrinking or growing the size of the file is still possible and allowed. Thus, this seal is normally used in combination with one of the other seals. This seal affects *write*(2) and *fallocate*(2) (only in combination with the **FALLOC_FL_PUNCH_HOLE** flag). Those calls fail with **EPERM** if this seal is set. Furthermore, trying to create new shared, writable memory-mappings via *mmap*(2) will also fail with **EPERM**.

Using the **F_ADD_SEALS** operation to set the **F_SEAL_WRITE** seal fails with **EBUSY** if any writable, shared mapping exists. Such mappings must be unmapped before you can add this seal. Furthermore, if there are any asynchronous I/O operations (**io_submit**(2)) pending on the file, all outstanding writes will be discarded.

F_SEAL_FUTURE_WRITE (since Linux 5.1)
The effect of this seal is similar to **F_SEAL_WRITE**, but the contents of the file can still be modified via shared writable mappings that were created prior to the seal being set. Any attempt to create a new writable mapping on the file via *mmap*(2) will fail with **EPERM**. Likewise, an attempt to write to the file via *write*(2) will fail with **EPERM**.

Using this seal, one process can create a memory buffer that it can continue to modify while sharing that buffer on a "read-only" basis with other processes.

File read/write hints
Write lifetime hints can be used to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. (See *open*(2) for an explanation of open file descriptions.) In this context, the term "write lifetime" means the expected time the data will live on media, before being overwritten or erased.

An application may use the different hint values specified below to separate writes into different write classes, so that multiple users or applications running on a single storage back-end can aggregate their I/O patterns in a consistent manner. However, there are no functional semantics implied by these flags, and different I/O classes can use the write lifetime hints in arbitrary ways, so long as the hints are used consistently.

The following operations can be applied to the file descriptor, *fd*:

F_GET_RW_HINT (*uint64_t* *; since Linux 4.13)
Returns the value of the read/write hint associated with the underlying inode referred to by *fd*.

F_SET_RW_HINT (*uint64_t* *; since Linux 4.13)
Sets the read/write hint value associated with the underlying inode referred to by *fd*. This hint persists until either it is explicitly modified or the underlying filesystem is unmounted.

F_GET_FILE_RW_HINT (*uint64_t* *; since Linux 4.13)
Returns the value of the read/write hint associated with the open file description referred to by *fd*.

F_SET_FILE_RW_HINT (*uint64_t* *; since Linux 4.13)
Sets the read/write hint value associated with the open file description referred to by *fd*.

If an open file description has not been assigned a read/write hint, then it shall use the value assigned to the inode, if any.

The following read/write hints are valid since Linux 4.13:

RWH_WRITE_LIFE_NOT_SET
No specific hint has been set. This is the default value.

RWH_WRITE_LIFE_NONE
No specific write lifetime is associated with this file or inode.

RWH_WRITE_LIFE_SHORT
Data written to this inode or via this open file description is expected to have a short lifetime.

RWH_WRITE_LIFE_MEDIUM
Data written to this inode or via this open file description is expected to have a lifetime longer than data written with **RWH_WRITE_LIFE_SHORT**.

RWH_WRITE_LIFE_LONG
Data written to this inode or via this open file description is expected to have a lifetime longer than data written with **RWH_WRITE_LIFE_MEDIUM**.

RWH_WRITE_LIFE_EXTREME
Data written to this inode or via this open file description is expected to have a lifetime longer than data written with **RWH_WRITE_LIFE_LONG**.

All the write-specific hints are relative to each other, and no individual absolute meaning should be attributed to them.

RETURN VALUE

For a successful call, the return value depends on the operation:

F_DUPFD
The new file descriptor.

F_GETFD
Value of file descriptor flags.

F_GETFL
Value of file status flags.

F_GETLEASE
Type of lease held on file descriptor.

F_GETOWN
Value of file descriptor owner.

F_GETSIG
Value of signal sent when read or write becomes possible, or zero for traditional **SIGIO** behavior.

F_GETPIPE_SZ
F_SETPIPE_SZ
The pipe capacity.

F_GET_SEALS
A bit mask identifying the seals that have been set for the inode referred to by *fd*.

All other operations
Zero.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES or **EAGAIN**
Operation is prohibited by locks held by other processes.

EAGAIN
The operation is prohibited because the file has been memory-mapped by another process.

EBADF
fd is not an open file descriptor

EBADF
op is **F_SETLK** or **F_SETLKW** and the file descriptor open mode doesn't match with the type of lock requested.

EBUSY
op is **F_SETPIPE_SZ** and the new pipe capacity specified in *arg* is smaller than the amount of buffer space currently used to store data in the pipe.

EBUSY
op is **F_ADD_SEALS**, *arg* includes **F_SEAL_WRITE**, and there exists a writable, shared mapping on the file referred to by *fd*.

EDEADLK
It was detected that the specified **F_SETLKW** operation would cause a deadlock.

EFAULT
lock is outside your accessible address space.

EINTR
op is **F_SETLKW** or **F_OFD_SETLKW** and the operation was interrupted by a signal; see *signal*(7).

EINTR
op is **F_GETLK**, **F_SETLK**, **F_OFD_GETLK**, or **F_OFD_SETLK**, and the operation was interrupted by a signal before the lock was checked or acquired. Most likely when locking a remote file (e.g., locking over NFS), but can sometimes happen locally.

EINVAL
The value specified in *op* is not recognized by this kernel.

EINVAL
op is **F_ADD_SEALS** and *arg* includes an unrecognized sealing bit.

EINVAL
op is **F_ADD_SEALS** or **F_GET_SEALS** and the filesystem containing the inode referred to by *fd* does not support sealing.

EINVAL
op is **F_DUPFD** and *arg* is negative or is greater than the maximum allowable value (see the discussion of **RLIMIT_NOFILE** in *getrlimit*(2)).

EINVAL
op is **F_SETSIG** and *arg* is not an allowable signal number.

EINVAL
op is **F_OFD_SETLK**, **F_OFD_SETLKW**, or **F_OFD_GETLK**, and *l_pid* was not specified as zero.

EMFILE
op is **F_DUPFD** and the per-process limit on the number of open file descriptors has been reached.

ENOLCK
Too many segment locks open, lock table is full, or a remote locking protocol failed (e.g., locking over NFS).

ENOTDIR
F_NOTIFY was specified in *op*, but *fd* does not refer to a directory.

EPERM
op is **F_SETPIPE_SZ** and the soft or hard user pipe limit has been reached; see *pipe*(7).

EPERM
Attempted to clear the **O_APPEND** flag on a file that has the append-only attribute set.

EPERM
op was **F_ADD_SEALS**, but *fd* was not open for writing or the current set of seals on the file already includes **F_SEAL_SEAL**.

STANDARDS

POSIX.1-2008.

F_GETOWN_EX, **F_SETOWN_EX**, **F_SETPIPE_SZ**, **F_GETPIPE_SZ**, **F_GETSIG**, **F_SETSIG**, **F_NOTIFY**, **F_GETLEASE**, and **F_SETLEASE** are Linux-specific. (Define the **_GNU_SOURCE** macro to obtain these definitions.)

F_OFD_SETLK, **F_OFD_SETLKW**, and **F_OFD_GETLK** are Linux-specific (and one must define **_GNU_SOURCE** to obtain their definitions), but work is being done to have them included in the next version of POSIX.1.

F_ADD_SEALS and **F_GET_SEALS** are Linux-specific.

HISTORY

SVr4, 4.3BSD, POSIX.1-2001.

Only the operations **F_DUPFD**, **F_GETFD**, **F_SETFD**, **F_GETFL**, **F_SETFL**, **F_GETLK**,

F_SETLK, and **F_SETLKW** are specified in POSIX.1-2001.

F_GETOWN and **F_SETOWN** are specified in POSIX.1-2001. (To get their definitions, define either **_XOPEN_SOURCE** with the value 500 or greater, or **_POSIX_C_SOURCE** with the value 200809L or greater.)

F_DUPFD_CLOEXEC is specified in POSIX.1-2008. (To get this definition, define **_POSIX_C_SOURCE** with the value 200809L or greater, or **_XOPEN_SOURCE** with the value 700 or greater.)

NOTES

The errors returned by *dup2*(2) are different from those returned by **F_DUPFD**.

File locking

The original Linux **fcntl**() system call was not designed to handle large file offsets (in the *flock* structure). Consequently, an **fcntl64**() system call was added in Linux 2.4. The newer system call employs a different structure for file locking, *flock64*, and corresponding operations, **F_GETLK64**, **F_SETLK64**, and **F_SETLKW64**. However, these details can be ignored by applications using glibc, whose **fcntl**() wrapper function transparently employs the more recent system call where it is available.

Record locks

Since Linux 2.0, there is no interaction between the types of lock placed by *flock*(2) and **fcntl**().

Several systems have more fields in *struct flock* such as, for example, *l_sysid* (to identify the machine where the lock is held). Clearly, *l_pid* alone is not going to be very useful if the process holding the lock may live on a different machine; on Linux, while present on some architectures (such as MIPS32), this field is not used.

The original Linux **fcntl**() system call was not designed to handle large file offsets (in the *flock* structure). Consequently, an **fcntl64**() system call was added in Linux 2.4. The newer system call employs a different structure for file locking, *flock64*, and corresponding operations, **F_GETLK64**, **F_SETLK64**, and **F_SETLKW64**. However, these details can be ignored by applications using glibc, whose **fcntl**() wrapper function transparently employs the more recent system call where it is available.

Record locking and NFS

Before Linux 3.12, if an NFSv4 client loses contact with the server for a period of time (defined as more than 90 seconds with no communication), it might lose and regain a lock without ever being aware of the fact. (The period of time after which contact is assumed lost is known as the NFSv4 lease-time. On a Linux NFS server, this can be determined by looking at */proc/fs/nfsd/nfsv4leasetime*, which expresses the period in seconds. The default value for this file is 90.) This scenario potentially risks data corruption, since another process might acquire a lock in the intervening period and perform file I/O.

Since Linux 3.12, if an NFSv4 client loses contact with the server, any I/O to the file by a process which "thinks" it holds a lock will fail until that process closes and reopens the file. A kernel parameter, *nfs.recover_lost_locks*, can be set to 1 to obtain the pre-3.12 behavior, whereby the client will attempt to recover lost locks when contact is reestablished with the server. Because of the attendant risk of data corruption, this parameter defaults to 0 (disabled).

BUGS

F_SETFL

It is not possible to use **F_SETFL** to change the state of the **O_DSYNC** and **O_SYNC** flags. Attempts to change the state of these flags are silently ignored.

F_GETOWN

A limitation of the Linux system call conventions on some architectures (notably i386) means that if a (negative) process group ID to be returned by **F_GETOWN** falls in the range −1 to −4095, then the return value is wrongly interpreted by glibc as an error in the system call; that is, the return value of **fcntl**() will be −1, and *errno* will contain the (positive) process group ID. The Linux-specific **F_GETOWN_EX** operation avoids this problem. Since glibc 2.11, glibc makes the kernel **F_GETOWN** problem invisible by implementing **F_GETOWN** using **F_GETOWN_EX**.

F_SETOWN

In Linux 2.4 and earlier, there is bug that can occur when an unprivileged process uses **F_SETOWN** to specify the owner of a socket file descriptor as a process (group) other than the caller. In this case, **fcntl**() can return −1 with *errno* set to **EPERM**, even when the owner process (group) is one that the

caller has permission to send signals to. Despite this error return, the file descriptor owner is set, and signals will be sent to the owner.

Deadlock detection

The deadlock-detection algorithm employed by the kernel when dealing with **F_SETLKW** requests can yield both false negatives (failures to detect deadlocks, leaving a set of deadlocked processes blocked indefinitely) and false positives (**EDEADLK** errors when there is no deadlock). For example, the kernel limits the lock depth of its dependency search to 10 steps, meaning that circular deadlock chains that exceed that size will not be detected. In addition, the kernel may falsely indicate a deadlock when two or more processes created using the *clone*(2) **CLONE_FILES** flag place locks that appear (to the kernel) to conflict.

Mandatory locking

The Linux implementation of mandatory locking is subject to race conditions which render it unreliable: a *write*(2) call that overlaps with a lock may modify data after the mandatory lock is acquired; a *read*(2) call that overlaps with a lock may detect changes to data that were made only after a write lock was acquired. Similar races exist between mandatory locks and *mmap*(2). It is therefore inadvisable to rely on mandatory locking.

SEE ALSO

dup2(2), *flock*(2), *open*(2), *socket*(2), *lockf*(3), *capabilities*(7), *feature_test_macros*(7), *lslocks*(8)

locks.txt, *mandatory−locking.txt*, and *dnotify.txt* in the Linux kernel source directory *Documentation/filesystems/* (on older kernels, these files are directly under the *Documentation/* directory, and *mandatory−locking.txt* is called *mandatory.txt*)

NAME

flock − apply or remove an advisory lock on an open file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/file.h>

int flock(int *fd***, int** *op***);**

DESCRIPTION

Apply or remove an advisory lock on the open file specified by *fd*. The argument *op* is one of the following:

LOCK_SH
Place a shared lock. More than one process may hold a shared lock for a given file at a given time.

LOCK_EX
Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

LOCK_UN
Remove an existing lock held by this process.

A call to **flock**() may block if an incompatible lock is held by another process. To make a nonblocking request, include **LOCK_NB** (by ORing) with any of the above operations.

A single file may not simultaneously have both shared and exclusive locks.

Locks created by **flock**() are associated with an open file description (see *open*(2)). This means that duplicate file descriptors (created by, for example, *fork*(2) or *dup*(2)) refer to the same lock, and this lock may be modified or released using any of these file descriptors. Furthermore, the lock is released either by an explicit **LOCK_UN** operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

If a process uses *open*(2) (or similar) to obtain more than one file descriptor for the same file, these file descriptors are treated independently by **flock**(). An attempt to lock the file using one of these file descriptors may be denied by a lock that the calling process has already placed via another file descriptor.

A process may hold only one type of lock (shared or exclusive) on a file. Subsequent **flock**() calls on an already locked file will convert an existing lock to the new lock mode.

Locks created by **flock**() are preserved across an *execve*(2).

A shared or exclusive lock can be placed on a file regardless of the mode in which the file was opened.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not an open file descriptor.

EINTR
While waiting to acquire a lock, the call was interrupted by delivery of a signal caught by a handler; see *signal*(7).

EINVAL
op is invalid.

ENOLCK
The kernel ran out of memory for allocating lock records.

EWOULDBLOCK
The file is locked and the **LOCK_NB** flag was selected.

VERSIONS

Since Linux 2.0, **flock**() is implemented as a system call in its own right rather than being emulated in the GNU C library as a call to *fcntl*(2). With this implementation, there is no interaction between the

types of lock placed by **flock**() and *fcntl*(2), and **flock**() does not detect deadlock. (Note, however, that on some systems, such as the modern BSDs, **flock**() and *fcntl*(2) locks *do* interact with one another.)

CIFS details

Up to Linux 5.4, **flock**() is not propagated over SMB. A file with such locks will not appear locked for remote clients.

Since Linux 5.5, **flock**() locks are emulated with SMB byte-range locks on the entire file. Similarly to NFS, this means that *fcntl*(2) and **flock**() locks interact with one another. Another important side-effect is that the locks are not advisory anymore: any IO on a locked file will always fail with **EACCES** when done from a separate file descriptor. This difference originates from the design of locks in the SMB protocol, which provides mandatory locking semantics.

Remote and mandatory locking semantics may vary with SMB protocol, mount options and server type. See *mount.cifs*(8) for additional information.

STANDARDS

BSD.

HISTORY

4.4BSD (the **flock**() call first appeared in 4.2BSD). A version of **flock**(), possibly implemented in terms of *fcntl*(2), appears on most UNIX systems.

NFS details

Up to Linux 2.6.11, **flock**() does not lock files over NFS (i.e., the scope of locks was limited to the local system). Instead, one could use *fcntl*(2) byte-range locking, which does work over NFS, given a sufficiently recent version of Linux and a server which supports locking.

Since Linux 2.6.12, NFS clients support **flock**() locks by emulating them as *fcntl*(2) byte-range locks on the entire file. This means that *fcntl*(2) and **flock**() locks *do* interact with one another over NFS. It also means that in order to place an exclusive lock, the file must be opened for writing.

Since Linux 2.6.37, the kernel supports a compatibility mode that allows **flock**() locks (and also *fcntl*(2) byte region locks) to be treated as local; see the discussion of the *local_lock* option in *nfs*(5)

NOTES

flock() places advisory locks only; given suitable permissions on a file, a process is free to ignore the use of **flock**() and perform I/O on the file.

flock() and *fcntl*(2) locks have different semantics with respect to forked processes and *dup*(2). On systems that implement **flock**() using *fcntl*(2), the semantics of **flock**() will be different from those described in this manual page.

Converting a lock (shared to exclusive, or vice versa) is not guaranteed to be atomic: the existing lock is first removed, and then a new lock is established. Between these two steps, a pending lock request by another process may be granted, with the result that the conversion either blocks, or fails if **LOCK_NB** was specified. (This is the original BSD behavior, and occurs on many other implementations.)

SEE ALSO

flock(1), *close*(2), *dup*(2), *execve*(2), *fcntl*(2), *fork*(2), *open*(2), *lockf*(3), *lslocks*(8)

Documentation/filesystems/locks.txt in the Linux kernel source tree (*Documentation/locks.txt* in older kernels)

NAME

fork − create a child process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (**mmap**(2)), and unmappings (**munmap**(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (**mlock**(2), *mlockall*(2)).
- Process resource utilizations (**getrusage**(2)) and CPU time counters (**times**(2)) are reset to zero in the child.
- The child's set of pending signals is initially empty (**sigpending**(2)).
- The child does not inherit semaphore adjustments from its parent (**semop**(2)).
- The child does not inherit process-associated record locks from its parent (**fcntl**(2)). (On the other hand, it does inherit *fcntl*(2) open file description locks and *flock*(2) locks from its parent.)
- The child does not inherit timers from its parent (**setitimer**(2), *alarm*(2), *timer_create*(2)).
- The child does not inherit outstanding asynchronous I/O operations from its parent (**aio_read**(3), *aio_write*(3)), nor does it inherit any asynchronous I/O contexts from its parent (see *io_setup*(2)).

The process attributes in the preceding list are all specified in POSIX.1. The parent and child also differ with respect to the following Linux-specific process attributes:

- The child does not inherit directory change notifications (dnotify) from its parent (see the description of **F_NOTIFY** in *fcntl*(2)).
- The *prctl*(2) **PR_SET_PDEATHSIG** setting is reset so that the child does not receive a signal when its parent terminates.
- The default timer slack value is set to the parent's current timer slack value. See the description of **PR_SET_TIMERSLACK** in *prctl*(2).
- Memory mappings that have been marked with the *madvise*(2) **MADV_DONTFORK** flag are not inherited across a **fork**().
- Memory in address ranges that have been marked with the *madvise*(2) **MADV_WIPEONFORK** flag is zeroed in the child after a **fork**(). (The **MADV_WIPEONFORK** setting remains in place for those address ranges in the child.)
- The termination signal of the child is always **SIGCHLD** (see *clone*(2)).
- The port access permission bits set by *ioperm*(2) are not inherited by the child; the child must turn on any bits that it requires using *ioperm*(2).

Note the following further points:

- The child process is created with a single thread—the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of *pthread_atfork*(3) may be helpful for dealing with problems that this can cause.

- After a **fork**() in a multithreaded program, the child can safely call only async-signal-safe functions (see *signal-safety*(7)) until such time as it calls *execve*(2).
- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see *open*(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in *fcntl*(2)).
- The child inherits copies of the parent's set of open message queue descriptors (see *mq_overview*(7)). Each file descriptor in the child refers to the same open message queue description as the corresponding file descriptor in the parent. This means that the two file descriptors share the same flags (*mq_flags*).
- The child inherits copies of the parent's set of open directory streams (see *opendir*(3)). POSIX.1 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, −1 is returned in the parent, no child process is created, and *errno* is set to indicate the error.

ERRORS

EAGAIN

A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- the **RLIMIT_NPROC** soft resource limit (set via *setrlimit*(2)), which limits the number of processes and threads for a real user ID, was reached;
- the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see *proc*(5));
- the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see *proc*(5)); or
- the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN

The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See *sched*(7).

ENOMEM

fork() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM

An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See *pid_namespaces*(7).

ENOSYS

fork() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)

System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

VERSIONS

C library/kernel differences

Since glibc 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper that is provided as part of the NPTL threading implementation invokes *clone*(2) with flags that provide the same effect as the traditional system call. (A call to **fork**() is equivalent to a call to *clone*(2) specifying *flags* as just **SIGCHLD**.) The glibc wrapper invokes any fork handlers that have been established using *pthread_atfork*(3).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

EXAMPLES

See *pipe*(2) and *wait*(2) for more examples.

```
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
    pid_t pid;

    if (signal(SIGCHLD, SIG_IGN) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    pid = fork();
    switch (pid) {
    case -1:
        perror("fork");
        exit(EXIT_FAILURE);
    case 0:
        puts("Child exiting.");
        exit(EXIT_SUCCESS);
    default:
        printf("Child is PID %jd\n", (intmax_t) pid);
        puts("Parent exiting.");
        exit(EXIT_SUCCESS);
    }
}
```

SEE ALSO

clone(2), *execve*(2), *exit*(2), *setrlimit*(2), *unshare*(2), *vfork*(2), *wait*(2), *daemon*(3), *pthread_atfork*(3), *capabilities*(7), *credentials*(7)

NAME

fsync, fdatasync − synchronize a file's in-core state with storage device

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fsync():
glibc 2.16 and later:
No feature test macros need be defined
glibc up to and including 2.15:
_BSD_SOURCE || _XOPEN_SOURCE
|| /* Since glibc 2.8: */ _POSIX_C_SOURCE >= 200112L

fdatasync():
_POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500

DESCRIPTION

fsync() transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor *fd* to the disk device (or other permanent storage device) so that all changed information can be retrieved even if the system crashes or is rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed.

As well as flushing the file data, **fsync**() also flushes the metadata information associated with the file (see *inode*(7)).

Calling **fsync**() does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit **fsync**() on a file descriptor for the directory is also needed.

fdatasync() is similar to **fsync**(), but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled. For example, changes to *st_atime* or *st_mtime* (respectively, time of last access and time of last modification; see *inode*(7)) do not require flushing because they are not necessary for a subsequent data read to be handled correctly. On the other hand, a change to the file size (*st_size*, as made by say *ftruncate*(2)), would require a metadata flush.

The aim of **fdatasync**() is to reduce disk activity for applications that do not require all metadata to be synchronized with the disk.

RETURN VALUE

On success, these system calls return zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not a valid open file descriptor.

EINTR
The function was interrupted by a signal; see *signal*(7).

EIO An error occurred during synchronization. This error may relate to data written to some other file descriptor on the same file. Since Linux 4.13, errors from write-back will be reported to all file descriptors that might have written the data which triggered the error. Some filesystems (e.g., NFS) keep close track of which data came through which file descriptor, and give more precise reporting. Other filesystems (e.g., most local filesystems) will report errors to all file descriptors that were open on the file when the error was recorded.

ENOSPC
Disk space was exhausted while synchronizing.

EROFS
EINVAL
fd is bound to a special file (e.g., a pipe, FIFO, or socket) which does not support synchronization.

ENOSPC
EDQUOT
fd is bound to a file on NFS or another filesystem which does not allocate space at the time of a *write*(2) system call, and some previous write failed due to insufficient storage space.

VERSIONS

On POSIX systems on which **fdatasync**() is available, **_POSIX_SYNCHRONIZED_IO** is defined in *<unistd.h>* to a value greater than 0. (See also *sysconf*(3).)

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.2BSD.

In Linux 2.2 and earlier, **fdatasync**() is equivalent to **fsync**(), and so has no performance advantage.

The **fsync**() implementations in older kernels and lesser used filesystems do not know how to flush disk caches. In these cases disk caches need to be disabled using *hdparm*(8) or *sdparm*(8) to guarantee safe operation.

Under AT&T UNIX System V Release 4 *fd* needs to be opened for writing. This is by itself incompatible with the original BSD interface and forbidden by POSIX, but nevertheless survives in HP-UX and AIX.

SEE ALSO

sync(1), *bdflush*(2), *open*(2), *posix_fadvise*(2), *pwritev*(2), *sync*(2), *sync_file_range*(2), *fflush*(3), *fileno*(3), *hdparm*(8), *mount*(8)

NAME

futex − fast user-space locking

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/futex.h>      /* Definition of FUTEX_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_futex, uint32_t *uaddr, int  futex_op, uint32_t val,
             const struct timespec *timeout,   /* or: uint32_t val2 */
             uint32_t *uaddr2, uint32_t val3);
```

Note: glibc provides no wrapper for **futex**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **futex**() system call provides a method for waiting until a certain condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronization. When using futexes, the majority of the synchronization operations are performed in user space. A user-space program employs the **futex**() system call only when it is likely that the program has to block for a longer time until the condition becomes true. Other **futex**() operations can be used to wake any processes or threads waiting for a particular condition.

A futex is a 32-bit value—referred to below as a *futex word*—whose address is supplied to the **futex**() system call. (Futexes are 32 bits in size on all platforms, including 64-bit systems.) All futex operations are governed by this value. In order to share a futex between processes, the futex is placed in a region of shared memory, created using (for example) *mmap*(2) or *shmat*(2). (Thus, the futex word may have different virtual addresses in different processes, but these addresses all refer to the same location in physical memory.) In a multithreaded program, it is sufficient to place the futex word in a global variable shared by all threads.

When executing a futex operation that requests to block a thread, the kernel will block only if the futex word has the value that the calling thread supplied (as one of the arguments of the **futex**() call) as the expected value of the futex word. The loading of the futex word's value, the comparison of that value with the expected value, and the actual blocking will happen atomically and will be totally ordered with respect to concurrent operations performed by other threads on the same futex word. Thus, the futex word is used to connect the synchronization in user space with the implementation of blocking by the kernel. Analogously to an atomic compare-and-exchange operation that potentially changes shared memory, blocking via a futex is an atomic compare-and-block operation.

One use of futexes is for implementing locks. The state of the lock (i.e., acquired or not acquired) can be represented as an atomically accessed flag in shared memory. In the uncontended case, a thread can access or modify the lock state with atomic instructions, for example atomically changing it from not acquired to acquired using an atomic compare-and-exchange instruction. (Such instructions are performed entirely in user mode, and the kernel maintains no information about the lock state.) On the other hand, a thread may be unable to acquire a lock because it is already acquired by another thread. It then may pass the lock's flag as a futex word and the value representing the acquired state as the expected value to a **futex**() wait operation. This **futex**() operation will block if and only if the lock is still acquired (i.e., the value in the futex word still matches the "acquired state"). When releasing the lock, a thread has to first reset the lock state to not acquired and then execute a futex operation that wakes threads blocked on the lock flag used as a futex word (this can be further optimized to avoid unnecessary wake-ups). See *futex*(7) for more detail on how to use futexes.

Besides the basic wait and wake-up futex functionality, there are further futex operations aimed at supporting more complex use cases.

Note that no explicit initialization or destruction is necessary to use futexes; the kernel maintains a futex (i.e., the kernel-internal implementation artifact) only while operations such as **FUTEX_WAIT**, described below, are being performed on a particular futex word.

Arguments

The *uaddr* argument points to the futex word. On all platforms, futexes are four-byte integers that must be aligned on a four-byte boundary. The operation to perform on the futex is specified in the *futex_op*

argument; *val* is a value whose meaning and purpose depends on *futex_op*.

The remaining arguments (*timeout*, *uaddr2*, and *val3*) are required only for certain of the futex operations described below. Where one of these arguments is not required, it is ignored.

For several blocking operations, the *timeout* argument is a pointer to a *timespec* structure that specifies a timeout for the operation. However, notwithstanding the prototype shown above, for some operations, the least significant four bytes of this argument are instead used as an integer whose meaning is determined by the operation. For these operations, the kernel casts the *timeout* value first to *unsigned long*, then to *uint32_t*, and in the remainder of this page, this argument is referred to as *val2* when interpreted in this fashion.

Where it is required, the *uaddr2* argument is a pointer to a second futex word that is employed by the operation.

The interpretation of the final integer argument, *val3*, depends on the operation.

Futex operations

The *futex_op* argument consists of two parts: a command that specifies the operation to be performed, bitwise ORed with zero or more options that modify the behaviour of the operation. The options that may be included in *futex_op* are as follows:

FUTEX_PRIVATE_FLAG (since Linux 2.6.22)

This option bit can be employed with all futex operations. It tells the kernel that the futex is process-private and not shared with another process (i.e., it is being used for synchronization only between threads of the same process). This allows the kernel to make some additional performance optimizations.

As a convenience, <*linux/futex.h*> defines a set of constants with the suffix **_PRIVATE** that are equivalents of all of the operations listed below, but with the **FUTEX_PRIVATE_FLAG** ORed into the constant value. Thus, there are **FUTEX_WAIT_PRIVATE**, **FUTEX_WAKE_PRIVATE**, and so on.

FUTEX_CLOCK_REALTIME (since Linux 2.6.28)

This option bit can be employed only with the **FUTEX_WAIT_BITSET**, **FUTEX_WAIT_REQUEUE_PI**, (since Linux 4.5) **FUTEX_WAIT**, and (since Linux 5.14) **FUTEX_LOCK_PI2** operations.

If this option is set, the kernel measures the *timeout* against the **CLOCK_REALTIME** clock.

If this option is not set, the kernel measures the *timeout* against the **CLOCK_MONOTONIC** clock.

The operation specified in *futex_op* is one of the following:

FUTEX_WAIT (since Linux 2.6.0)

This operation tests that the value at the futex word pointed to by the address *uaddr* still contains the expected value *val*, and if so, then sleeps waiting for a **FUTEX_WAKE** operation on the futex word. The load of the value of the futex word is an atomic memory access (i.e., using atomic machine instructions of the respective architecture). This load, the comparison with the expected value, and starting to sleep are performed atomically and totally ordered with respect to other futex operations on the same futex word. If the thread starts to sleep, it is considered a waiter on this futex word. If the futex value does not match *val*, then the call fails immediately with the error **EAGAIN**.

The purpose of the comparison with the expected value is to prevent lost wake-ups. If another thread changed the value of the futex word after the calling thread decided to block based on the prior value, and if the other thread executed a **FUTEX_WAKE** operation (or similar wake-up) after the value change and before this **FUTEX_WAIT** operation, then the calling thread will observe the value change and will not start to sleep.

If the *timeout* is not NULL, the structure it points to specifies a timeout for the wait. (This interval will be rounded up to the system clock granularity, and is guaranteed not to expire early.) The timeout is by default measured according to the **CLOCK_MONOTONIC** clock, but, since Linux 4.5, the **CLOCK_REALTIME** clock can be selected by specifying **FUTEX_CLOCK_REALTIME** in *futex_op*. If *timeout* is NULL, the call blocks indefinitely.

Note: for **FUTEX_WAIT**, *timeout* is interpreted as a *relative* value. This differs from other futex operations, where *timeout* is interpreted as an absolute value. To obtain the equivalent of **FUTEX_WAIT** with an absolute timeout, employ **FUTEX_WAIT_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**.

The arguments *uaddr2* and *val3* are ignored.

FUTEX_WAKE (since Linux 2.6.0)
This operation wakes at most *val* of the waiters that are waiting (e.g., inside **FUTEX_WAIT**) on the futex word at the address *uaddr*. Most commonly, *val* is specified as either 1 (wake up a single waiter) or **INT_MAX** (wake up all waiters). No guarantee is provided about which waiters are awoken (e.g., a waiter with a higher scheduling priority is not guaranteed to be awoken in preference to a waiter with a lower priority).

The arguments *timeout*, *uaddr2*, and *val3* are ignored.

FUTEX_FD (from Linux 2.6.0 up to and including Linux 2.6.25)
This operation creates a file descriptor that is associated with the futex at *uaddr*. The caller must close the returned file descriptor after use. When another process or thread performs a **FUTEX_WAKE** on the futex word, the file descriptor indicates as being readable with *select*(2), *poll*(2), and *epoll*(7)

The file descriptor can be used to obtain asynchronous notifications: if *val* is nonzero, then, when another process or thread executes a **FUTEX_WAKE**, the caller will receive the signal number that was passed in *val*.

The arguments *timeout*, *uaddr2*, and *val3* are ignored.

Because it was inherently racy, **FUTEX_FD** has been removed from Linux 2.6.26 onward.

FUTEX_REQUEUE (since Linux 2.6.0)
This operation performs the same task as **FUTEX_CMP_REQUEUE** (see below), except that no check is made using the value in *val3*. (The argument *val3* is ignored.)

FUTEX_CMP_REQUEUE (since Linux 2.6.7)
This operation first checks whether the location *uaddr* still contains the value *val3*. If not, the operation fails with the error **EAGAIN**. Otherwise, the operation wakes up a maximum of *val* waiters that are waiting on the futex at *uaddr*. If there are more than *val* waiters, then the remaining waiters are removed from the wait queue of the source futex at *uaddr* and added to the wait queue of the target futex at *uaddr2*. The *val2* argument specifies an upper limit on the number of waiters that are requeued to the futex at *uaddr2*.

The load from *uaddr* is an atomic memory access (i.e., using atomic machine instructions of the respective architecture). This load, the comparison with *val3*, and the requeueing of any waiters are performed atomically and totally ordered with respect to other operations on the same futex word.

Typical values to specify for *val* are 0 or 1. (Specifying **INT_MAX** is not useful, because it would make the **FUTEX_CMP_REQUEUE** operation equivalent to **FUTEX_WAKE**.) The limit value specified via *val2* is typically either 1 or **INT_MAX**. (Specifying the argument as 0 is not useful, because it would make the **FUTEX_CMP_REQUEUE** operation equivalent to **FUTEX_WAIT**.)

The **FUTEX_CMP_REQUEUE** operation was added as a replacement for the earlier **FUTEX_REQUEUE**. The difference is that the check of the value at *uaddr* can be used to ensure that requeueing happens only under certain conditions, which allows race conditions to be avoided in certain use cases.

Both **FUTEX_REQUEUE** and **FUTEX_CMP_REQUEUE** can be used to avoid "thundering herd" wake-ups that could occur when using **FUTEX_WAKE** in cases where all of the waiters that are woken need to acquire another futex. Consider the following scenario, where multiple waiter threads are waiting on B, a wait queue implemented using a futex:

```
lock(A)
while (!check_value(V)) {
    unlock(A);
    block_on(B);
```

```
    lock(A);
};
unlock(A);
```

If a waker thread used **FUTEX_WAKE**, then all waiters waiting on B would be woken up, and they would all try to acquire lock A. However, waking all of the threads in this manner would be pointless because all except one of the threads would immediately block on lock A again. By contrast, a requeue operation wakes just one waiter and moves the other waiters to lock A, and when the woken waiter unlocks A then the next waiter can proceed.

FUTEX_WAKE_OP (since Linux 2.6.14)

This operation was added to support some user-space use cases where more than one futex must be handled at the same time. The most notable example is the implementation of *pthread_cond_signal*(3), which requires operations on two futexes, the one used to implement the mutex and the one used in the implementation of the wait queue associated with the condition variable. **FUTEX_WAKE_OP** allows such cases to be implemented without leading to high rates of contention and context switching.

The **FUTEX_WAKE_OP** operation is equivalent to executing the following code atomically and totally ordered with respect to other futex operations on any of the two supplied futex words:

```
uint32_t oldval = *(uint32_t *) uaddr2;
*(uint32_t *) uaddr2 = oldval op oparg;
futex(uaddr, FUTEX_WAKE, val, 0, 0, 0);
if (oldval cmp cmparg)
    futex(uaddr2, FUTEX_WAKE, val2, 0, 0, 0);
```

In other words, **FUTEX_WAKE_OP** does the following:

- saves the original value of the futex word at *uaddr2* and performs an operation to modify the value of the futex at *uaddr2*; this is an atomic read-modify-write memory access (i.e., using atomic machine instructions of the respective architecture)
- wakes up a maximum of *val* waiters on the futex for the futex word at *uaddr*; and
- dependent on the results of a test of the original value of the futex word at *uaddr2*, wakes up a maximum of *val2* waiters on the futex for the futex word at *uaddr2*.

The operation and comparison that are to be performed are encoded in the bits of the argument *val3*. Pictorially, the encoding is:

```
+---+---+-----------+-----------+
|op |cmp|   oparg   |  cmparg   |
+---+---+-----------+-----------+
  4   4       12          12    <== # of bits
```

Expressed in code, the encoding is:

```
#define FUTEX_OP(op, oparg, cmp, cmparg) \
                ((((op & 0xf) << 28) | \
                ((cmp & 0xf) << 24) | \
                ((oparg & 0xfff) << 12) | \
                (cmparg & 0xfff))
```

In the above, *op* and *cmp* are each one of the codes listed below. The *oparg* and *cmparg* components are literal numeric values, except as noted below.

The *op* component has one of the following values:

```
FUTEX_OP_SET        0  /* uaddr2 = oparg; */
FUTEX_OP_ADD        1  /* uaddr2 += oparg; */
FUTEX_OP_OR         2  /* uaddr2 |= oparg; */
FUTEX_OP_ANDN       3  /* uaddr2 &= ~oparg; */
FUTEX_OP_XOR        4  /* uaddr2 ^= oparg; */
```

In addition, bitwise ORing the following value into *op* causes *(1 << oparg)* to be used as the operand:

```
FUTEX_OP_ARG_SHIFT  8  /* Use (1 << oparg) as operand */
```

The *cmp* field is one of the following:

```
FUTEX_OP_CMP_EQ     0  /* if (oldval == cmparg) wake */
FUTEX_OP_CMP_NE     1  /* if (oldval != cmparg) wake */
FUTEX_OP_CMP_LT     2  /* if (oldval < cmparg) wake */
FUTEX_OP_CMP_LE     3  /* if (oldval <= cmparg) wake */
FUTEX_OP_CMP_GT     4  /* if (oldval > cmparg) wake */
FUTEX_OP_CMP_GE     5  /* if (oldval >= cmparg) wake */
```

The return value of **FUTEX_WAKE_OP** is the sum of the number of waiters woken on the futex *uaddr* plus the number of waiters woken on the futex *uaddr2*.

FUTEX_WAIT_BITSET (since Linux 2.6.25)

This operation is like **FUTEX_WAIT** except that *val3* is used to provide a 32-bit bit mask to the kernel. This bit mask, in which at least one bit must be set, is stored in the kernel-internal state of the waiter. See the description of **FUTEX_WAKE_BITSET** for further details.

If *timeout* is not NULL, the structure it points to specifies an absolute timeout for the wait operation. If *timeout* is NULL, the operation can block indefinitely.

The *uaddr2* argument is ignored.

FUTEX_WAKE_BITSET (since Linux 2.6.25)

This operation is the same as **FUTEX_WAKE** except that the *val3* argument is used to provide a 32-bit bit mask to the kernel. This bit mask, in which at least one bit must be set, is used to select which waiters should be woken up. The selection is done by a bitwise AND of the "wake" bit mask (i.e., the value in *val3*) and the bit mask which is stored in the kernel-internal state of the waiter (the "wait" bit mask that is set using **FUTEX_WAIT_BITSET**). All of the waiters for which the result of the AND is nonzero are woken up; the remaining waiters are left sleeping.

The effect of **FUTEX_WAIT_BITSET** and **FUTEX_WAKE_BITSET** is to allow selective wake-ups among multiple waiters that are blocked on the same futex. However, note that, depending on the use case, employing this bit-mask multiplexing feature on a futex can be less efficient than simply using multiple futexes, because employing bit-mask multiplexing requires the kernel to check all waiters on a futex, including those that are not interested in being woken up (i.e., they do not have the relevant bit set in their "wait" bit mask).

The constant **FUTEX_BITSET_MATCH_ANY**, which corresponds to all 32 bits set in the bit mask, can be used as the *val3* argument for **FUTEX_WAIT_BITSET** and **FUTEX_WAKE_BITSET**. Other than differences in the handling of the *timeout* argument, the **FUTEX_WAIT** operation is equivalent to **FUTEX_WAIT_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**; that is, allow a wake-up by any waker. The **FUTEX_WAKE** operation is equivalent to **FUTEX_WAKE_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**; that is, wake up any waiter(s).

The *uaddr2* and *timeout* arguments are ignored.

Priority-inheritance futexes

Linux supports priority-inheritance (PI) futexes in order to handle priority-inversion problems that can be encountered with normal futex locks. Priority inversion is the problem that occurs when a high-priority task is blocked waiting to acquire a lock held by a low-priority task, while tasks at an intermediate priority continuously preempt the low-priority task from the CPU. Consequently, the low-priority task makes no progress toward releasing the lock, and the high-priority task remains blocked.

Priority inheritance is a mechanism for dealing with the priority-inversion problem. With this mechanism, when a high-priority task becomes blocked by a lock held by a low-priority task, the priority of the low-priority task is temporarily raised to that of the high-priority task, so that it is not preempted by any intermediate level tasks, and can thus make progress toward releasing the lock. To be effective, priority inheritance must be transitive, meaning that if a high-priority task blocks on a lock held by a lower-priority task that is itself blocked by a lock held by another intermediate-priority task (and so on, for chains of arbitrary length), then both of those tasks (or more generally, all of the tasks in a lock chain) have their priorities raised to be the same as the high-priority task.

From a user-space perspective, what makes a futex PI-aware is a policy agreement (described below) between user space and the kernel about the value of the futex word, coupled with the use of the PI-futex operations described below. (Unlike the other futex operations described above, the PI-futex operations are designed for the implementation of very specific IPC mechanisms.)

The PI-futex operations described below differ from the other futex operations in that they impose policy on the use of the value of the futex word:

- If the lock is not acquired, the futex word's value shall be 0.
- If the lock is acquired, the futex word's value shall be the thread ID (TID; see *gettid*(2)) of the owning thread.
- If the lock is owned and there are threads contending for the lock, then the **FUTEX_WAITERS** bit shall be set in the futex word's value; in other words, this value is:

  ```
  FUTEX_WAITERS | TID
  ```

  (Note that is invalid for a PI futex word to have no owner and **FUTEX_WAITERS** set.)

With this policy in place, a user-space application can acquire an unacquired lock or release a lock using atomic instructions executed in user mode (e.g., a compare-and-swap operation such as *cmpxchg* on the x86 architecture). Acquiring a lock simply consists of using compare-and-swap to atomically set the futex word's value to the caller's TID if its previous value was 0. Releasing a lock requires using compare-and-swap to set the futex word's value to 0 if the previous value was the expected TID.

If a futex is already acquired (i.e., has a nonzero value), waiters must employ the **FUTEX_LOCK_PI** or **FUTEX_LOCK_PI2** operations to acquire the lock. If other threads are waiting for the lock, then the **FUTEX_WAITERS** bit is set in the futex value; in this case, the lock owner must employ the **FUTEX_UNLOCK_PI** operation to release the lock.

In the cases where callers are forced into the kernel (i.e., required to perform a **futex**() call), they then deal directly with a so-called RT-mutex, a kernel locking mechanism which implements the required priority-inheritance semantics. After the RT-mutex is acquired, the futex value is updated accordingly, before the calling thread returns to user space.

It is important to note that the kernel will update the futex word's value prior to returning to user space. (This prevents the possibility of the futex word's value ending up in an invalid state, such as having an owner but the value being 0, or having waiters but not having the **FUTEX_WAITERS** bit set.)

If a futex has an associated RT-mutex in the kernel (i.e., there are blocked waiters) and the owner of the futex/RT-mutex dies unexpectedly, then the kernel cleans up the RT-mutex and hands it over to the next waiter. This in turn requires that the user-space value is updated accordingly. To indicate that this is required, the kernel sets the **FUTEX_OWNER_DIED** bit in the futex word along with the thread ID of the new owner. User space can detect this situation via the presence of the **FUTEX_OWNER_DIED** bit and is then responsible for cleaning up the stale state left over by the dead owner.

PI futexes are operated on by specifying one of the values listed below in *futex_op*. Note that the PI futex operations must be used as paired operations and are subject to some additional requirements:

- **FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, and **FUTEX_TRYLOCK_PI** pair with **FUTEX_UNLOCK_PI**. **FUTEX_UNLOCK_PI** must be called only on a futex owned by the calling thread, as defined by the value policy, otherwise the error **EPERM** results.
- **FUTEX_WAIT_REQUEUE_PI** pairs with **FUTEX_CMP_REQUEUE_PI**. This must be performed from a non-PI futex to a distinct PI futex (or the error **EINVAL** results). Additionally, *val* (the number of waiters to be woken) must be 1 (or the error **EINVAL** results).

The PI futex operations are as follows:

FUTEX_LOCK_PI (since Linux 2.6.18)

This operation is used after an attempt to acquire the lock via an atomic user-mode instruction failed because the futex word has a nonzero value—specifically, because it contained the (PID-namespace-specific) TID of the lock owner.

The operation checks the value of the futex word at the address *uaddr*. If the value is 0, then the kernel tries to atomically set the futex value to the caller's TID. If the futex word's value is nonzero, the kernel atomically sets the **FUTEX_WAITERS** bit, which signals the futex owner that it cannot unlock the futex in user space atomically by setting the futex value to 0.

After that, the kernel:

(1) Tries to find the thread which is associated with the owner TID.

(2) Creates or reuses kernel state on behalf of the owner. (If this is the first waiter, there is no kernel state for this futex, so kernel state is created by locking the RT-mutex and the futex owner is made the owner of the RT-mutex. If there are existing waiters, then the existing state is reused.)

(3) Attaches the waiter to the futex (i.e., the waiter is enqueued on the RT-mutex waiter list).

If more than one waiter exists, the enqueueing of the waiter is in descending priority order. (For information on priority ordering, see the discussion of the **SCHED_DEADLINE**, **SCHED_FIFO**, and **SCHED_RR** scheduling policies in *sched*(7).) The owner inherits either the waiter's CPU bandwidth (if the waiter is scheduled under the **SCHED_DEADLINE** policy) or the waiter's priority (if the waiter is scheduled under the **SCHED_RR** or **SCHED_FIFO** policy). This inheritance follows the lock chain in the case of nested locking and performs deadlock detection.

The *timeout* argument provides a timeout for the lock attempt. If *timeout* is not NULL, the structure it points to specifies an absolute timeout, measured against the **CLOCK_REALTIME** clock. If *timeout* is NULL, the operation will block indefinitely.

The *uaddr2*, *val*, and *val3* arguments are ignored.

FUTEX_LOCK_PI2 (since Linux 5.14)
This operation is the same as **FUTEX_LOCK_PI**, except that the clock against which *timeout* is measured is selectable. By default, the (absolute) timeout specified in *timeout* is measured against the **CLOCK_MONOTONIC** clock, but if the **FUTEX_CLOCK_REALTIME** flag is specified in *futex_op*, then the timeout is measured against the **CLOCK_REALTIME** clock.

FUTEX_TRYLOCK_PI (since Linux 2.6.18)
This operation tries to acquire the lock at *uaddr*. It is invoked when a user-space atomic acquire did not succeed because the futex word was not 0.

Because the kernel has access to more state information than user space, acquisition of the lock might succeed if performed by the kernel in cases where the futex word (i.e., the state information accessible to use-space) contains stale state (**FUTEX_WAITERS** and/or **FUTEX_OWNER_DIED**). This can happen when the owner of the futex died. User space cannot handle this condition in a race-free manner, but the kernel can fix this up and acquire the futex.

The *uaddr2*, *val*, *timeout*, and *val3* arguments are ignored.

FUTEX_UNLOCK_PI (since Linux 2.6.18)
This operation wakes the top priority waiter that is waiting in **FUTEX_LOCK_PI** or **FUTEX_LOCK_PI2** on the futex address provided by the *uaddr* argument.

This is called when the user-space value at *uaddr* cannot be changed atomically from a TID (of the owner) to 0.

The *uaddr2*, *val*, *timeout*, and *val3* arguments are ignored.

FUTEX_CMP_REQUEUE_PI (since Linux 2.6.31)
This operation is a PI-aware variant of **FUTEX_CMP_REQUEUE**. It requeues waiters that are blocked via **FUTEX_WAIT_REQUEUE_PI** on *uaddr* from a non-PI source futex (*uaddr*) to a PI target futex (*uaddr2*).

As with **FUTEX_CMP_REQUEUE**, this operation wakes up a maximum of *val* waiters that are waiting on the futex at *uaddr*. However, for **FUTEX_CMP_REQUEUE_PI**, *val* is required to be 1 (since the main point is to avoid a thundering herd). The remaining waiters are removed from the wait queue of the source futex at *uaddr* and added to the wait queue of the target futex at *uaddr2*.

The *val2* and *val3* arguments serve the same purposes as for **FUTEX_CMP_REQUEUE**.

FUTEX_WAIT_REQUEUE_PI (since Linux 2.6.31)
Wait on a non-PI futex at *uaddr* and potentially be requeued (via a **FUTEX_CMP_REQUEUE_PI** operation in another task) onto a PI futex at *uaddr2*. The wait operation on *uaddr* is the same as for **FUTEX_WAIT**.

The waiter can be removed from the wait on *uaddr* without requeueing on *uaddr2* via a **FUTEX_WAKE** operation in another task. In this case, the **FUTEX_WAIT_REQUEUE_PI** operation fails with the error **EAGAIN**.

If *timeout* is not NULL, the structure it points to specifies an absolute timeout for the wait operation. If *timeout* is NULL, the operation can block indefinitely.

The *val3* argument is ignored.

The **FUTEX_WAIT_REQUEUE_PI** and **FUTEX_CMP_REQUEUE_PI** were added to support a fairly specific use case: support for priority-inheritance-aware POSIX threads condition variables. The idea is that these operations should always be paired, in order to ensure that user space and the kernel remain in sync. Thus, in the **FUTEX_WAIT_REQUEUE_PI** operation, the user-space application pre-specifies the target of the requeue that takes place in the **FUTEX_CMP_REQUEUE_PI** operation.

RETURN VALUE

In the event of an error (and assuming that **futex**() was invoked via *syscall*(2)), all operations return −1 and set *errno* to indicate the error.

The return value on success depends on the operation, as described in the following list:

FUTEX_WAIT
Returns 0 if the caller was woken up. Note that a wake-up can also be caused by common futex usage patterns in unrelated code that happened to have previously used the futex word's memory location (e.g., typical futex-based implementations of Pthreads mutexes can cause this under some conditions). Therefore, callers should always conservatively assume that a return value of 0 can mean a spurious wake-up, and use the futex word's value (i.e., the user-space synchronization scheme) to decide whether to continue to block or not.

FUTEX_WAKE
Returns the number of waiters that were woken up.

FUTEX_FD
Returns the new file descriptor associated with the futex.

FUTEX_REQUEUE
Returns the number of waiters that were woken up.

FUTEX_CMP_REQUEUE
Returns the total number of waiters that were woken up or requeued to the futex for the futex word at *uaddr2*. If this value is greater than *val*, then the difference is the number of waiters requeued to the futex for the futex word at *uaddr2*.

FUTEX_WAKE_OP
Returns the total number of waiters that were woken up. This is the sum of the woken waiters on the two futexes for the futex words at *uaddr* and *uaddr2*.

FUTEX_WAIT_BITSET
Returns 0 if the caller was woken up. See **FUTEX_WAIT** for how to interpret this correctly in practice.

FUTEX_WAKE_BITSET
Returns the number of waiters that were woken up.

FUTEX_LOCK_PI
Returns 0 if the futex was successfully locked.

FUTEX_LOCK_PI2
Returns 0 if the futex was successfully locked.

FUTEX_TRYLOCK_PI
Returns 0 if the futex was successfully locked.

FUTEX_UNLOCK_PI
Returns 0 if the futex was successfully unlocked.

FUTEX_CMP_REQUEUE_PI
Returns the total number of waiters that were woken up or requeued to the futex for the futex word at *uaddr2*. If this value is greater than *val*, then difference is the number of waiters requeued to the futex for the futex word at *uaddr2*.

FUTEX_WAIT_REQUEUE_PI
Returns 0 if the caller was successfully requeued to the futex for the futex word at *uaddr2*.

ERRORS

EACCES
No read access to the memory of a futex word.

EAGAIN
(**FUTEX_WAIT**, **FUTEX_WAIT_BITSET**, **FUTEX_WAIT_REQUEUE_PI**) The value pointed to by *uaddr* was not equal to the expected value *val* at the time of the call.

Note: on Linux, the symbolic names **EAGAIN** and **EWOULDBLOCK** (both of which appear in different parts of the kernel futex code) have the same value.

EAGAIN
(**FUTEX_CMP_REQUEUE**, **FUTEX_CMP_REQUEUE_PI**) The value pointed to by *uaddr* is not equal to the expected value *val3*.

EAGAIN
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The futex owner thread ID of *uaddr* (for **FUTEX_CMP_REQUEUE_PI**: *uaddr2*) is about to exit, but has not yet handled the internal state cleanup. Try again.

EDEADLK
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The futex word at *uaddr* is already locked by the caller.

EDEADLK
(**FUTEX_CMP_REQUEUE_PI**) While requeueing a waiter to the PI futex for the futex word at *uaddr2*, the kernel detected a deadlock.

EFAULT
A required pointer argument (i.e., *uaddr*, *uaddr2*, or *timeout*) did not point to a valid user-space address.

EINTR
A **FUTEX_WAIT** or **FUTEX_WAIT_BITSET** operation was interrupted by a signal (see *signal*(7)). Before Linux 2.6.22, this error could also be returned for a spurious wakeup; since Linux 2.6.22, this no longer happens.

EINVAL
The operation in *futex_op* is one of those that employs a timeout, but the supplied *timeout* argument was invalid (*tv_sec* was less than zero, or *tv_nsec* was not less than 1,000,000,000).

EINVAL
The operation specified in *futex_op* employs one or both of the pointers *uaddr* and *uaddr2*, but one of these does not point to a valid object—that is, the address is not four-byte-aligned.

EINVAL
(**FUTEX_WAIT_BITSET**, **FUTEX_WAKE_BITSET**) The bit mask supplied in *val3* is zero.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) *uaddr* equals *uaddr2* (i.e., an attempt was made to requeue to the same futex).

EINVAL
(**FUTEX_FD**) The signal number supplied in *val* is invalid.

EINVAL
(**FUTEX_WAKE**, **FUTEX_WAKE_OP**, **FUTEX_WAKE_BITSET**, **FUTEX_REQUEUE**, **FUTEX_CMP_REQUEUE**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state—that is, it detected a waiter which waits in **FUTEX_LOCK_PI** or **FUTEX_LOCK_PI2** on *uaddr*.

EINVAL
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_UNLOCK_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state. This indicates either state corruption or that the kernel found a waiter on *uaddr* which is waiting via **FUTEX_WAIT** or **FUTEX_WAIT_BITSET**.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr2* and the kernel state; that is, the kernel detected a waiter which waits via **FUTEX_WAIT** or **FUTEX_WAIT_BITSET** on *uaddr2*.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state; that is, the kernel detected a waiter which waits via **FUTEX_WAIT** or **FUTEX_WAIT_BITSET** on *uaddr*.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state; that is, the kernel detected a waiter which waits on *uaddr* via **FUTEX_LOCK_PI** or **FUTEX_LOCK_PI2** (instead of **FUTEX_WAIT_REQUEUE_PI**).

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) An attempt was made to requeue a waiter to a futex other than that specified by the matching **FUTEX_WAIT_REQUEUE_PI** call for that waiter.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The *val* argument is not 1.

EINVAL
Invalid argument.

ENFILE
(**FUTEX_FD**) The system-wide limit on the total number of open files has been reached.

ENOMEM
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The kernel could not allocate memory to hold state information.

ENOSYS
Invalid operation specified in *futex_op*.

ENOSYS
The **FUTEX_CLOCK_REALTIME** option was specified in *futex_op*, but the accompanying operation was neither **FUTEX_WAIT**, **FUTEX_WAIT_BITSET**, **FUTEX_WAIT_REQUEUE_PI**, nor **FUTEX_LOCK_PI2**.

ENOSYS
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_UNLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**, **FUTEX_WAIT_REQUEUE_PI**) A run-time check determined that the operation is not available. The PI-futex operations are not implemented on all architectures and are not supported on some CPU variants.

EPERM
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The caller is not allowed to attach itself to the futex at *uaddr* (for **FUTEX_CMP_REQUEUE_PI**: the futex at *uaddr2*). (This may be caused by a state corruption in user space.)

EPERM
(**FUTEX_UNLOCK_PI**) The caller does not own the lock represented by the futex word.

ESRCH
(**FUTEX_LOCK_PI**, **FUTEX_LOCK_PI2**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The thread ID in the futex word at *uaddr* does not exist.

ESRCH
(**FUTEX_CMP_REQUEUE_PI**) The thread ID in the futex word at *uaddr2* does not exist.

ETIMEDOUT
The operation in *futex_op* employed the timeout specified in *timeout*, and the timeout expired before the operation completed.

STANDARDS
Linux.

HISTORY
Linux 2.6.0.

Initial futex support was merged in Linux 2.5.7 but with different semantics from what was described above. A four-argument system call with the semantics described in this page was introduced in Linux 2.5.40. A fifth argument was added in Linux 2.5.70, and a sixth argument was added in Linux 2.6.7.

EXAMPLES
The program below demonstrates use of futexes in a program where a parent process and a child process use a pair of futexes located inside a shared anonymous mapping to synchronize access to a shared resource: the terminal. The two processes each write *nloops* (a command-line argument that defaults to 5 if omitted) messages to the terminal and employ a synchronization protocol that ensures that they alternate in writing messages. Upon running this program we see output such as the following:

```
$ ./futex_demo
Parent (18534) 0
Child  (18535) 0
Parent (18534) 1
Child  (18535) 1
Parent (18534) 2
Child  (18535) 2
Parent (18534) 3
Child  (18535) 3
Parent (18534) 4
Child  (18535) 4
```

Program source

```
/* futex_demo.c

   Usage: futex_demo [nloops]
                     (Default: 5)

   Demonstrate the use of futexes in a program where parent and child
   use a pair of futexes located inside a shared anonymous mapping to
   synchronize access to a shared resource: the terminal. The two
   processes each write 'num-loops' messages to the terminal and employ
   a synchronization protocol that ensures that they alternate in
   writing messages.
*/
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <stdint.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>

static uint32_t *futex1, *futex2, *iaddr;

static int
futex(uint32_t *uaddr, int futex_op, uint32_t val,
      const struct timespec *timeout, uint32_t *uaddr2, uint32_t val3)
{
    return syscall(SYS_futex, uaddr, futex_op, val,
                   timeout, uaddr2, val3);
}

/* Acquire the futex pointed to by 'futexp': wait for its value to
   become 1, and then set the value to 0. */

static void
fwait(uint32_t *futexp)
{
    long            s;
    const uint32_t  one = 1;

    /* atomic_compare_exchange_strong(ptr, oldval, newval)
       atomically performs the equivalent of:

           if (*ptr == *oldval)
               *ptr = newval;

       It returns true if the test yielded true and *ptr was updated. */

    while (1) {

        /* Is the futex available? */
        if (atomic_compare_exchange_strong(futexp, &one, 0))
            break;      /* Yes */

        /* Futex is not available; wait. */

        s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}

/* Release the futex pointed to by 'futexp': if the futex currently
   has the value 0, set its value to 1 and then wake any futex waiters,
   so that if the peer is blocked in fwait(), it can proceed. */

static void
fpost(uint32_t *futexp)
{
    long            s;
    const uint32_t  zero = 0;

    /* atomic_compare_exchange_strong() was described
```

```
           in comments above. */

        if (atomic_compare_exchange_strong(futexp, &zero, 1)) {
            s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
            if (s  == -1)
                err(EXIT_FAILURE, "futex-FUTEX_WAKE");
        }
    }

    int
    main(int argc, char *argv[])
    {
        pid_t         childPid;
        unsigned int  nloops;

        setbuf(stdout, NULL);

        nloops = (argc > 1) ? atoi(argv[1]) : 5;

        /* Create a shared anonymous mapping that will hold the futexes.
           Since the futexes are being shared between processes, we
           subsequently use the "shared" futex operations (i.e., not the
           ones suffixed "_PRIVATE"). */

        iaddr = mmap(NULL, sizeof(*iaddr) * 2, PROT_READ | PROT_WRITE,
                     MAP_ANONYMOUS | MAP_SHARED, -1, 0);
        if (iaddr == MAP_FAILED)
            err(EXIT_FAILURE, "mmap");

        futex1 = &iaddr[0];
        futex2 = &iaddr[1];

        *futex1 = 0;        /* State: unavailable */
        *futex2 = 1;        /* State: available */

        /* Create a child process that inherits the shared anonymous
           mapping. */

        childPid = fork();
        if (childPid == -1)
            err(EXIT_FAILURE, "fork");

        if (childPid == 0) {        /* Child */
            for (unsigned int j = 0; j < nloops; j++) {
                fwait(futex1);
                printf("Child  (%jd) %u\n", (intmax_t) getpid(), j);
                fpost(futex2);
            }

            exit(EXIT_SUCCESS);
        }

        /* Parent falls through to here. */

        for (unsigned int j = 0; j < nloops; j++) {
            fwait(futex2);
            printf("Parent (%jd) %u\n", (intmax_t) getpid(), j);
            fpost(futex1);
        }
```

```
        wait(NULL);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

get_robust_list(2), *restart_syscall*(2), *pthread_mutexattr_getprotocol*(3), *futex*(7), *sched*(7)

The following kernel source files:

- *Documentation/pi−futex.txt*
- *Documentation/futex−requeue−pi.txt*
- *Documentation/locking/rt−mutex.txt*
- *Documentation/locking/rt−mutex−design.txt*
- *Documentation/robust−futex−ABI.txt*

Franke, H., Russell, R., and Kirwood, M., 2002. *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux* (from proceedings of the Ottawa Linux Symposium 2002),

Hart, D., 2009. *A futex overview and update*,

Hart, D. and Guniguntala, D., 2009. *Requeue-PI: Making glibc Condvars PI-Aware* (from proceedings of the 2009 Real-Time Linux Workshop),

Drepper, U., 2011. *Futexes Are Tricky*,

Futex example library, futex−*.tar.bz2 at

NAME

futimesat − change timestamps of a file relative to a directory file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>          /* Definition of AT_* constants */
#include <sys/time.h>

[[deprecated]] int futimesat(int dirfd, const char *pathname,
                             const struct timeval times[2]);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

futimesat():
_GNU_SOURCE

DESCRIPTION

This system call is obsolete. Use *utimensat*(2) instead.

The **futimesat**() system call operates in exactly the same way as *utimes*(2), except for the differences described in this manual page.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by *utimes*(2) for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *utimes*(2)).

If *pathname* is absolute, then *dirfd* is ignored. (See *openat*(2) for an explanation of why the *dirfd* argument is useful.)

RETURN VALUE

On success, **futimesat**() returns a 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

The same errors that occur for *utimes*(2) can also occur for **futimesat**(). The following additional errors can occur for **futimesat**():

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

ENOTDIR
pathname is relative and *dirfd* is a file descriptor referring to a file other than a directory.

VERSIONS

glibc

If *pathname* is NULL, then the glibc **futimesat**() wrapper function updates the times for the file referred to by *dirfd*.

STANDARDS

None.

HISTORY

Linux 2.6.16, glibc 2.4.

It was implemented from a specification that was proposed for POSIX.1, but that specification was replaced by the one for *utimensat*(2).

A similar system call exists on Solaris.

NOTES

SEE ALSO

stat(2), *utimensat*(2), *utimes*(2), *futimes*(3), *path_resolution*(7)

NAME

get_kernel_syms − retrieve exported kernel and module symbols

SYNOPSIS

#include <linux/module.h>

[[deprecated]] int get_kernel_syms(struct kernel_sym ****table*);**

DESCRIPTION

Note: This system call is present only before Linux 2.6.

If *table* is NULL, **get_kernel_syms**() returns the number of symbols available for query. Otherwise, it fills in a table of structures:

```
struct kernel_sym {
    unsigned long value;
    char          name[60];
};
```

The symbols are interspersed with magic symbols of the form #*module-name* with the kernel having an empty name. The value associated with a symbol of this form is the address at which the module is loaded.

The symbols exported from each module follow their magic module tag and the modules are returned in the reverse of the order in which they were loaded.

RETURN VALUE

On success, returns the number of symbols copied to *table*. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

There is only one possible error return:

ENOSYS
get_kernel_syms() is not supported in this version of the kernel.

STANDARDS

Linux.

HISTORY

Removed in Linux 2.6.

This obsolete system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc versions before glibc 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it was sufficient to manually declare the interface in your code; alternatively, you could invoke the system call using *syscall*(2).

BUGS

There is no way to indicate the size of the buffer allocated for *table*. If symbols have been added to the kernel since the program queried for the symbol table size, memory will be corrupted.

The length of exported symbol names is limited to 59 characters.

Because of these limitations, this system call is deprecated in favor of *query_module*(2) (which is itself nowadays deprecated in favor of other interfaces described on its manual page).

SEE ALSO

create_module(2), *delete_module*(2), *init_module*(2), *query_module*(2)

NAME

get_mempolicy − retrieve NUMA memory policy for a thread

LIBRARY

NUMA (Non-Uniform Memory Access) policy library (*libnuma*, −*lnuma*)

SYNOPSIS

```
#include <numaif.h>

long get_mempolicy(int *mode,
                   unsigned long nodemask[(.maxnode + ULONG_WIDTH - 1)
                                          / ULONG_WIDTH],
                   unsigned long maxnode, void *addr,
                   unsigned long  flags);
```

DESCRIPTION

get_mempolicy() retrieves the NUMA policy of the calling thread or of a memory address, depending on the setting of *flags*.

A NUMA machine has different memory controllers with different distances to specific CPUs. The memory policy defines from which node memory is allocated for the thread.

If *flags* is specified as 0, then information about the calling thread's default policy (as set by *set_mempolicy*(2)) is returned, in the buffers pointed to by *mode* and *nodemask*. The value returned in these arguments may be used to restore the thread's policy to its state at the time of the call to **get_mempolicy**() using *set_mempolicy*(2). When *flags* is 0, *addr* must be specified as NULL.

If *flags* specifies **MPOL_F_MEMS_ALLOWED** (available since Linux 2.6.24), the *mode* argument is ignored and the set of nodes (memories) that the thread is allowed to specify in subsequent calls to *mbind*(2) or *set_mempolicy*(2) (in the absence of any *mode flags*) is returned in *nodemask*. It is not permitted to combine **MPOL_F_MEMS_ALLOWED** with either **MPOL_F_ADDR** or **MPOL_F_NODE**.

If *flags* specifies **MPOL_F_ADDR**, then information is returned about the policy governing the memory address given in *addr*. This policy may be different from the thread's default policy if *mbind*(2) or one of the helper functions described in *numa*(3) has been used to establish a policy for the memory range containing *addr*.

If the *mode* argument is not NULL, then **get_mempolicy**() will store the policy mode and any optional *mode flags* of the requested NUMA policy in the location pointed to by this argument. If *nodemask* is not NULL, then the nodemask associated with the policy will be stored in the location pointed to by this argument. *maxnode* specifies the number of node IDs that can be stored into *nodemask*—that is, the maximum node ID plus one. The value specified by *maxnode* is always rounded to a multiple of *sizeof(unsigned long)*8*.

If *flags* specifies both **MPOL_F_NODE** and **MPOL_F_ADDR**, **get_mempolicy**() will return the node ID of the node on which the address *addr* is allocated into the location pointed to by *mode*. If no page has yet been allocated for the specified address, **get_mempolicy**() will allocate a page as if the thread had performed a read (load) access to that address, and return the ID of the node where that page was allocated.

If *flags* specifies **MPOL_F_NODE**, but not **MPOL_F_ADDR**, and the thread's current policy is **MPOL_INTERLEAVE**, then **get_mempolicy**() will return in the location pointed to by a non-NULL *mode* argument, the node ID of the next node that will be used for interleaving of internal kernel pages allocated on behalf of the thread. These allocations include pages for memory-mapped files in process memory ranges mapped using the *mmap*(2) call with the **MAP_PRIVATE** flag for read accesses, and in memory ranges mapped with the **MAP_SHARED** flag for all accesses.

Other flag values are reserved.

For an overview of the possible policies see *set_mempolicy*(2).

RETURN VALUE

On success, **get_mempolicy**() returns 0; on error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
Part of all of the memory range specified by *nodemask* and *maxnode* points outside your accessible address space.

EINVAL
The value specified by *maxnode* is less than the number of node IDs supported by the system. Or *flags* specified values other than **MPOL_F_NODE** or **MPOL_F_ADDR**; or *flags* specified **MPOL_F_ADDR** and *addr* is NULL, or *flags* did not specify **MPOL_F_ADDR** and *addr* is not NULL. Or, *flags* specified **MPOL_F_NODE** but not **MPOL_F_ADDR** and the current thread policy is not **MPOL_INTERLEAVE**. Or, *flags* specified **MPOL_F_MEMS_ALLOWED** with either **MPOL_F_ADDR** or **MPOL_F_NODE**. (And there are other **EINVAL** cases.)

STANDARDS
Linux.

HISTORY
Linux 2.6.7.

NOTES
For information on library support, see *numa*(7).

SEE ALSO
getcpu(2), *mbind*(2), *mmap*(2), *set_mempolicy*(2), *numa*(3), *numa*(7), *numactl*(8)

NAME

get_robust_list, set_robust_list − get/set list of robust futexes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/futex.h>   /* Definition of struct robust_list_head */
#include <sys/syscall.h>   /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_get_robust_list, int pid,
             struct robust_list_head **head_ptr, size_t *len_ptr);
long syscall(SYS_set_robust_list,
             struct robust_list_head *head, size_t len);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

These system calls deal with per-thread robust futex lists. These lists are managed in user space: the kernel knows only about the location of the head of the list. A thread can inform the kernel of the location of its robust futex list using **set_robust_list**(). The address of a thread's robust futex list can be obtained using **get_robust_list**().

The purpose of the robust futex list is to ensure that if a thread accidentally fails to unlock a futex before terminating or calling *execve*(2), another thread that is waiting on that futex is notified that the former owner of the futex has died. This notification consists of two pieces: the **FUTEX_OWNER_DIED** bit is set in the futex word, and the kernel performs a *futex*(2) **FUTEX_WAKE** operation on one of the threads waiting on the futex.

The **get_robust_list**() system call returns the head of the robust futex list of the thread whose thread ID is specified in *pid*. If *pid* is 0, the head of the list for the calling thread is returned. The list head is stored in the location pointed to by *head_ptr*. The size of the object pointed to by **head_ptr* is stored in *len_ptr*.

Permission to employ **get_robust_list**() is governed by a ptrace access mode **PTRACE_MODE_READ_REALCREDS** check; see *ptrace*(2).

The **set_robust_list**() system call requests the kernel to record the head of the list of robust futexes owned by the calling thread. The *head* argument is the list head to record. The *len* argument should be *sizeof(*head)*.

RETURN VALUE

The **set_robust_list**() and **get_robust_list**() system calls return zero when the operation is successful, an error code otherwise.

ERRORS

The **set_robust_list**() system call can fail with the following error:

EINVAL
: *len* does not equal *sizeof(struct robust_list_head)*.

The **get_robust_list**() system call can fail with the following errors:

EFAULT
: The head of the robust futex list can't be stored at the location *head*.

EPERM
: The calling process does not have permission to see the robust futex list of the thread with the thread ID *pid*, and does not have the **CAP_SYS_PTRACE** capability.

ESRCH
: No thread with the thread ID *pid* could be found.

VERSIONS

These system calls were added in Linux 2.6.17.

NOTES

These system calls are not needed by normal applications.

A thread can have only one robust futex list; therefore applications that wish to use this functionality should use the robust mutexes provided by glibc.

In the initial implementation, a thread waiting on a futex was notified that the owner had died only if the owner terminated. Starting with Linux 2.6.28, notification was extended to include the case where the owner performs an *execve*(2).

The thread IDs mentioned in the main text are *kernel* thread IDs of the kind returned by *clone*(2) and *gettid*(2).

SEE ALSO

futex(2), *pthread_mutexattr_setrobust*(3)

Documentation/robust−futexes.txt and *Documentation/robust−futex−ABI.txt* in the Linux kernel source tree

NAME

getcpu − determine CPU and NUMA node on which the calling thread is running

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sched.h>

int getcpu(unsigned int *_Nullable cpu, unsigned int *_Nullable node);
```

DESCRIPTION

The **getcpu**() system call identifies the processor and node on which the calling thread or process is currently running and writes them into the integers pointed to by the *cpu* and *node* arguments. The processor is a unique small integer identifying a CPU. The node is a unique small identifier identifying a NUMA node. When either *cpu* or *node* is NULL nothing is written to the respective pointer.

The information placed in *cpu* is guaranteed to be current only at the time of the call: unless the CPU affinity has been fixed using *sched_setaffinity*(2), the kernel might change the CPU at any time. (Normally this does not happen because the scheduler tries to minimize movements between CPUs to keep caches hot, but it is possible.) The caller must allow for the possibility that the information returned in *cpu* and *node* is no longer current by the time the call returns.

RETURN VALUE

On success, 0 is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: Arguments point outside the calling process's address space.

STANDARDS

Linux.

HISTORY

Linux 2.6.19 (x86-64 and i386), glibc 2.29.

C library/kernel differences

The kernel system call has a third argument:

```
int getcpu(unsigned int *cpu, unsigned int *node,
           struct getcpu_cache *tcache);
```

The *tcache* argument is unused since Linux 2.6.24, and (when invoking the system call directly) should be specified as NULL, unless portability to Linux 2.6.23 or earlier is required.

In Linux 2.6.23 and earlier, if the *tcache* argument was non-NULL, then it specified a pointer to a caller-allocated buffer in thread-local storage that was used to provide a caching mechanism for **getcpu**(). Use of the cache could speed **getcpu**() calls, at the cost that there was a very small chance that the returned information would be out of date. The caching mechanism was considered to cause problems when migrating threads between CPUs, and so the argument is now ignored.

NOTES

Linux makes a best effort to make this call as fast as possible. (On some architectures, this is done via an implementation in the *vdso*(7).) The intention of **getcpu**() is to allow programs to make optimizations with per-CPU data or for NUMA optimization.

SEE ALSO

mbind(2), *sched_setaffinity*(2), *set_mempolicy*(2), *sched_getcpu*(3), *cpuset*(7), *vdso*(7)

NAME

getdents, getdents64 − get directory entries

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_getdents, unsigned int  fd, struct linux_dirent *dirp,
        unsigned int count);

#define _GNU_SOURCE           /* See feature_test_macros(7) */
#include <dirent.h>

ssize_t getdents64(int  fd, void dirp[.count], size_t count);
```

Note: glibc provides no wrapper for **getdents**(), necessitating the use of *syscall*(2).

Note: There is no definition of *struct linux_dirent* in glibc; see NOTES.

DESCRIPTION

These are not the interfaces you are interested in. Look at *readdir*(3) for the POSIX-conforming C library interface. This page documents the bare kernel system call interfaces.

getdents()

The system call **getdents**() reads several *linux_dirent* structures from the directory referred to by the open file descriptor *fd* into the buffer pointed to by *dirp*. The argument *count* specifies the size of that buffer.

The *linux_dirent* structure is declared as follows:

```
struct linux_dirent {
    unsigned long  d_ino;     /* Inode number */
    unsigned long  d_off;     /* Not an offset; see below */
    unsigned short d_reclen;  /* Length of this linux_dirent */
    char           d_name[];  /* Filename (null-terminated) */
                       /* length is actually (d_reclen - 2 -
                          offsetof(struct linux_dirent, d_name)) */
    /*
    char           pad;       // Zero padding byte
    char           d_type;    // File type (only since Linux
                              // 2.6.4); offset is (d_reclen - 1)
    */
}
```

d_ino is an inode number. *d_off* is a filesystem-specific value with no specific meaning to user space, though on older filesystems it used to be the distance from the start of the directory to the start of the next *linux_dirent*; see *readdir*(3). *d_reclen* is the size of this entire *linux_dirent*. *d_name* is a null-terminated filename.

d_type is a byte at the end of the structure that indicates the file type. It contains one of the following values (defined in <*dirent.h*>):

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

DT_REG This is a regular file.

DT_SOCK This is a UNIX domain socket.

DT_UNKNOWN
The file type is unknown.

The *d_type* field is implemented since Linux 2.6.4. It occupies a space that was previously a zero-filled padding byte in the *linux_dirent* structure. Thus, on kernels up to and including Linux 2.6.3, attempting to access this field always provides the value 0 (**DT_UNKNOWN**).

Currently, only some filesystems (among them: Btrfs, ext2, ext3, and ext4) have full support for returning the file type in *d_type*. All applications must properly handle a return of **DT_UNKNOWN**.

getdents64()
The original Linux **getdents**() system call did not handle large filesystems and large file offsets. Consequently, Linux 2.4 added **getdents64**(), with wider types for the *d_ino* and *d_off* fields. In addition, **getdents64**() supports an explicit *d_type* field.

The **getdents64**() system call is like **getdents**(), except that its second argument is a pointer to a buffer containing structures of the following type:

```
struct linux_dirent64 {
    ino64_t        d_ino;    /* 64-bit inode number */
    off64_t        d_off;    /* Not an offset; see getdents() */
    unsigned short d_reclen; /* Size of this dirent */
    unsigned char  d_type;   /* File type */
    char           d_name[]; /* Filename (null-terminated) */
};
```

RETURN VALUE
On success, the number of bytes read is returned. On end of directory, 0 is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS
EBADF
Invalid file descriptor *fd*.

EFAULT
Argument points outside the calling process's address space.

EINVAL
Result buffer is too small.

ENOENT
No such directory.

ENOTDIR
File descriptor does not refer to a directory.

STANDARDS
None.

HISTORY
SVr4.

getdents64()
glibc 2.30.

NOTES
glibc does not provide a wrapper for **getdents**(); call **getdents**() using *syscall*(2). In that case you will need to define the *linux_dirent* or *linux_dirent64* structure yourself.

Probably, you want to use *readdir*(3) instead of these system calls.

These calls supersede *readdir*(2).

EXAMPLES
The program below demonstrates the use of **getdents**(). The following output shows an example of what we see when running this program on an ext2 directory:

```
$ ./a.out /testfs/
-------------- nread=120 ---------------
inode#    file type  d_reclen  d_off   d_name
```

```
             2  directory    16         12  .
             2  directory    16         24  ..
            11  directory    24         44  lost+found
            12  regular      16         56  a
        228929  directory    16         68  sub
         16353  directory    16         80  sub2
        130817  directory    16       4096  sub3
```

Program source

```
#define _GNU_SOURCE
#include <dirent.h>     /* Defines DT_* constants */
#include <err.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>

struct linux_dirent {
    unsigned long  d_ino;
    off_t          d_off;
    unsigned short d_reclen;
    char           d_name[];
};

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int                 fd;
    char                d_type;
    char                buf[BUF_SIZE];
    long                nread;
    struct linux_dirent  *d;

    fd = open(argc > 1 ? argv[1] : ".", O_RDONLY | O_DIRECTORY);
    if (fd == -1)
        err(EXIT_FAILURE, "open");

    for (;;) {
        nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);
        if (nread == -1)
            err(EXIT_FAILURE, "getdents");

        if (nread == 0)
            break;

        printf("--------------- nread=%ld ---------------\n", nread);
        printf("inode#    file type  d_reclen  d_off   d_name\n");
        for (size_t bpos = 0; bpos < nread;) {
            d = (struct linux_dirent *) (buf + bpos);
            printf("%8lu  ", d->d_ino);
            d_type = *(buf + bpos + d->d_reclen - 1);
            printf("%-10s ", (d_type == DT_REG) ?  "regular" :
                             (d_type == DT_DIR) ?  "directory" :
                             (d_type == DT_FIFO) ? "FIFO" :
```

```
                                (d_type == DT_SOCK) ? "socket" :
                                (d_type == DT_LNK) ?  "symlink" :
                                (d_type == DT_BLK) ?  "block dev" :
                                (d_type == DT_CHR) ?  "char dev" : "???");
                    printf("%4d %10jd  %s\n", d->d_reclen,
                           (intmax_t) d->d_off, d->d_name);
                    bpos += d->d_reclen;
                }
            }

            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

readdir(2), *readdir*(3), *inode*(7)

NAME

getdomainname, setdomainname − get/set NIS domain name

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int getdomainname(char *name, size_t len);
int setdomainname(const char *name, size_t len);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getdomainname(), **setdomainname**():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

These functions are used to access or to change the NIS domain name of the host system. More precisely, they operate on the NIS domain name associated with the calling process's UTS namespace.

setdomainname() sets the domain name to the value given in the character array *name*. The *len* argument specifies the number of bytes in *name*. (Thus, *name* does not require a terminating null byte.)

getdomainname() returns the null-terminated domain name in the character array *name*, which has a length of *len* bytes. If the null-terminated domain name requires more than *len* bytes, **getdomainname**() returns the first *len* bytes (glibc) or gives an error (libc).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

setdomainname() can fail with the following errors:

EFAULT
name pointed outside of user address space.

EINVAL
len was negative or too large.

EPERM
The caller did not have the **CAP_SYS_ADMIN** capability in the user namespace associated with its UTS namespace (see *namespaces*(7)).

getdomainname() can fail with the following errors:

EINVAL
For **getdomainname**() under libc: *name* is NULL or *name* is longer than *len* bytes.

VERSIONS

On most Linux architectures (including x86), there is no **getdomainname**() system call; instead, glibc implements **getdomainname**() as a library function that returns a copy of the *domainname* field returned from a call to *uname*(2).

STANDARDS

None.

HISTORY

Since Linux 1.0, the limit on the length of a domain name, including the terminating null byte, is 64 bytes. In older kernels, it was 8 bytes.

SEE ALSO

gethostname(2), *sethostname*(2), *uname*(2), *uts_namespaces*(7)

NAME

getgid, getegid − get group identity

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

gid_t getgid(void);
gid_t getegid(void);
```

DESCRIPTION

getgid() returns the real group ID of the calling process.

getegid() returns the effective group ID of the calling process.

ERRORS

These functions are always successful and never modify *errno*.

VERSIONS

On Alpha, instead of a pair of **getgid**() and **getegid**() system calls, a single **getxgid**() system call is provided, which returns a pair of real and effective GIDs. The glibc **getgid**() and **getegid**() wrapper functions transparently deal with this. See *syscall*(2) for details regarding register mapping.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

The original Linux **getgid**() and **getegid**() system calls supported only 16-bit group IDs. Subsequently, Linux 2.4 added **getgid32**() and **getegid32**(), supporting 32-bit IDs. The glibc **getgid**() and **getegid**() wrapper functions transparently deal with the variations across kernel versions.

SEE ALSO

getresgid(2), *setgid*(2), *setregid*(2), *credentials*(7)

NAME

getgroups, setgroups − get/set list of supplementary group IDs

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int getgroups(int *size***, gid_t** *list***[]);**

#include <grp.h>

int setgroups(size_t *size***, const gid_t *_Nullable** *list***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setgroups():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

getgroups() returns the supplementary group IDs of the calling process in *list*. The argument *size* should be set to the maximum number of items that can be stored in the buffer pointed to by *list*. If the calling process is a member of more than *size* supplementary groups, then an error results.

It is unspecified whether the effective group ID of the calling process is included in the returned list. (Thus, an application should also call *getegid*(2) and add or remove the resulting value.)

If *size* is zero, *list* is not modified, but the total number of supplementary group IDs for the process is returned. This allows the caller to determine the size of a dynamically allocated *list* to be used in a further call to **getgroups**().

setgroups() sets the supplementary group IDs for the calling process. Appropriate privileges are required (see the description of the **EPERM** error, below). The *size* argument specifies the number of supplementary group IDs in the buffer pointed to by *list*. A process can drop all of its supplementary groups with the call:

```
setgroups(0, NULL);
```

RETURN VALUE

On success, **getgroups**() returns the number of supplementary group IDs. On error, −1 is returned, and *errno* is set to indicate the error.

On success, **setgroups**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
list has an invalid address.

getgroups() can additionally fail with the following error:

EINVAL
size is less than the number of supplementary group IDs, but is not zero.

setgroups() can additionally fail with the following errors:

EINVAL
size is greater than **NGROUPS_MAX** (32 before Linux 2.6.4; 65536 since Linux 2.6.4).

ENOMEM
Out of memory.

EPERM
The calling process has insufficient privilege (the caller does not have the **CAP_SETGID** capability in the user namespace in which it resides).

EPERM (since Linux 3.19)
The use of **setgroups**() is denied in this user namespace. See the description of */proc/*pid*/setgroups* in *user_namespaces*(7).

VERSIONS

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for *setgroups*()) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see *nptl*(7).

STANDARDS

getgroups()
: POSIX.1-2008.

setgroups()
: None.

HISTORY

getgroups()
: SVr4, 4.3BSD, POSIX.1-2001.

setgroups()
: SVr4, 4.3BSD. Since **setgroups**() requires privilege, it is not covered by POSIX.1.

The original Linux **getgroups**() system call supported only 16-bit group IDs. Subsequently, Linux 2.4 added **getgroups32**(), supporting 32-bit IDs. The glibc **getgroups**() wrapper function transparently deals with the variation across kernel versions.

NOTES

A process can have up to **NGROUPS_MAX** supplementary group IDs in addition to the effective group ID. The constant **NGROUPS_MAX** is defined in <*limits.h*>. The set of supplementary group IDs is inherited from the parent process, and preserved across an *execve*(2).

The maximum number of supplementary group IDs can be found at run time using *sysconf*(3):

```
long ngroups_max;
ngroups_max = sysconf(_SC_NGROUPS_MAX);
```

The maximum return value of **getgroups**() cannot be larger than one more than this value. Since Linux 2.6.4, the maximum number of supplementary group IDs is also exposed via the Linux-specific read-only file, */proc/sys/kernel/ngroups_max*.

SEE ALSO

getgid(2), *setgid*(2), *getgrouplist*(3), *group_member*(3), *initgroups*(3), *capabilities*(7), *credentials*(7)

NAME

gethostname, sethostname − get/set hostname

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int gethostname(char *name, size_t len);
int sethostname(const char *name, size_t len);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gethostname():
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200112L
|| /* glibc 2.19 and earlier */ _BSD_SOURCE

sethostname():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

These system calls are used to access or to change the system hostname. More precisely, they operate on the hostname associated with the calling process's UTS namespace.

sethostname() sets the hostname to the value given in the character array *name*. The *len* argument specifies the number of bytes in *name*. (Thus, *name* does not require a terminating null byte.)

gethostname() returns the null-terminated hostname in the character array *name*, which has a length of *len* bytes. If the null-terminated hostname is too large to fit, then the name is truncated, and no error is returned (but see NOTES below). POSIX.1 says that if such truncation occurs, then it is unspecified whether the returned buffer includes a terminating null byte.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
name is an invalid address.

EINVAL
len is negative or, for **sethostname**(), *len* is larger than the maximum allowed size.

ENAMETOOLONG
(glibc **gethostname**()) *len* is smaller than the actual size. (Before glibc 2.1, glibc uses **EINVAL** for this case.)

EPERM
For **sethostname**(), the caller did not have the **CAP_SYS_ADMIN** capability in the user namespace associated with its UTS namespace (see *namespaces*(7)).

VERSIONS

SUSv2 guarantees that "Host names are limited to 255 bytes". POSIX.1 guarantees that "Host names (not including the terminating null byte) are limited to **HOST_NAME_MAX** bytes". On Linux, **HOST_NAME_MAX** is defined with the value 64, which has been the limit since Linux 1.0 (earlier kernels imposed a limit of 8 bytes).

C library/kernel differences

The GNU C library does not employ the **gethostname**() system call; instead, it implements **gethostname**() as a library function that calls *uname*(2) and copies up to *len* bytes from the returned *nodename* field into *name*. Having performed the copy, the function then checks if the length of the *nodename* was greater than or equal to *len*, and if it is, then the function returns −1 with *errno* set to **ENAMETOOLONG**; in this case, a terminating null byte is not included in the returned *name*.

STANDARDS

gethostname()
POSIX.1-2008.

sethostname()
None.

HISTORY

SVr4, 4.4BSD (these interfaces first appeared in 4.2BSD). POSIX.1-2001 and POSIX.1-2008 specify **gethostname**() but not **sethostname**().

Versions of glibc before glibc 2.2 handle the case where the length of the *nodename* was greater than or equal to *len* differently: nothing is copied into *name* and the function returns −1 with *errno* set to **ENAMETOOLONG**.

SEE ALSO

hostname(1), *getdomainname*(2), *setdomainname*(2), *uname*(2), *uts_namespaces*(7)

NAME

getitimer, setitimer − get or set value of an interval timer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *restrict new_value,
              struct itimerval *_Nullable restrict old_value);
```

DESCRIPTION

These system calls provide access to interval timers, that is, timers that initially expire at some point in the future, and (optionally) at regular intervals after that. When a timer expires, a signal is generated for the calling process, and the timer is reset to the specified interval (if the interval is nonzero).

Three types of timers—specified via the *which* argument—are provided, each of which counts against a different clock and generates a different signal on timer expiration:

ITIMER_REAL
: This timer counts down in real (i.e., wall clock) time. At each expiration, a **SIGALRM** signal is generated.

ITIMER_VIRTUAL
: This timer counts down against the user-mode CPU time consumed by the process. (The measurement includes CPU time consumed by all threads in the process.) At each expiration, a **SIGVTALRM** signal is generated.

ITIMER_PROF
: This timer counts down against the total (i.e., both user and system) CPU time consumed by the process. (The measurement includes CPU time consumed by all threads in the process.) At each expiration, a **SIGPROF** signal is generated.

  In conjunction with **ITIMER_VIRTUAL**, this timer can be used to profile user and system CPU time consumed by the process.

A process has only one of each of the three types of timers.

Timer values are defined by the following structures:

```
struct itimerval {
    struct timeval it_interval; /* Interval for periodic timer */
    struct timeval it_value;    /* Time until next expiration */
};

struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

getitimer()

The function **getitimer**() places the current value of the timer specified by *which* in the buffer pointed to by *curr_value*.

The *it_value* substructure is populated with the amount of time remaining until the next expiration of the specified timer. This value changes as the timer counts down, and will be reset to *it_interval* when the timer expires. If both fields of *it_value* are zero, then this timer is currently disarmed (inactive).

The *it_interval* substructure is populated with the timer interval. If both fields of *it_interval* are zero, then this is a single-shot timer (i.e., it expires just once).

setitimer()

The function **setitimer**() arms or disarms the timer specified by *which*, by setting the timer to the value specified by *new_value*. If *old_value* is non-NULL, the buffer it points to is used to return the previous value of the timer (i.e., the same information that is returned by *getitimer*())

If either field in *new_value.it_value* is nonzero, then the timer is armed to initially expire at the

specified time. If both fields in *new_value.it_value* are zero, then the timer is disarmed.

The *new_value.it_interval* field specifies the new interval for the timer; if both of its subfields are zero, the timer is single-shot.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT

new_value, *old_value*, or *curr_value* is not valid a pointer.

EINVAL

which is not one of **ITIMER_REAL**, **ITIMER_VIRTUAL**, or **ITIMER_PROF**; or (since Linux 2.6.22) one of the *tv_usec* fields in the structure pointed to by *new_value* contains a value outside the range [0, 999999].

VERSIONS

The standards are silent on the meaning of the call:

```
setitimer(which, NULL, &old_value);
```

Many systems (Solaris, the BSDs, and perhaps others) treat this as equivalent to:

```
getitimer(which, &old_value);
```

In Linux, this is treated as being equivalent to a call in which the *new_value* fields are zero; that is, the timer is disabled. *Don't use this Linux misfeature*: it is nonportable and unnecessary.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD (this call first appeared in 4.2BSD). POSIX.1-2008 marks **getitimer**() and **setitimer**() obsolete, recommending the use of the POSIX timers API (**timer_gettime**(2), *timer_settime*(2), etc.) instead.

NOTES

Timers will never expire before the requested time, but may expire some (short) time afterward, which depends on the system timer resolution and on the system load; see *time*(7). (But see BUGS below.) If the timer expires while the process is active (always true for **ITIMER_VIRTUAL**), the signal will be delivered immediately when generated.

A child created via *fork*(2) does not inherit its parent's interval timers. Interval timers are preserved across an *execve*(2).

POSIX.1 leaves the interaction between **setitimer**() and the three interfaces *alarm*(2), *sleep*(3), and *usleep*(3) unspecified.

BUGS

The generation and delivery of a signal are distinct, and only one instance of each of the signals listed above may be pending for a process. Under very heavy loading, an **ITIMER_REAL** timer may expire before the signal from a previous expiration has been delivered. The second signal in such an event will be lost.

Before Linux 2.6.16, timer values are represented in jiffies. If a request is made set a timer with a value whose jiffies representation exceeds **MAX_SEC_IN_JIFFIES** (defined in *include/linux/jiffies.h*), then the timer is silently truncated to this ceiling value. On Linux/i386 (where, since Linux 2.6.13, the default jiffy is 0.004 seconds), this means that the ceiling value for a timer is approximately 99.42 days. Since Linux 2.6.16, the kernel uses a different internal representation for times, and this ceiling is removed.

On certain systems (including i386), Linux kernels before Linux 2.6.12 have a bug which will produce premature timer expirations of up to one jiffy under some circumstances. This bug is fixed in Linux 2.6.12.

POSIX.1-2001 says that **setitimer**() should fail if a *tv_usec* value is specified that is outside of the range [0, 999999]. However, up to and including Linux 2.6.21, Linux does not give an error, but instead silently adjusts the corresponding seconds value for the timer. From Linux 2.6.22 onward, this

nonconformance has been repaired: an improper *tv_usec* value results in an **EINVAL** error.

SEE ALSO

gettimeofday(2), *sigaction*(2), *signal*(2), *timer_create*(2), *timerfd_create*(2), *time*(7)

NAME

getpagesize − get memory page size

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int getpagesize(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpagesize():
Since glibc 2.20:
_DEFAULT_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)
glibc 2.12 to glibc 2.19:
_BSD_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)
Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

The function **getpagesize**() returns the number of bytes in a memory page, where "page" is a fixed-length block, the unit for memory allocation and file mapping performed by *mmap*(2).

STANDARDS

None.

HISTORY

This call first appeared in 4.2BSD. SVr4, 4.4BSD, SUSv2. In SUSv2 the **getpagesize**() call is labeled LEGACY, and in POSIX.1-2001 it has been dropped; HP-UX does not have this call.

NOTES

Portable applications should employ *sysconf(_SC_PAGESIZE)* instead of **getpagesize**():

```
#include <unistd.h>
long sz = sysconf(_SC_PAGESIZE);
```

(Most systems allow the synonym **_SC_PAGE_SIZE** for **_SC_PAGESIZE**.)

Whether **getpagesize**() is present as a Linux system call depends on the architecture. If it is, it returns the kernel symbol **PAGE_SIZE**, whose value depends on the architecture and machine model. Generally, one uses binaries that are dependent on the architecture but not on the machine model, in order to have a single binary distribution per architecture. This means that a user program should not find **PAGE_SIZE** at compile time from a header file, but use an actual system call, at least for those architectures (like sun4) where this dependency exists. Here glibc 2.0 fails because its **getpagesize**() returns a statically derived value, and does not use a system call. Things are OK in glibc 2.1.

SEE ALSO

mmap(2), *sysconf*(3)

NAME

getpeername − get name of connected peer socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict addrlen);
```

DESCRIPTION

getpeername() returns the address of the peer connected to the socket *sockfd*, in the buffer pointed to by *addr*. The *addrlen* argument should be initialized to indicate the amount of space pointed to by *addr*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
The argument *sockfd* is not a valid file descriptor.

EFAULT
The *addr* argument points to memory not in a valid part of the process address space.

EINVAL
addrlen is invalid (e.g., is negative).

ENOBUFS
Insufficient resources were available in the system to perform the operation.

ENOTCONN
The socket is not connected.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD (first appeared in 4.2BSD).

NOTES

For stream sockets, once a *connect*(2) has been performed, either socket can call **getpeername**() to obtain the address of the peer socket. On the other hand, datagram sockets are connectionless. Calling *connect*(2) on a datagram socket merely sets the peer address for outgoing datagrams sent with *write*(2) or *recv*(2). The caller of *connect*(2) can use **getpeername**() to obtain the peer address that it earlier set for the socket. However, the peer socket is unaware of this information, and calling **getpeername**() on the peer socket will return no useful information (unless a *connect*(2) call was also executed on the peer). Note also that the receiver of a datagram can obtain the address of the sender when using *recvfrom*(2).

SEE ALSO

accept(2), *bind*(2), *getsockname*(2), *ip*(7), *socket*(7), *unix*(7)

NAME

getpid, getppid − get process identification

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

DESCRIPTION

getpid() returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using **fork**(), or, if that process has already terminated, the ID of the process to which this process has been reparented (either *init*(1) or a "subreaper" process defined via the *prctl*(2) **PR_SET_CHILD_SUBREAPER** operation).

ERRORS

These functions are always successful.

VERSIONS

On Alpha, instead of a pair of **getpid**() and **getppid**() system calls, a single **getxpid**() system call is provided, which returns a pair of PID and parent PID. The glibc **getpid**() and **getppid**() wrapper functions transparently deal with this. See *syscall*(2) for details regarding register mapping.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD, SVr4.

C library/kernel differences

From glibc 2.3.4 up to and including glibc 2.24, the glibc wrapper function for **getpid**() cached PIDs, with the goal of avoiding additional system calls when a process calls **getpid**() repeatedly. Normally this caching was invisible, but its correct operation relied on support in the wrapper functions for *fork*(2), *vfork*(2), and *clone*(2): if an application bypassed the glibc wrappers for these system calls by using *syscall*(2), then a call to **getpid**() in the child would return the wrong value (to be precise: it would return the PID of the parent process). In addition, there were cases where **getpid**() could return the wrong value even when invoking *clone*(2) via the glibc wrapper function. (For a discussion of one such case, see BUGS in *clone*(2).) Furthermore, the complexity of the caching code had been the source of a few bugs within glibc over the years.

Because of the aforementioned problems, since glibc 2.25, the PID cache is removed: calls to **getpid**() always invoke the actual system call, rather than returning a cached value.

NOTES

If the caller's parent is in a different PID namespace (see *pid_namespaces*(7)), **getppid**() returns 0.

From a kernel perspective, the PID (which is shared by all of the threads in a multithreaded process) is sometimes also known as the thread group ID (TGID). This contrasts with the kernel thread ID (TID), which is unique for each thread. For further details, see *gettid*(2) and the discussion of the **CLONE_THREAD** flag in *clone*(2).

SEE ALSO

clone(2), *fork*(2), *gettid*(2), *kill*(2), *exec*(3), *mkstemp*(3), *tempnam*(3), *tmpfile*(3), *tmpnam*(3), *credentials*(7), *pid_namespaces*(7)

NAME

getpriority, setpriority − get/set program scheduling priority

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority**() call and set with the **setpriority**() call. The process attribute dealt with by these system calls is the same attribute (also known as the "nice" value) that is dealt with by *nice*(2).

The value *which* is one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, and *who* is interpreted relative to *which* (a process identifier for **PRIO_PROCESS**, process group identifier for **PRIO_PGRP**, and a user ID for **PRIO_USER**). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

The *prio* argument is a value in the range −20 to 19 (but see NOTES below), with −20 being the highest priority and 19 being the lowest priority. Attempts to set a priority outside this range are silently clamped to the range. The default priority is 0; lower values give a process a higher scheduling priority.

The **getpriority**() call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority**() call sets the priorities of all of the specified processes to the specified value.

Traditionally, only a privileged process could lower the nice value (i.e., set a higher priority). However, since Linux 2.6.12, an unprivileged process can decrease the nice value of a target process that has a suitable **RLIMIT_NICE** soft limit; see *getrlimit*(2) for details.

RETURN VALUE

On success, **getpriority**() returns the calling thread's nice value, which may be a negative number. On error, it returns −1 and sets *errno* to indicate the error.

Since a successful call to **getpriority**() can legitimately return the value −1, it is necessary to clear *errno* prior to the call, then check *errno* afterward to determine if −1 is an error or a legitimate value.

setpriority() returns 0 on success. On failure, it returns −1 and sets *errno* to indicate the error.

ERRORS

EACCES
: The caller attempted to set a lower nice value (i.e., a higher process priority), but did not have the required privilege (on Linux: did not have the **CAP_SYS_NICE** capability).

EINVAL
: *which* was not one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**.

EPERM
: A process was located, but its effective user ID did not match either the effective or the real user ID of the caller, and was not privileged (on Linux: did not have the **CAP_SYS_NICE** capability). But see NOTES below.

ESRCH
: No process was located using the *which* and *who* values specified.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD (these interfaces first appeared in 4.2BSD).

NOTES

For further details on the nice value, see *sched*(7).

Note: the addition of the "autogroup" feature in Linux 2.6.38 means that the nice value no longer has

its traditional effect in many circumstances. For details, see *sched*(7).

A child created by *fork*(2) inherits its parent's nice value. The nice value is preserved across *execve*(2).

The details on the condition for **EPERM** depend on the system. The above description is what POSIX.1-2001 says, and seems to be followed on all System V-like systems. Linux kernels before Linux 2.6.12 required the real or effective user ID of the caller to match the real user of the process *who* (instead of its effective user ID). Linux 2.6.12 and later require the effective user ID of the caller to match the real or effective user ID of the process *who*. All BSD-like systems (SunOS 4.1.3, Ultrix 4.2, 4.3BSD, FreeBSD 4.3, OpenBSD-2.5, ...) behave in the same manner as Linux 2.6.12 and later.

C library/kernel differences

The getpriority system call returns nice values translated to the range 40..1, since a negative return value would be interpreted as an error. The glibc wrapper function for **getpriority**() translates the value back according to the formula *unice = 20 − knice* (thus, the 40..1 range returned by the kernel corresponds to the range −20..19 as seen by user space).

BUGS

According to POSIX, the nice value is a per-process setting. However, under the current Linux/NPTL implementation of POSIX threads, the nice value is a per-thread attribute: different threads in the same process can have different nice values. Portable applications should avoid relying on the Linux behavior, which may be made standards conformant in the future.

SEE ALSO

nice(1), *renice*(1), *fork*(2), *capabilities*(7), *sched*(7)

Documentation/scheduler/sched−nice−design.txt in the Linux kernel source tree (since Linux 2.6.23)

NAME

getrandom − obtain a series of random bytes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/random.h>

ssize_t getrandom(void *buf* **[.***buflen***], size_t** *buflen***, unsigned int** *flags***);**

DESCRIPTION

The **getrandom**() system call fills the buffer pointed to by *buf* with up to *buflen* random bytes. These bytes can be used to seed user-space random number generators or for cryptographic purposes.

By default, **getrandom**() draws entropy from the *urandom* source (i.e., the same source as the */dev/urandom* device). This behavior can be changed via the *flags* argument.

If the *urandom* source has been initialized, reads of up to 256 bytes will always return as many bytes as requested and will not be interrupted by signals. No such guarantees apply for larger buffer sizes. For example, if the call is interrupted by a signal handler, it may return a partially filled buffer, or fail with the error **EINTR**.

If the *urandom* source has not yet been initialized, then **getrandom**() will block, unless **GRND_NONBLOCK** is specified in *flags*.

The *flags* argument is a bit mask that can contain zero or more of the following values ORed together:

GRND_RANDOM
: If this bit is set, then random bytes are drawn from the *random* source (i.e., the same source as the */dev/random* device) instead of the *urandom* source. The *random* source is limited based on the entropy that can be obtained from environmental noise. If the number of available bytes in the *random* source is less than requested in *buflen*, the call returns just the available random bytes. If no random bytes are available, the behavior depends on the presence of **GRND_NONBLOCK** in the *flags* argument.

GRND_NONBLOCK
: By default, when reading from the *random* source, **getrandom**() blocks if no random bytes are available, and when reading from the *urandom* source, it blocks if the entropy pool has not yet been initialized. If the **GRND_NONBLOCK** flag is set, then **getrandom**() does not block in these cases, but instead immediately returns −1 with *errno* set to **EAGAIN**.

RETURN VALUE

On success, **getrandom**() returns the number of bytes that were copied to the buffer *buf*. This may be less than the number of bytes requested via *buflen* if either **GRND_RANDOM** was specified in *flags* and insufficient entropy was present in the *random* source or the system call was interrupted by a signal.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: The requested entropy was not available, and **getrandom**() would have blocked if the **GRND_NONBLOCK** flag was not set.

EFAULT
: The address referred to by *buf* is outside the accessible address space.

EINTR
: The call was interrupted by a signal handler; see the description of how interrupted *read*(2) calls on "slow" devices are handled with and without the **SA_RESTART** flag in the *signal*(7) man page.

EINVAL
: An invalid flag was specified in *flags*.

ENOSYS
: The glibc wrapper function for **getrandom**() determined that the underlying kernel does not implement this system call.

STANDARDS

Linux.

HISTORY

Linux 3.17, glibc 2.25.

NOTES

For an overview and comparison of the various interfaces that can be used to obtain randomness, see *random*(7).

Unlike */dev/random* and */dev/urandom*, **getrandom**() does not involve the use of pathnames or file descriptors. Thus, **getrandom**() can be useful in cases where *chroot*(2) makes */dev* pathnames invisible, and where an application (e.g., a daemon during start-up) closes a file descriptor for one of these files that was opened by a library.

Maximum number of bytes returned

As of Linux 3.19 the following limits apply:

- When reading from the *urandom* source, a maximum of 32Mi-1 bytes is returned by a single call to **getrandom**() on systems where *int* has a size of 32 bits.
- When reading from the *random* source, a maximum of 512 bytes is returned.

Interruption by a signal handler

When reading from the *urandom* source (**GRND_RANDOM** is not set), **getrandom**() will block until the entropy pool has been initialized (unless the **GRND_NONBLOCK** flag was specified). If a request is made to read a large number of bytes (more than 256), **getrandom**() will block until those bytes have been generated and transferred from kernel memory to *buf*. When reading from the *random* source (**GRND_RANDOM** is set), **getrandom**() will block until some random bytes become available (unless the **GRND_NONBLOCK** flag was specified).

The behavior when a call to **getrandom**() that is blocked while reading from the *urandom* source is interrupted by a signal handler depends on the initialization state of the entropy buffer and on the request size, *buflen*. If the entropy is not yet initialized, then the call fails with the **EINTR** error. If the entropy pool has been initialized and the request size is large (*buflen* > 256), the call either succeeds, returning a partially filled buffer, or fails with the error **EINTR**. If the entropy pool has been initialized and the request size is small (*buflen* <= 256), then **getrandom**() will not fail with **EINTR**. Instead, it will return all of the bytes that have been requested.

When reading from the *random* source, blocking requests of any size can be interrupted by a signal handler (the call fails with the error **EINTR**).

Using **getrandom**() to read small buffers (<= 256 bytes) from the *urandom* source is the preferred mode of usage.

The special treatment of small values of *buflen* was designed for compatibility with OpenBSD's *getentropy*(3), which is nowadays supported by glibc.

The user of **getrandom**() *must* always check the return value, to determine whether either an error occurred or fewer bytes than requested were returned. In the case where **GRND_RANDOM** is not specified and *buflen* is less than or equal to 256, a return of fewer bytes than requested should never happen, but the careful programmer will check for this anyway!

BUGS

As of Linux 3.19, the following bug exists:

- Depending on CPU load, **getrandom**() does not react to interrupts before reading all bytes requested.

SEE ALSO

getentropy(3), *random*(4), *urandom*(4), *random*(7), *signal*(7)

NAME

getresuid, getresgid − get real, effective, and saved user/group IDs

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>

int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

DESCRIPTION

getresuid() returns the real UID, the effective UID, and the saved set-user-ID of the calling process, in the arguments *ruid*, *euid*, and *suid*, respectively. **getresgid**() performs the analogous task for the process's group IDs.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: One of the arguments specified an address outside the calling program's address space.

STANDARDS

None. These calls also appear on HP-UX and some of the BSDs.

HISTORY

Linux 2.1.44, glibc 2.3.2.

The original Linux **getresuid**() and **getresgid**() system calls supported only 16-bit user and group IDs. Subsequently, Linux 2.4 added **getresuid32**() and **getresgid32**(), supporting 32-bit IDs. The glibc **getresuid**() and **getresgid**() wrapper functions transparently deal with the variations across kernel versions.

SEE ALSO

getuid(2), *setresuid*(2), *setreuid*(2), *setuid*(2), *credentials*(7)

NAME

getrlimit, setrlimit, prlimit − get/set resource limits

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource,
        const struct rlimit *_Nullable new_limit,
        struct rlimit *_Nullable old_limit);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

prlimit():
_GNU_SOURCE

DESCRIPTION

The **getrlimit**() and **setrlimit**() system calls get and set resource limits. Each resource has an associated soft and hard limit, as defined by the *rlimit* structure:

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may set only its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the **CAP_SYS_RESOURCE** capability in the initial user namespace) may make arbitrary changes to either limit value.

The value **RLIM_INFINITY** denotes no limit on a resource (both in the structure returned by **getrlimit**() and in the structure passed to *setrlimit*())

The *resource* argument must be one of:

RLIMIT_AS
: This is the maximum size of the process's virtual memory (address space). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to *brk*(2), *mmap*(2), and *mremap*(2), which fail with the error **ENOMEM** upon exceeding this limit. In addition, automatic stack expansion fails (and generates a **SIGSEGV** that kills the process if no alternate stack has been made available via *sigaltstack*(2)). Since the value is a *long*, on machines with a 32-bit *long* either this limit is at most 2 GiB, or this resource is unlimited.

RLIMIT_CORE
: This is the maximum size of a *core* file (see *core*(5)) in bytes that the process may dump. When 0 no core dump files are created. When nonzero, larger dumps are truncated to this size.

RLIMIT_CPU
: This is a limit, in seconds, on the amount of CPU time that the process can consume. When the process reaches the soft limit, it is sent a **SIGXCPU** signal. The default action for this signal is to terminate the process. However, the signal can be caught, and the handler can return control to the main program. If the process continues to consume CPU time, it will be sent **SIGXCPU** once per second until the hard limit is reached, at which time it is sent **SIGKILL**. (This latter point describes Linux behavior. Implementations vary in how they treat processes which continue to consume CPU time after reaching the soft limit. Portable applications that need to catch this signal should perform an orderly termination upon first receipt of **SIGXCPU**.)

RLIMIT_DATA
: This is the maximum size of the process's data segment (initialized data, uninitialized data, and heap). The limit is specified in bytes, and is rounded down to the system page size. This

limit affects calls to *brk*(2), *sbrk*(2), and (since Linux 4.7) *mmap*(2), which fail with the error **ENOMEM** upon encountering the soft limit of this resource.

RLIMIT_FSIZE
This is the maximum size in bytes of files that the process may create. Attempts to extend a file beyond this limit result in delivery of a **SIGXFSZ** signal. By default, this signal terminates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., *write*(2), *truncate*(2)) fails with the error **EFBIG**.

RLIMIT_LOCKS (Linux 2.4.0 to Linux 2.4.24)
This is a limit on the combined number of *flock*(2) locks and *fcntl*(2) leases that this process may establish.

RLIMIT_MEMLOCK
This is the maximum number of bytes of memory that may be locked into RAM. This limit is in effect rounded down to the nearest multiple of the system page size. This limit affects *mlock*(2), *mlockall*(2), and the *mmap*(2) **MAP_LOCKED** operation. Since Linux 2.6.9, it also affects the *shmctl*(2) **SHM_LOCK** operation, where it sets a maximum on the total bytes in shared memory segments (see *shmget*(2)) that may be locked by the real user ID of the calling process. The *shmctl*(2) **SHM_LOCK** locks are accounted for separately from the per-process memory locks established by *mlock*(2), *mlockall*(2), and *mmap*(2) **MAP_LOCKED**; a process can lock bytes up to this limit in each of these two categories.

Before Linux 2.6.9, this limit controlled the amount of memory that could be locked by a privileged process. Since Linux 2.6.9, no limits are placed on the amount of memory that a privileged process may lock, and this limit instead governs the amount of memory that an unprivileged process may lock.

RLIMIT_MSGQUEUE (since Linux 2.6.8)
This is a limit on the number of bytes that can be allocated for POSIX message queues for the real user ID of the calling process. This limit is enforced for *mq_open*(3). Each message queue that the user creates counts (until it is removed) against this limit according to the formula:

Since Linux 3.5:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg) +
        MIN(attr.mq_maxmsg, MQ_PRIO_MAX) *
              sizeof(struct posix_msg_tree_node)+
                        /* For overhead */
        attr.mq_maxmsg * attr.mq_msgsize;
                        /* For message data */
```

Linux 3.4 and earlier:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
                        /* For overhead */
        attr.mq_maxmsg * attr.mq_msgsize;
                        /* For message data */
```

where *attr* is the *mq_attr* structure specified as the fourth argument to *mq_open*(3), and the *msg_msg* and *posix_msg_tree_node* structures are kernel-internal structures.

The "overhead" addend in the formula accounts for overhead bytes required by the implementation and ensures that the user cannot create an unlimited number of zero-length messages (such messages nevertheless each consume some system memory for bookkeeping overhead).

RLIMIT_NICE (since Linux 2.6.12, but see BUGS below)
This specifies a ceiling to which the process's nice value can be raised using *setpriority*(2) or *nice*(2). The actual ceiling for the nice value is calculated as *20 − rlim_cur*. The useful range for this limit is thus from 1 (corresponding to a nice value of 19) to 40 (corresponding to a nice value of −20). This unusual choice of range was necessary because negative numbers cannot be specified as resource limit values, since they typically have special meanings. For example, **RLIM_INFINITY** typically is the same as −1. For more detail on the nice value, see *sched*(7).

RLIMIT_NOFILE
This specifies a value one greater than the maximum file descriptor number that can be opened by this process. Attempts (**open**(2), *pipe*(2), *dup*(2), etc.) to exceed this limit yield the error **EMFILE**. (Historically, this limit was named **RLIMIT_OFILE** on BSD.)

Since Linux 4.5, this limit also defines the maximum number of file descriptors that an unprivileged process (one without the **CAP_SYS_RESOURCE** capability) may have "in flight" to other processes, by being passed across UNIX domain sockets. This limit applies to the *sendmsg*(2) system call. For further details, see *unix*(7).

RLIMIT_NPROC
This is a limit on the number of extant process (or, more precisely on Linux, threads) for the real user ID of the calling process. So long as the current number of processes belonging to this process's real user ID is greater than or equal to this limit, *fork*(2) fails with the error **EAGAIN**.

The **RLIMIT_NPROC** limit is not enforced for processes that have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability, or run with real user ID 0.

RLIMIT_RSS
This is a limit (in bytes) on the process's resident set (the number of virtual pages resident in RAM). This limit has effect only in Linux 2.4.x, x < 30, and there affects only calls to *madvise*(2) specifying **MADV_WILLNEED**.

RLIMIT_RTPRIO (since Linux 2.6.12, but see BUGS)
This specifies a ceiling on the real-time priority that may be set for this process using *sched_setscheduler*(2) and *sched_setparam*(2).

For further details on real-time scheduling policies, see *sched*(7)

RLIMIT_RTTIME (since Linux 2.6.25)
This is a limit (in microseconds) on the amount of CPU time that a process scheduled under a real-time scheduling policy may consume without making a blocking system call. For the purpose of this limit, each time a process makes a blocking system call, the count of its consumed CPU time is reset to zero. The CPU time count is not reset if the process continues trying to use the CPU but is preempted, its time slice expires, or it calls *sched_yield*(2).

Upon reaching the soft limit, the process is sent a **SIGXCPU** signal. If the process catches or ignores this signal and continues consuming CPU time, then **SIGXCPU** will be generated once each second until the hard limit is reached, at which point the process is sent a **SIGKILL** signal.

The intended use of this limit is to stop a runaway real-time process from locking up the system.

For further details on real-time scheduling policies, see *sched*(7)

RLIMIT_SIGPENDING (since Linux 2.6.8)
This is a limit on the number of signals that may be queued for the real user ID of the calling process. Both standard and real-time signals are counted for the purpose of checking this limit. However, the limit is enforced only for *sigqueue*(3); it is always possible to use *kill*(2) to queue one instance of any of the signals that are not already queued to the process.

RLIMIT_STACK
This is the maximum size of the process stack, in bytes. Upon reaching this limit, a **SIGSEGV** signal is generated. To handle this signal, a process must employ an alternate signal stack (**sigaltstack**(2)).

Since Linux 2.6.23, this limit also determines the amount of space used for the process's command-line arguments and environment variables; for details, see *execve*(2).

prlimit()
The Linux-specific **prlimit**() system call combines and extends the functionality of **setrlimit**() and **getrlimit**(). It can be used to both set and get the resource limits of an arbitrary process.

The *resource* argument has the same meaning as for **setrlimit**() and **getrlimit**().

If the *new_limit* argument is not NULL, then the *rlimit* structure to which it points is used to set new

values for the soft and hard limits for *resource*. If the *old_limit* argument is not NULL, then a successful call to **prlimit**() places the previous soft and hard limits for *resource* in the *rlimit* structure pointed to by *old_limit*.

The *pid* argument specifies the ID of the process on which the call is to operate. If *pid* is 0, then the call applies to the calling process. To set or get the resources of a process other than itself, the caller must have the **CAP_SYS_RESOURCE** capability in the user namespace of the process whose resource limits are being changed, or the real, effective, and saved set user IDs of the target process must match the real user ID of the caller *and* the real, effective, and saved set group IDs of the target process must match the real group ID of the caller.

RETURN VALUE

On success, these system calls return 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
A pointer argument points to a location outside the accessible address space.

EINVAL
The value specified in *resource* is not valid; or, for **setrlimit**() or **prlimit**(): *rlim->rlim_cur* was greater than *rlim->rlim_max*.

EPERM
An unprivileged process tried to raise the hard limit; the **CAP_SYS_RESOURCE** capability is required to do this.

EPERM
The caller tried to increase the hard **RLIMIT_NOFILE** limit above the maximum defined by */proc/sys/fs/nr_open* (see *proc*(5))

EPERM
(**prlimit**()) The calling process did not have permission to set limits for the process specified by *pid*.

ESRCH
Could not find a process with the ID specified in *pid*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getrlimit(), **setrlimit**(), **prlimit**()	Thread safety	MT-Safe

STANDARDS

getrlimit()
setrlimit()
POSIX.1-2008.

prlimit()
Linux.

RLIMIT_MEMLOCK and **RLIMIT_NPROC** derive from BSD and are not specified in POSIX.1; they are present on the BSDs and Linux, but on few other implementations. **RLIMIT_RSS** derives from BSD and is not specified in POSIX.1; it is nevertheless present on most implementations. **RLIMIT_MSGQUEUE**, **RLIMIT_NICE**, **RLIMIT_RTPRIO**, **RLIMIT_RTTIME**, and **RLIMIT_SIGPENDING** are Linux-specific.

HISTORY

getrlimit()
setrlimit()
POSIX.1-2001, SVr4, 4.3BSD.

prlimit()
Linux 2.6.36, glibc 2.13.

NOTES

A child process created via *fork*(2) inherits its parent's resource limits. Resource limits are preserved across *execve*(2).

Resource limits are per-process attributes that are shared by all of the threads in a process.

Lowering the soft limit for a resource below the process's current consumption of that resource will succeed (but will prevent the process from further increasing its consumption of the resource).

One can set the resource limits of the shell using the built-in *ulimit* command (*limit* in *csh*(1)). The shell's resource limits are inherited by the processes that it creates to execute commands.

Since Linux 2.6.24, the resource limits of any process can be inspected via */proc/*pid*/limits*; see *proc*(5).

Ancient systems provided a **vlimit**() function with a similar purpose to **setrlimit**(). For backward compatibility, glibc also provides **vlimit**(). All new applications should be written using **setrlimit**().

C library/kernel ABI differences

Since glibc 2.13, the glibc **getrlimit**() and **setrlimit**() wrapper functions no longer invoke the corresponding system calls, but instead employ **prlimit**(), for the reasons described in BUGS.

The name of the glibc wrapper function is **prlimit**(); the underlying system call is **prlimit64**().

BUGS

In older Linux kernels, the **SIGXCPU** and **SIGKILL** signals delivered when a process encountered the soft and hard **RLIMIT_CPU** limits were delivered one (CPU) second later than they should have been. This was fixed in Linux 2.6.8.

In Linux 2.6.x kernels before Linux 2.6.17, a **RLIMIT_CPU** limit of 0 is wrongly treated as "no limit" (like **RLIM_INFINITY**). Since Linux 2.6.17, setting a limit of 0 does have an effect, but is actually treated as a limit of 1 second.

A kernel bug means that **RLIMIT_RTPRIO** does not work in Linux 2.6.12; the problem is fixed in Linux 2.6.13.

In Linux 2.6.12, there was an off-by-one mismatch between the priority ranges returned by *getpriority*(2) and **RLIMIT_NICE**. This had the effect that the actual ceiling for the nice value was calculated as *19 − rlim_cur*. This was fixed in Linux 2.6.13.

Since Linux 2.6.12, if a process reaches its soft **RLIMIT_CPU** limit and has a handler installed for **SIGXCPU**, then, in addition to invoking the signal handler, the kernel increases the soft limit by one second. This behavior repeats if the process continues to consume CPU time, until the hard limit is reached, at which point the process is killed. Other implementations do not change the **RLIMIT_CPU** soft limit in this manner, and the Linux behavior is probably not standards conformant; portable applications should avoid relying on this Linux-specific behavior. The Linux-specific **RLIMIT_RTTIME** limit exhibits the same behavior when the soft limit is encountered.

Kernels before Linux 2.4.22 did not diagnose the error **EINVAL** for **setrlimit**() when *rlim->rlim_cur* was greater than *rlim->rlim_max*.

Linux doesn't return an error when an attempt to set **RLIMIT_CPU** has failed, for compatibility reasons.

Representation of "large" resource limit values on 32-bit platforms

The glibc **getrlimit**() and **setrlimit**() wrapper functions use a 64-bit *rlim_t* data type, even on 32-bit platforms. However, the *rlim_t* data type used in the **getrlimit**() and **setrlimit**() system calls is a (32-bit) *unsigned long*. Furthermore, in Linux, the kernel represents resource limits on 32-bit platforms as *unsigned long*. However, a 32-bit data type is not wide enough. The most pertinent limit here is **RLIMIT_FSIZE**, which specifies the maximum size to which a file can grow: to be useful, this limit must be represented using a type that is as wide as the type used to represent file offsets—that is, as wide as a 64-bit **off_t** (assuming a program compiled with *_FILE_OFFSET_BITS=64*).

To work around this kernel limitation, if a program tried to set a resource limit to a value larger than can be represented in a 32-bit *unsigned long*, then the glibc **setrlimit**() wrapper function silently converted the limit value to **RLIM_INFINITY**. In other words, the requested resource limit setting was silently ignored.

Since glibc 2.13, glibc works around the limitations of the **getrlimit**() and **setrlimit**() system calls by implementing **setrlimit**() and **getrlimit**() as wrapper functions that call **prlimit**().

EXAMPLES

The program below demonstrates the use of **prlimit**().

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <err.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    pid_t          pid;
    struct rlimit  old, new;
    struct rlimit  *newp;

    if (!(argc == 2 || argc == 4)) {
        fprintf(stderr, "Usage: %s <pid> [<new-soft-limit> "
                "<new-hard-limit>]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    pid = atoi(argv[1]);        /* PID of target process */

    newp = NULL;
    if (argc == 4) {
        new.rlim_cur = atoi(argv[2]);
        new.rlim_max = atoi(argv[3]);
        newp = &new;
    }

    /* Set CPU time limit of target process; retrieve and display
       previous limit */

    if (prlimit(pid, RLIMIT_CPU, newp, &old) == -1)
        err(EXIT_FAILURE, "prlimit-1");
    printf("Previous limits: soft=%jd; hard=%jd\n",
           (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

    /* Retrieve and display new CPU time limit */

    if (prlimit(pid, RLIMIT_CPU, NULL, &old) == -1)
        err(EXIT_FAILURE, "prlimit-2");
    printf("New limits: soft=%jd; hard=%jd\n",
           (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

prlimit(1), *dup*(2), *fcntl*(2), *fork*(2), *getrusage*(2), *mlock*(2), *mmap*(2), *open*(2), *quotactl*(2), *sbrk*(2), *shmctl*(2), *malloc*(3), *sigqueue*(3), *ulimit*(3), *core*(5), *capabilities*(7), *cgroups*(7), *credentials*(7), *signal*(7)

NAME

getrusage − get resource usage

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/resource.h>

int getrusage(int *who***, struct rusage *****usage***);**

DESCRIPTION

getrusage() returns resource usage measures for *who*, which can be one of the following:

RUSAGE_SELF
: Return resource usage statistics for the calling process, which is the sum of resources used by all threads in the process.

RUSAGE_CHILDREN
: Return resource usage statistics for all children of the calling process that have terminated and been waited for. These statistics will include the resources used by grandchildren, and further removed descendants, if all of the intervening descendants waited on their terminated children.

RUSAGE_THREAD (since Linux 2.6.26)
: Return resource usage statistics for the calling thread. The **_GNU_SOURCE** feature test macro must be defined (before including *any* header file) in order to obtain the definition of this constant from <*sys/resource.h*>.

The resource usages are returned in the structure pointed to by *usage*, which has the following form:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Not all fields are completed; unmaintained fields are set to zero by the kernel. (The unmaintained fields are provided for compatibility with other systems, and because they may one day be supported on Linux.) The fields are interpreted as follows:

ru_utime
: This is the total amount of time spent executing in user mode, expressed in a *timeval* structure (seconds plus microseconds).

ru_stime
: This is the total amount of time spent executing in kernel mode, expressed in a *timeval* structure (seconds plus microseconds).

ru_maxrss (since Linux 2.6.32)
: This is the maximum resident set size used (in kilobytes). For **RUSAGE_CHILDREN**, this is the resident set size of the largest child, not the maximum resident set size of the process tree.

ru_ixrss (unmaintained)
This field is currently unused on Linux.

ru_idrss (unmaintained)
This field is currently unused on Linux.

ru_isrss (unmaintained)
This field is currently unused on Linux.

ru_minflt
The number of page faults serviced without any I/O activity; here I/O activity is avoided by "reclaiming" a page frame from the list of pages awaiting reallocation.

ru_majflt
The number of page faults serviced that required I/O activity.

ru_nswap (unmaintained)
This field is currently unused on Linux.

ru_inblock (since Linux 2.6.22)
The number of times the filesystem had to perform input.

ru_oublock (since Linux 2.6.22)
The number of times the filesystem had to perform output.

ru_msgsnd (unmaintained)
This field is currently unused on Linux.

ru_msgrcv (unmaintained)
This field is currently unused on Linux.

ru_nsignals (unmaintained)
This field is currently unused on Linux.

ru_nvcsw (since Linux 2.6)
The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw (since Linux 2.6)
The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
usage points outside the accessible address space.

EINVAL
who is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getrusage()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

POSIX.1 specifies **getrusage**(), but specifies only the fields *ru_utime* and *ru_stime*.

RUSAGE_THREAD is Linux-specific.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

Before Linux 2.6.9, if the disposition of **SIGCHLD** is set to **SIG_IGN** then the resource usages of child processes are automatically included in the value returned by **RUSAGE_CHILDREN**, although POSIX.1-2001 explicitly prohibits this. This nonconformance is rectified in Linux 2.6.9 and later.

The structure definition shown at the start of this page was taken from 4.3BSD Reno.

Ancient systems provided a **vtimes**() function with a similar purpose to **getrusage**(). For backward compatibility, glibc (up until Linux 2.32) also provides **vtimes**(). All new applications should be written using **getrusage**(). (Since Linux 2.33, glibc no longer provides an **vtimes**() implementation.)

NOTES

Resource usage metrics are preserved across an *execve*(2).

SEE ALSO

clock_gettime(2), *getrlimit*(2), *times*(2), *wait*(2), *wait4*(2), *clock*(3), *proc_pid_stat*(5), *proc_pid_io*(5)

NAME

getsid − get session ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

pid_t getsid(pid_t *pid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getsid():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION

getsid() returns the session ID of the process with process ID *pid*. If *pid* is 0, **getsid**() returns the session ID of the calling process.

RETURN VALUE

On success, a session ID is returned. On error, *(pid_t) −1* is returned, and *errno* is set to indicate the error.

ERRORS

EPERM
A process with process ID *pid* exists, but it is not in the same session as the calling process, and the implementation considers this an error.

ESRCH
No process with process ID *pid* was found.

VERSIONS

Linux does not return **EPERM**.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4. Linux 2.0.

NOTES

See *credentials*(7) for a description of sessions and session IDs.

SEE ALSO

getpgid(2), *setsid*(2), *credentials*(7)

NAME

getsockname − get socket name

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict addrlen);
```

DESCRIPTION

getsockname() returns the current address to which the socket *sockfd* is bound, in the buffer pointed to by *addr*. The *addrlen* argument should be initialized to indicate the amount of space (in bytes) pointed to by *addr*. On return it contains the actual size of the socket address.

The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
The argument *sockfd* is not a valid file descriptor.

EFAULT
The *addr* argument points to memory not in a valid part of the process address space.

EINVAL
addrlen is invalid (e.g., is negative).

ENOBUFS
Insufficient resources were available in the system to perform the operation.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD (first appeared in 4.2BSD).

SEE ALSO

bind(2), *socket*(2), *getifaddrs*(3), *ip*(7), *socket*(7), *unix*(7)

NAME

getsockopt, setsockopt − get and set options on sockets

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname,
               void optval[restrict *.optlen],
               socklen_t *restrict optlen);
int setsockopt(int sockfd, int level, int optname,
               const void optval[.optlen],
               socklen_t optlen);
```

DESCRIPTION

getsockopt() and **setsockopt**() manipulate options for the socket referred to by the file descriptor *sockfd*. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the sockets API level, *level* is specified as **SOL_SOCKET**. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the **TCP** protocol, *level* should be set to the protocol number of **TCP**; see *getprotoent*(3).

The arguments *optval* and *optlen* are used to access option values for **setsockopt**(). For **getsockopt**() they identify a buffer in which the value for the requested option(s) are to be returned. For **getsockopt**(), *optlen* is a value-result argument, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be NULL.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual.

Most socket-level options utilize an *int* argument for *optval*. For **setsockopt**(), the argument should be nonzero to enable a boolean option, or zero if the option is to be disabled.

For a description of the available socket options see *socket*(7) and the appropriate protocol man pages.

RETURN VALUE

On success, zero is returned for the standard options. On error, −1 is returned, and *errno* is set to indicate the error.

Netfilter allows the programmer to define custom socket options with associated handlers; for such options, the return value on success is the value returned by the handler.

ERRORS

EBADF
: The argument *sockfd* is not a valid file descriptor.

EFAULT
: The address pointed to by *optval* is not in a valid part of the process address space. For **getsockopt**(), this error may also be returned if *optlen* is not in a valid part of the process address space.

EINVAL
: *optlen* invalid in **setsockopt**(). In some cases this error can also occur for an invalid value in *optval* (e.g., for the **IP_ADD_MEMBERSHIP** option described in *ip*(7)).

ENOPROTOOPT
: The option is unknown at the level indicated.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

STANDARDS
POSIX.1-2008.

HISTORY
POSIX.1-2001, SVr4, 4.4BSD (first appeared in 4.2BSD).

BUGS
Several of the socket options should be handled at lower levels of the system.

SEE ALSO
ioctl(2), *socket*(2), *getprotoent*(3), *protocols*(5), *ip*(7), *packet*(7), *socket*(7), *tcp*(7), *udp*(7), *unix*(7)

NAME

gettid − get thread identification

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE
#include <unistd.h>

pid_t gettid(void);
```

DESCRIPTION

gettid() returns the caller's thread ID (TID). In a single-threaded process, the thread ID is equal to the process ID (PID, as returned by *getpid*(2)). In a multithreaded process, all threads have the same PID, but each one has a unique TID. For further details, see the discussion of **CLONE_THREAD** in *clone*(2).

RETURN VALUE

On success, returns the thread ID of the calling thread.

ERRORS

This call is always successful.

STANDARDS

Linux.

HISTORY

Linux 2.4.11, glibc 2.30.

NOTES

The thread ID returned by this call is not the same thing as a POSIX thread ID (i.e., the opaque value returned by *pthread_self*(3)).

In a new thread group created by a *clone*(2) call that does not specify the **CLONE_THREAD** flag (or, equivalently, a new process created by *fork*(2)), the new process is a thread group leader, and its thread group ID (the value returned by *getpid*(2)) is the same as its thread ID (the value returned by *gettid*())

SEE ALSO

capget(2), *clone*(2), *fcntl*(2), *fork*(2), *get_robust_list*(2), *getpid*(2), *ioprio_set*(2), *perf_event_open*(2), *sched_setaffinity*(2), *sched_setparam*(2), *sched_setscheduler*(2), *tgkill*(2), *timer_create*(2)

NAME

gettimeofday, settimeofday − get / set time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tv,
                 struct timezone *_Nullable restrict tz);
int settimeofday(const struct timeval *tv,
                 const struct timezone *_Nullable tz);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

settimeofday():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The functions **gettimeofday**() and **settimeofday**() can get and set the time as well as a timezone.

The *tv* argument is a *struct timeval* (as specified in <*sys/time.h*>):

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

and gives the number of seconds and microseconds since the Epoch (see *time*(2)).

The *tz* argument is a *struct timezone*:

```
struct timezone {
    int tz_minuteswest;     /* minutes west of Greenwich */
    int tz_dsttime;         /* type of DST correction */
};
```

If either *tv* or *tz* is NULL, the corresponding structure is not set or returned. (However, compilation warnings will result if *tv* is NULL.)

The use of the *timezone* structure is obsolete; the *tz* argument should normally be specified as NULL. (See NOTES below.)

Under Linux, there are some peculiar "warp clock" semantics associated with the **settimeofday**() system call if on the very first call (after booting) that has a non-NULL *tz* argument, the *tv* argument is NULL and the *tz_minuteswest* field is nonzero. (The *tz_dsttime* field should be zero for this case.) In such a case it is assumed that the CMOS clock is on local time, and that it has to be incremented by this amount to get UTC system time. No doubt it is a bad idea to use this feature.

RETURN VALUE

gettimeofday() and **settimeofday**() return 0 for success. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
One of *tv* or *tz* pointed outside the accessible address space.

EINVAL
(**settimeofday**()): *timezone* is invalid.

EINVAL
(**settimeofday**()): *tv.tv_sec* is negative or *tv.tv_usec* is outside the range [0, 999,999].

EINVAL (since Linux 4.3)
(**settimeofday**()): An attempt was made to set the time to a value less than the current value of the **CLOCK_MONOTONIC** clock (see *clock_gettime*(2)).

EPERM
The calling process has insufficient privilege to call **settimeofday**(); under Linux the **CAP_SYS_TIME** capability is required.

VERSIONS

C library/kernel differences

On some architectures, an implementation of **gettimeofday**() is provided in the *vdso*(7).

The kernel accepts NULL for both *tv* and *tz*. The timezone argument is ignored by glibc and musl, and not passed to/from the kernel. Android's bionic passes the timezone argument to/from the kernel, but Android does not update the kernel timezone based on the device timezone in Settings, so the kernel's timezone is typically UTC.

STANDARDS

gettimeofday()
POSIX.1-2008 (obsolete).

settimeofday()
None.

HISTORY

SVr4, 4.3BSD. POSIX.1-2001 describes **gettimeofday**() but not **settimeofday**(). POSIX.1-2008 marks **gettimeofday**() as obsolete, recommending the use of *clock_gettime*(2) instead.

Traditionally, the fields of *struct timeval* were of type *long*.

The tz_dsttime field

On a non-Linux kernel, with glibc, the *tz_dsttime* field of *struct timezone* will be set to a nonzero value by **gettimeofday**() if the current timezone has ever had or will have a daylight saving rule applied. In this sense it exactly mirrors the meaning of *daylight*(3) for the current zone. On Linux, with glibc, the setting of the *tz_dsttime* field of *struct timezone* has never been used by **settimeofday**() or **gettimeofday**(). Thus, the following is purely of historical interest.

On old systems, the field *tz_dsttime* contains a symbolic constant (values are given below) that indicates in which part of the year Daylight Saving Time is in force. (Note: this value is constant throughout the year: it does not indicate that DST is in force, it just selects an algorithm.) The daylight saving time algorithms defined are as follows:

```
DST_NONE     /* not on DST */
DST_USA      /* USA style DST */
DST_AUST     /* Australian style DST */
DST_WET      /* Western European DST */
DST_MET      /* Middle European DST */
DST_EET      /* Eastern European DST */
DST_CAN      /* Canada */
DST_GB       /* Great Britain and Eire */
DST_RUM      /* Romania */
DST_TUR      /* Turkey */
DST_AUSTALT  /* Australian style with shift in 1986 */
```

Of course it turned out that the period in which Daylight Saving Time is in force cannot be given by a simple algorithm, one per country; indeed, this period is determined by unpredictable political decisions. So this method of representing timezones has been abandoned.

NOTES

The time returned by **gettimeofday**() *is* affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the system time). If you need a monotonically increasing clock, see *clock_gettime*(2).

Macros for operating on *timeval* structures are described in *timeradd*(3).

SEE ALSO

date(1), *adjtimex*(2), *clock_gettime*(2), *time*(2), *ctime*(3), *ftime*(3), *timeradd*(3), *capabilities*(7), *time*(7), *vdso*(7), *hwclock*(8)

NAME

getuid, geteuid − get user identity

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

uid_t getuid(void);
uid_t geteuid(void);
```

DESCRIPTION

getuid() returns the real user ID of the calling process.

geteuid() returns the effective user ID of the calling process.

ERRORS

These functions are always successful and never modify *errno*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

In UNIX V6 the **getuid**() call returned *(euid << 8) + uid*. UNIX V7 introduced separate calls **getuid**() and **geteuid**().

The original Linux **getuid**() and **geteuid**() system calls supported only 16-bit user IDs. Subsequently, Linux 2.4 added **getuid32**() and **geteuid32**(), supporting 32-bit IDs. The glibc **getuid**() and **geteuid**() wrapper functions transparently deal with the variations across kernel versions.

On Alpha, instead of a pair of **getuid**() and **geteuid**() system calls, a single **getxuid**() system call is provided, which returns a pair of real and effective UIDs. The glibc **getuid**() and **geteuid**() wrapper functions transparently deal with this. See *syscall*(2) for details regarding register mapping.

SEE ALSO

getresuid(2), *setreuid*(2), *setuid*(2), *credentials*(7)

NAME

getunwind − copy the unwind data to caller's buffer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/unwind.h>
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

[[deprecated]] long syscall(SYS_getunwind, void *buf***[.***buf_size***],**
size_t *buf_size***);**

DESCRIPTION

Note: this system call is obsolete.

The IA-64-specific **getunwind**() system call copies the kernel's call frame unwind data into the buffer pointed to by *buf* and returns the size of the unwind data; this data describes the gate page (kernel code that is mapped into user space).

The size of the buffer *buf* is specified in *buf_size*. The data is copied only if *buf_size* is greater than or equal to the size of the unwind data and *buf* is not NULL; otherwise, no data is copied, and the call succeeds, returning the size that would be needed to store the unwind data.

The first part of the unwind data contains an unwind table. The rest contains the associated unwind information, in no particular order. The unwind table contains entries of the following form:

```
u64 start;      (64−bit address of start of function)
u64 end;        (64−bit address of end of function)
u64 info;       (BUF−relative offset to unwind info)
```

An entry whose *start* value is zero indicates the end of the table. For more information about the format, see the *IA-64 Software Conventions and Runtime Architecture* manual.

RETURN VALUE

On success, **getunwind**() returns the size of the unwind data. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

getunwind() fails with the error **EFAULT** if the unwind info can't be stored in the space specified by *buf*.

STANDARDS

Linux on IA-64.

HISTORY

Linux 2.4.

This system call has been deprecated. The modern way to obtain the kernel's unwind data is via the *vdso*(7).

SEE ALSO

getauxval(3)

NAME

getxattr, lgetxattr, fgetxattr − retrieve an extended attribute value

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/xattr.h>

ssize_t getxattr(const char *path, const char *name,
                 void value[.size], size_t size);
ssize_t lgetxattr(const char *path, const char *name,
                 void value[.size], size_t size);
ssize_t fgetxattr(int fd, const char *name,
                 void value[.size], size_t size);
```

DESCRIPTION

Extended attributes are *name*:*value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the *stat*(2) data). A complete overview of extended attributes concepts can be found in *xattr*(7).

getxattr() retrieves the value of the extended attribute identified by *name* and associated with the given *path* in the filesystem. The attribute value is placed in the buffer pointed to by *value*; *size* specifies the size of that buffer. The return value of the call is the number of bytes placed in *value*.

lgetxattr() is identical to **getxattr**(), except in the case of a symbolic link, where the link itself is interrogated, not the file that it refers to.

fgetxattr() is identical to **getxattr**(), only the open file referred to by *fd* (as returned by *open*(2)) is interrogated in place of *path*.

An extended attribute *name* is a null-terminated string. The name includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode. The value of an extended attribute is a chunk of arbitrary textual or binary data that was assigned using *setxattr*(2).

If *size* is specified as zero, these calls return the current size of the named extended attribute (and leave *value* unchanged). This can be used to determine the size of the buffer that should be supplied in a subsequent call. (But, bear in mind that there is a possibility that the attribute value may change between the two calls, so that it is still necessary to check the return status from the second call.)

RETURN VALUE

On success, these calls return a nonnegative value which is the size (in bytes) of the extended attribute value. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

E2BIG The size of the attribute value is larger than the maximum size allowed; the attribute cannot be retrieved. This can happen on filesystems that support very large attribute values such as NFSv4, for example.

ENODATA
The named attribute does not exist, or the process has no access to this attribute.

ENOTSUP
Extended attributes are not supported by the filesystem, or are disabled.

ERANGE
The *size* of the *value* buffer is too small to hold the result.

In addition, the errors documented in *stat*(2) can also occur.

STANDARDS

Linux.

HISTORY

Linux 2.4, glibc 2.3.

EXAMPLES

See *listxattr*(2).

SEE ALSO

getfattr(1), *setfattr*(1), *listxattr*(2), *open*(2), *removexattr*(2), *setxattr*(2), *stat*(2), *symlink*(7), *xattr*(7)

NAME

idle − make process 0 idle

SYNOPSIS

#include <unistd.h>

[[deprecated]] int idle(void);

DESCRIPTION

idle() is an internal system call used during bootstrap. It marks the process's pages as swappable, lowers its priority, and enters the main scheduling loop. **idle**() never returns.

Only process 0 may call **idle**(). Any user process, even a process with superuser permission, will receive **EPERM**.

RETURN VALUE

idle() never returns for process 0, and always returns −1 for a user process.

ERRORS

EPERM

Always, for a user process.

STANDARDS

Linux.

HISTORY

Removed in Linux 2.3.13.

NAME

init_module, finit_module − load a kernel module

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/module.h>    /* Definition of MODULE_* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_init_module, void module_image[.len], unsigned long len,
        const char *param_values);
int syscall(SYS_finit_module, int fd,
        const char *param_values, int flags);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

init_module() loads an ELF image into kernel space, performs any necessary symbol relocations, initializes module parameters to values provided by the caller, and then runs the module's *init* function. This system call requires privilege.

The *module_image* argument points to a buffer containing the binary image to be loaded; *len* specifies the size of that buffer. The module image should be a valid ELF image, built for the running kernel.

The *param_values* argument is a string containing space-delimited specifications of the values for module parameters (defined inside the module using **module_param**() and *module_param_array*())The kernel parses this string and initializes the specified parameters. Each of the parameter specifications has the form:

name[=*value*[,*value*...]]

The parameter *name* is one of those defined within the module using *module_param*() (see the Linux kernel source file *include/linux/moduleparam.h*). The parameter *value* is optional in the case of *bool* and *invbool* parameters. Values for array parameters are specified as a comma-separated list.

finit_module()

The **finit_module**() system call is like **init_module**(), but reads the module to be loaded from the file descriptor *fd*. It is useful when the authenticity of a kernel module can be determined from its location in the filesystem; in cases where that is possible, the overhead of using cryptographically signed modules to determine the authenticity of a module can be avoided. The *param_values* argument is as for **init_module**().

The *flags* argument modifies the operation of **finit_module**(). It is a bit mask value created by ORing together zero or more of the following flags:

MODULE_INIT_IGNORE_MODVERSIONS
: Ignore symbol version hashes.

MODULE_INIT_IGNORE_VERMAGIC
: Ignore kernel version magic.

There are some safety checks built into a module to ensure that it matches the kernel against which it is loaded. These checks are recorded when the module is built and verified when the module is loaded. First, the module records a "vermagic" string containing the kernel version number and prominent features (such as the CPU type). Second, if the module was built with the **CONFIG_MODVERSIONS** configuration option enabled, a version hash is recorded for each symbol the module uses. This hash is based on the types of the arguments and return value for the function named by the symbol. In this case, the kernel version number within the "vermagic" string is ignored, as the symbol version hashes are assumed to be sufficiently reliable.

Using the **MODULE_INIT_IGNORE_VERMAGIC** flag indicates that the "vermagic" string is to be ignored, and the **MODULE_INIT_IGNORE_MODVERSIONS** flag indicates that the symbol version hashes are to be ignored. If the kernel is built to permit forced loading (i.e., configured with **CONFIG_MODULE_FORCE_LOAD**), then loading continues, otherwise it fails with the error **ENOEXEC** as expected for malformed modules.

RETURN VALUE

On success, these system calls return 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADMSG (since Linux 3.7)
Module signature is misformatted.

EBUSY
Timeout while trying to resolve a symbol reference by this module.

EFAULT
An address argument referred to a location that is outside the process's accessible address space.

ENOKEY (since Linux 3.7)
Module signature is invalid or the kernel does not have a key for this module. This error is returned only if the kernel was configured with **CONFIG_MODULE_SIG_FORCE**; if the kernel was not configured with this option, then an invalid or unsigned module simply taints the kernel.

ENOMEM
Out of memory.

EPERM
The caller was not privileged (did not have the **CAP_SYS_MODULE** capability), or module loading is disabled (see */proc/sys/kernel/modules_disabled* in *proc*(5)).

The following errors may additionally occur for **init_module**():

EEXIST
A module with this name is already loaded.

EINVAL
param_values is invalid, or some part of the ELF image in *module_image* contains inconsistencies.

ENOEXEC
The binary image supplied in *module_image* is not an ELF image, or is an ELF image that is invalid or for a different architecture.

The following errors may additionally occur for **finit_module**():

EBADF
The file referred to by *fd* is not opened for reading.

EFBIG
The file referred to by *fd* is too large.

EINVAL
flags is invalid.

ENOEXEC
fd does not refer to an open file.

ETXTBSY (since Linux 4.7)
The file referred to by *fd* is opened for read-write.

In addition to the above errors, if the module's *init* function is executed and returns an error, then **init_module**() or **finit_module**() fails and *errno* is set to the value returned by the *init* function.

STANDARDS

Linux.

HISTORY

finit_module()
Linux 3.8.

The **init_module**() system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc versions before glibc 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it is (before glibc 2.23) sufficient to manually declare the interface in your code; alternatively, you can invoke the system call using *syscall*(2).

Linux 2.4 and earlier

In Linux 2.4 and earlier, the **init_module**() system call was rather different:

#include <linux/module.h>

int init_module(const char ****name*, struct module ****image***);**

(User-space applications can detect which version of **init_module**() is available by calling **query_module**(); the latter call fails with the error **ENOSYS** on Linux 2.6 and later.)

The older version of the system call loads the relocated module image pointed to by *image* into kernel space and runs the module's *init* function. The caller is responsible for providing the relocated image (since Linux 2.6, the **init_module**() system call does the relocation).

The module image begins with a module structure and is followed by code and data as appropriate. Since Linux 2.2, the module structure is defined as follows:

```
struct module {
    unsigned long         size_of_struct;
    struct module        *next;
    const char           *name;
    unsigned long         size;
    long                  usecount;
    unsigned long         flags;
    unsigned int          nsyms;
    unsigned int          ndeps;
    struct module_symbol *syms;
    struct module_ref    *deps;
    struct module_ref    *refs;
    int                 (*init)(void);
    void                (*cleanup)(void);
    const struct exception_table_entry *ex_table_start;
    const struct exception_table_entry *ex_table_end;
#ifdef __alpha__
    unsigned long gp;
#endif
};
```

All of the pointer fields, with the exception of *next* and *refs*, are expected to point within the module body and be initialized as appropriate for kernel space, that is, relocated with the rest of the module.

NOTES

Information about currently loaded modules can be found in */proc/modules* and in the file trees under the per-module subdirectories under */sys/module*.

See the Linux kernel source file *include/linux/module.h* for some useful background information.

SEE ALSO

create_module(2), *delete_module*(2), *query_module*(2), *lsmod*(8), *modprobe*(8)

NAME

inotify_add_watch − add a watch to an initialized inotify instance

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/inotify.h>

int inotify_add_watch(int *fd***, const char *** *pathname***, uint32_t** *mask***);**

DESCRIPTION

inotify_add_watch() adds a new watch, or modifies an existing watch, for the file whose location is specified in *pathname*; the caller must have read permission for this file. The *fd* argument is a file descriptor referring to the inotify instance whose watch list is to be modified. The events to be monitored for *pathname* are specified in the *mask* bit-mask argument. See *inotify*(7) for a description of the bits that can be set in *mask*.

A successful call to **inotify_add_watch**() returns a unique watch descriptor for this inotify instance, for the filesystem object (inode) that corresponds to *pathname*. If the filesystem object was not previously being watched by this inotify instance, then the watch descriptor is newly allocated. If the filesystem object was already being watched (perhaps via a different link to the same object), then the descriptor for the existing watch is returned.

The watch descriptor is returned by later *read*(2)s from the inotify file descriptor. These reads fetch *inotify_event* structures (see *inotify*(7)) indicating filesystem events; the watch descriptor inside this structure identifies the object for which the event occurred.

RETURN VALUE

On success, **inotify_add_watch**() returns a watch descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
Read access to the given file is not permitted.

EBADF
The given file descriptor is not valid.

EEXIST
mask contains **IN_MASK_CREATE** and *pathname* refers to a file already being watched by the same *fd*.

EFAULT
pathname points outside of the process's accessible address space.

EINVAL
The given event mask contains no valid events; or *mask* contains both **IN_MASK_ADD** and **IN_MASK_CREATE**; or *fd* is not an inotify file descriptor.

ENAMETOOLONG
pathname is too long.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The user limit on the total number of inotify watches was reached or the kernel failed to allocate a needed resource.

ENOTDIR
mask contains **IN_ONLYDIR** and *pathname* is not a directory.

STANDARDS

Linux.

HISTORY

Linux 2.6.13.

EXAMPLES

See *inotify*(7).

SEE ALSO

inotify_init(2), *inotify_rm_watch*(2), *inotify*(7)

NAME

inotify_init, inotify_init1 − initialize an inotify instance

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/inotify.h>

int inotify_init(void);
int inotify_init1(int *flags***);**

DESCRIPTION

For an overview of the inotify API, see *inotify*(7).

inotify_init() initializes a new inotify instance and returns a file descriptor associated with a new inotify event queue.

If *flags* is 0, then **inotify_init1**() is the same as **inotify_init**(). The following values can be bitwise ORed in *flags* to obtain different behavior:

IN_NONBLOCK
: Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

IN_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

RETURN VALUE

On success, these system calls return a new file descriptor. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: (**inotify_init1**()) An invalid value was specified in *flags*.

EMFILE
: The user limit on the total number of inotify instances has been reached.

EMFILE
: The per-process limit on the number of open file descriptors has been reached.

ENFILE
: The system-wide limit on the total number of open files has been reached.

ENOMEM
: Insufficient kernel memory is available.

STANDARDS

Linux.

HISTORY

inotify_init()
: Linux 2.6.13, glibc 2.4.

inotify_init1()
: Linux 2.6.27, glibc 2.9.

SEE ALSO

inotify_add_watch(2), *inotify_rm_watch*(2), *inotify*(7)

NAME

inotify_rm_watch − remove an existing watch from an inotify instance

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/inotify.h>

int inotify_rm_watch(int *fd***, int** *wd***);**

DESCRIPTION

inotify_rm_watch() removes the watch associated with the watch descriptor *wd* from the inotify instance associated with the file descriptor *fd*.

Removing a watch causes an **IN_IGNORED** event to be generated for this watch descriptor. (See *inotify*(7).)

RETURN VALUE

On success, **inotify_rm_watch**() returns zero. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not a valid file descriptor.

EINVAL
The watch descriptor *wd* is not valid; or *fd* is not an inotify file descriptor.

STANDARDS

Linux.

HISTORY

Linux 2.6.13.

SEE ALSO

inotify_add_watch(2), *inotify_init*(2), *inotify*(7)

NAME

io_cancel − cancel an outstanding asynchronous I/O operation

LIBRARY

Standard C library (*libc*, *−lc*)

Alternatively, Asynchronous I/O library (*libaio*, *−laio*); see VERSIONS.

SYNOPSIS

```
#include <linux/aio_abi.h>    /* Definition of needed types */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_io_cancel, aio_context_t ctx_id, struct iocb *iocb,
            struct io_event *result);
```

DESCRIPTION

Note: this page describes the raw Linux system call interface. The wrapper function provided by *libaio* uses a different type for the *ctx_id* argument. See VERSIONS.

The **io_cancel**() system call attempts to cancel an asynchronous I/O operation previously submitted with *io_submit*(2). The *iocb* argument describes the operation to be canceled and the *ctx_id* argument is the AIO context to which the operation was submitted. If the operation is successfully canceled, the event will be copied into the memory pointed to by *result* without being placed into the completion queue.

RETURN VALUE

On success, **io_cancel**() returns 0. For the failure return, see VERSIONS.

ERRORS

EAGAIN
: The *iocb* specified was not canceled.

EFAULT
: One of the data structures points to invalid data.

EINVAL
: The AIO context specified by *ctx_id* is invalid.

ENOSYS
: **io_cancel**() is not implemented on this architecture.

VERSIONS

You probably want to use the **io_cancel**() wrapper function provided by *libaio*.

Note that the *libaio* wrapper function uses a different type (*io_context_t*) for the *ctx_id* argument. Note also that the *libaio* wrapper does not follow the usual C library conventions for indicating errors: on error it returns a negated error number (the negative of one of the values listed in ERRORS). If the system call is invoked via *syscall*(2), then the return value follows the usual conventions for indicating an error: −1, with *errno* set to a (positive) value that indicates the error.

STANDARDS

Linux.

HISTORY

Linux 2.5.

SEE ALSO

io_destroy(2), *io_getevents*(2), *io_setup*(2), *io_submit*(2), *aio*(7)

NAME

io_destroy − destroy an asynchronous I/O context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/aio_abi.h>    /* Definition of aio_context_t */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_io_destroy, aio_context_t ctx_id);
```

Note: glibc provides no wrapper for **io_destroy**(), necessitating the use of *syscall*(2).

DESCRIPTION

Note: this page describes the raw Linux system call interface. The wrapper function provided by *libaio* uses a different type for the *ctx_id* argument. See VERSIONS.

The **io_destroy**() system call will attempt to cancel all outstanding asynchronous I/O operations against *ctx_id*, will block on the completion of all operations that could not be canceled, and will destroy the *ctx_id*.

RETURN VALUE

On success, **io_destroy**() returns 0. For the failure return, see VERSIONS.

ERRORS

EFAULT
: The context pointed to is invalid.

EINVAL
: The AIO context specified by *ctx_id* is invalid.

ENOSYS
: **io_destroy**() is not implemented on this architecture.

VERSIONS

You probably want to use the **io_destroy**() wrapper function provided by *libaio*.

Note that the *libaio* wrapper function uses a different type (*io_context_t*) for the *ctx_id* argument. Note also that the *libaio* wrapper does not follow the usual C library conventions for indicating errors: on error it returns a negated error number (the negative of one of the values listed in ERRORS). If the system call is invoked via *syscall*(2), then the return value follows the usual conventions for indicating an error: −1, with *errno* set to a (positive) value that indicates the error.

STANDARDS

Linux.

HISTORY

Linux 2.5.

SEE ALSO

io_cancel(2), *io_getevents*(2), *io_setup*(2), *io_submit*(2), *aio*(7)

NAME

io_getevents − read asynchronous I/O events from the completion queue

LIBRARY

Standard C library (*libc*, *−lc*)

Alternatively, Asynchronous I/O library (*libaio*, *−laio*); see VERSIONS.

SYNOPSIS

```
#include <linux/aio_abi.h>    /* Definition of *io_* types */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_io_getevents, aio_context_t ctx_id,
            long min_nr, long nr, struct io_event *events,
            struct timespec *timeout);
```

Note: glibc provides no wrapper for **io_getevents**(), necessitating the use of *syscall*(2).

DESCRIPTION

Note: this page describes the raw Linux system call interface. The wrapper function provided by *libaio* uses a different type for the *ctx_id* argument. See VERSIONS.

The **io_getevents**() system call attempts to read at least *min_nr* events and up to *nr* events from the completion queue of the AIO context specified by *ctx_id*.

The *timeout* argument specifies the amount of time to wait for events, and is specified as a relative timeout in a *timespec*(3) structure.

The specified time will be rounded up to the system clock granularity and is guaranteed not to expire early.

Specifying *timeout* as NULL means block indefinitely until at least *min_nr* events have been obtained.

RETURN VALUE

On success, **io_getevents**() returns the number of events read. This may be 0, or a value less than *min_nr*, if the *timeout* expired. It may also be a nonzero value less than *min_nr*, if the call was interrupted by a signal handler.

For the failure return, see VERSIONS.

ERRORS

EFAULT
: Either *events* or *timeout* is an invalid pointer.

EINTR
: Interrupted by a signal handler; see *signal*(7).

EINVAL
: *ctx_id* is invalid. *min_nr* is out of range or *nr* is out of range.

ENOSYS
: **io_getevents**() is not implemented on this architecture.

VERSIONS

You probably want to use the **io_getevents**() wrapper function provided by *libaio*.

Note that the *libaio* wrapper function uses a different type (*io_context_t*) for the *ctx_id* argument. Note also that the *libaio* wrapper does not follow the usual C library conventions for indicating errors: on error it returns a negated error number (the negative of one of the values listed in ERRORS). If the system call is invoked via *syscall*(2), then the return value follows the usual conventions for indicating an error: −1, with *errno* set to a (positive) value that indicates the error.

STANDARDS

Linux.

HISTORY

Linux 2.5.

BUGS

An invalid *ctx_id* may cause a segmentation fault instead of generating the error **EINVAL**.

SEE ALSO

io_cancel(2), *io_destroy*(2), *io_setup*(2), *io_submit*(2), *timespec*(3), *aio*(7), *time*(7)

NAME

io_setup − create an asynchronous I/O context

LIBRARY

Standard C library (*libc*, *−lc*)

Alternatively, Asynchronous I/O library (*libaio*, *−laio*); see VERSIONS.

SYNOPSIS

#include <linux/aio_abi.h> /* Defines needed types */

long io_setup(unsigned int *nr_events***, aio_context_t ***_ctx_idp_**);**

Note: There is no glibc wrapper for this system call; see VERSIONS.

DESCRIPTION

Note: this page describes the raw Linux system call interface. The wrapper function provided by *libaio* uses a different type for the *ctx_idp* argument. See VERSIONS.

The **io_setup**() system call creates an asynchronous I/O context suitable for concurrently processing *nr_events* operations. The *ctx_idp* argument must not point to an AIO context that already exists, and must be initialized to 0 prior to the call. On successful creation of the AIO context, **ctx_idp* is filled in with the resulting handle.

RETURN VALUE

On success, **io_setup**() returns 0. For the failure return, see VERSIONS.

ERRORS

EAGAIN
: The specified *nr_events* exceeds the limit of available events, as defined in */proc/sys/fs/aio−max−nr* (see *proc*(5)).

EFAULT
: An invalid pointer is passed for *ctx_idp*.

EINVAL
: *ctx_idp* is not initialized, or the specified *nr_events* exceeds internal limits. *nr_events* should be greater than 0.

ENOMEM
: Insufficient kernel resources are available.

ENOSYS
: **io_setup**() is not implemented on this architecture.

VERSIONS

glibc does not provide a wrapper for this system call. You could invoke it using *syscall*(2). But instead, you probably want to use the **io_setup**() wrapper function provided by *libaio*.

Note that the *libaio* wrapper function uses a different type (*io_context_t* *) for the *ctx_idp* argument. Note also that the *libaio* wrapper does not follow the usual C library conventions for indicating errors: on error it returns a negated error number (the negative of one of the values listed in ERRORS). If the system call is invoked via *syscall*(2), then the return value follows the usual conventions for indicating an error: −1, with *errno* set to a (positive) value that indicates the error.

STANDARDS

Linux.

HISTORY

Linux 2.5.

SEE ALSO

io_cancel(2), *io_destroy*(2), *io_getevents*(2), *io_submit*(2), *aio*(7)

NAME

io_submit − submit asynchronous I/O blocks for processing

LIBRARY

Standard C library (*libc*, *−lc*)

Alternatively, Asynchronous I/O library (*libaio*, *−laio*); see VERSIONS.

SYNOPSIS

#include <linux/aio_abi.h> /* Defines needed types */

int io_submit(aio_context_t *ctx_id***, long** *nr***, struct iocb *****iocbpp***);**

Note: There is no glibc wrapper for this system call; see VERSIONS.

DESCRIPTION

Note: this page describes the raw Linux system call interface. The wrapper function provided by *libaio* uses a different type for the *ctx_id* argument. See VERSIONS.

The **io_submit**() system call queues *nr* I/O request blocks for processing in the AIO context *ctx_id*. The *iocbpp* argument should be an array of *nr* AIO control blocks, which will be submitted to context *ctx_id*.

The *iocb* (I/O control block) structure defined in *linux/aio_abi.h* defines the parameters that control the I/O operation.

```
#include <linux/aio_abi.h>

struct iocb {
    __u64   aio_data;
    __u32   PADDED(aio_key, aio_rw_flags);
    __u16   aio_lio_opcode;
    __s16   aio_reqprio;
    __u32   aio_fildes;
    __u64   aio_buf;
    __u64   aio_nbytes;
    __s64   aio_offset;
    __u64   aio_reserved2;
    __u32   aio_flags;
    __u32   aio_resfd;
};
```

The fields of this structure are as follows:

aio_data
: This data is copied into the *data* field of the *io_event* structure upon I/O completion (see *io_getevents*(2)).

aio_key
: This is an internal field used by the kernel. Do not modify this field after an **io_submit**() call.

aio_rw_flags
: This defines the R/W flags passed with structure. The valid values are:

  RWF_APPEND (since Linux 4.16)
  : Append data to the end of the file. See the description of the flag of the same name in *pwritev2*(2) as well as the description of **O_APPEND** in *open*(2). The *aio_offset* field is ignored. The file offset is not changed.

  RWF_DSYNC (since Linux 4.13)
  : Write operation complete according to requirement of synchronized I/O data integrity. See the description of the flag of the same name in *pwritev2*(2) as well the description of **O_DSYNC** in *open*(2).

  RWF_HIPRI (since Linux 4.13)
  : High priority request, poll if possible

RWF_NOWAIT (since Linux 4.14)
Don't wait if the I/O will block for operations such as file block allocations, dirty page flush, mutex locks, or a congested block device inside the kernel. If any of these conditions are met, the control block is returned immediately with a return value of **−EAGAIN** in the *res* field of the *io_event* structure (see *io_getevents*(2)).

RWF_SYNC (since Linux 4.13)
Write operation complete according to requirement of synchronized I/O file integrity. See the description of the flag of the same name in *pwritev2*(2) as well the description of **O_SYNC** in *open*(2).

aio_lio_opcode
This defines the type of I/O to be performed by the *iocb* structure. The valid values are defined by the enum defined in *linux/aio_abi.h*:

```
enum {
    IOCB_CMD_PREAD = 0,
    IOCB_CMD_PWRITE = 1,
    IOCB_CMD_FSYNC = 2,
    IOCB_CMD_FDSYNC = 3,
    IOCB_CMD_POLL = 5,
    IOCB_CMD_NOOP = 6,
    IOCB_CMD_PREADV = 7,
    IOCB_CMD_PWRITEV = 8,
};
```

aio_reqprio
This defines the requests priority.

aio_fildes
The file descriptor on which the I/O operation is to be performed.

aio_buf
This is the buffer used to transfer data for a read or write operation.

aio_nbytes
This is the size of the buffer pointed to by *aio_buf*.

aio_offset
This is the file offset at which the I/O operation is to be performed.

aio_flags
This is the set of flags associated with the *iocb* structure. The valid values are:

IOCB_FLAG_RESFD
Asynchronous I/O control must signal the file descriptor mentioned in *aio_resfd* upon completion.

IOCB_FLAG_IOPRIO (since Linux 4.18)
Interpret the *aio_reqprio* field as an **IOPRIO_VALUE** as defined by *linux/ioprio.h*.

aio_resfd
The file descriptor to signal in the event of asynchronous I/O completion.

RETURN VALUE

On success, **io_submit**() returns the number of *iocb*s submitted (which may be less than *nr*, or 0 if *nr* is zero). For the failure return, see VERSIONS.

ERRORS

EAGAIN
Insufficient resources are available to queue any *iocb*s.

EBADF
The file descriptor specified in the first *iocb* is invalid.

EFAULT
One of the data structures points to invalid data.

EINVAL
The AIO context specified by *ctx_id* is invalid. *nr* is less than 0. The *iocb* at **iocbpp[0]* is not properly initialized, the operation specified is invalid for the file descriptor in the *iocb*, or the value in the *aio_reqprio* field is invalid.

ENOSYS
io_submit() is not implemented on this architecture.

EPERM
The *aio_reqprio* field is set with the class **IOPRIO_CLASS_RT**, but the submitting context does not have the **CAP_SYS_ADMIN** capability.

VERSIONS

glibc does not provide a wrapper for this system call. You could invoke it using *syscall*(2). But instead, you probably want to use the **io_submit**() wrapper function provided by *libaio*.

Note that the *libaio* wrapper function uses a different type (*io_context_t*) for the *ctx_id* argument. Note also that the *libaio* wrapper does not follow the usual C library conventions for indicating errors: on error it returns a negated error number (the negative of one of the values listed in ERRORS). If the system call is invoked via *syscall*(2), then the return value follows the usual conventions for indicating an error: −1, with *errno* set to a (positive) value that indicates the error.

STANDARDS

Linux.

HISTORY

Linux 2.5.

SEE ALSO

io_cancel(2), *io_destroy*(2), *io_getevents*(2), *io_setup*(2), *aio*(7)

NAME

ioctl − control device

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long op, ...);  /* glibc, BSD */
int ioctl(int fd, int op, ...);            /* musl, other UNIX */
```

DESCRIPTION

The **ioctl**() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() operations. The argument *fd* must be an open file descriptor.

The second argument is a device-dependent operation code. The third argument is an untyped pointer to memory. It's traditionally **char ***argp* (from the days before **void *** was valid C), and will be so named for this discussion.

An **ioctl**() *op* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl**() *op* are located in the file *<sys/ioctl.h>*. See NOTES.

RETURN VALUE

Usually, on success zero is returned. A few **ioctl**() operations use the return value as an output parameter and return a nonnegative value on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
: *fd* is not a valid file descriptor.

EFAULT
: *argp* references an inaccessible memory area.

EINVAL
: *op* or *argp* is not valid.

ENOTTY
: *fd* is not associated with a character special device.

ENOTTY
: The specified operation does not apply to the kind of object that the file descriptor *fd* references.

VERSIONS

Arguments, returns, and semantics of **ioctl**() vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model).

STANDARDS

None.

HISTORY

Version 7 AT&T UNIX has

```
ioctl(int fildes, int op, struct sgttyb *argp);
```

(where **struct sgttyb** has historically been used by *stty*(2) and *gtty*(2), and is polymorphic by operation type (like a **void *** would be, if it had been available)).

SysIII documents *arg* without a type at all.

4.3BSD has

```
ioctl(int d, unsigned long op, char *argp);
```

(with **char *** similarly in for **void ***).

SysVr4 has

```
int ioctl(int fildes, int op, ... /* arg */);
```

NOTES

In order to use this call, one needs an open file descriptor. Often the *open*(2) call has unwanted side effects, that can be avoided under Linux by giving it the **O_NONBLOCK** flag.

ioctl structure

Ioctl *op* values are 32-bit constants. In principle these constants are completely arbitrary, but people have tried to build some structure into them.

The old Linux situation was that of mostly 16-bit constants, where the last byte is a serial number, and the preceding byte(s) give a type indicating the driver. Sometimes the major number was used: 0x03 for the **HDIO_*** ioctls, 0x06 for the **LP*** ioctls. And sometimes one or more ASCII letters were used. For example, **TCGETS** has value 0x00005401, with 0x54 = 'T' indicating the terminal driver, and **CYGETTIMEOUT** has value 0x00435906, with 0x43 0x59 = 'C' 'Y' indicating the cyclades driver.

Later (0.98p5) some more information was built into the number. One has 2 direction bits (00: none, 01: write, 10: read, 11: read/write) followed by 14 size bits (giving the size of the argument), followed by an 8-bit type (collecting the ioctls in groups for a common purpose or a common driver), and an 8-bit serial number.

The macros describing this structure live in *<asm/ioctl.h>* and are **_IO(type,nr)** and **{_IOR,_IOW,_IOWR}(type,nr,size)**. They use *sizeof(size)* so that size is a misnomer here: this third argument is a data type.

Note that the size bits are very unreliable: in lots of cases they are wrong, either because of buggy macros using *sizeof(sizeof(struct))*, or because of legacy values.

Thus, it seems that the new structure only gave disadvantages: it does not help in checking, but it causes varying values for the various architectures.

SEE ALSO

execve(2), *fcntl*(2), *ioctl_console*(2), *ioctl_fat*(2), *ioctl_ficlone*(2), *ioctl_ficlonerange*(2), *ioctl_fideduperange*(2), *ioctl_fslabel*(2), *ioctl_getfsmap*(2), *ioctl_iflags*(2), *ioctl_ns*(2), *ioctl_tty*(2), *ioctl_userfaultfd*(2), *open*(2), *sd*(4), *tty*(4)

NAME

ioctl_console − ioctls for console terminal and virtual consoles

DESCRIPTION

The following Linux-specific *ioctl*(2) operations are supported for console terminals and virtual consoles. Each operation requires a third argument, assumed here to be *argp*.

KDGETLED

Get state of LEDs. *argp* points to a *char*. The lower three bits of **argp* are set to the state of the LEDs, as follows:

LED_CAP	0x04	caps lock led
LED_NUM	0x02	num lock led
LED_SCR	0x01	scroll lock led

KDSETLED

Set the LEDs. The LEDs are set to correspond to the lower three bits of the unsigned long integer in *argp*. However, if a higher order bit is set, the LEDs revert to normal: displaying the state of the keyboard functions of caps lock, num lock, and scroll lock.

Before Linux 1.1.54, the LEDs just reflected the state of the corresponding keyboard flags, and KDGETLED/KDSETLED would also change the keyboard flags. Since Linux 1.1.54 the LEDs can be made to display arbitrary information, but by default they display the keyboard flags. The following two ioctls are used to access the keyboard flags.

KDGKBLED

Get keyboard flags CapsLock, NumLock, ScrollLock (not lights). *argp* points to a char which is set to the flag state. The low order three bits (mask 0x7) get the current flag state, and the low order bits of the next nibble (mask 0x70) get the default flag state. (Since Linux 1.1.54.)

KDSKBLED

Set keyboard flags CapsLock, NumLock, ScrollLock (not lights). *argp* is an unsigned long integer that has the desired flag state. The low order three bits (mask 0x7) have the flag state, and the low order bits of the next nibble (mask 0x70) have the default flag state. (Since Linux 1.1.54.)

KDGKBTYPE

Get keyboard type. This returns the value KB_101, defined as 0x02.

KDADDIO

Add I/O port as valid. Equivalent to *ioperm(arg,1,1)*.

KDDELIO

Delete I/O port as valid. Equivalent to *ioperm(arg,1,0)*.

KDENABIO

Enable I/O to video board. Equivalent to *ioperm(0x3b4, 0x3df−0x3b4+1, 1)*.

KDDISABIO

Disable I/O to video board. Equivalent to *ioperm(0x3b4, 0x3df−0x3b4+1, 0)*.

KDSETMODE

Set text/graphics mode. *argp* is an unsigned integer containing one of:

KD_TEXT	0x00
KD_GRAPHICS	0x01

KDGETMODE

Get text/graphics mode. *argp* points to an *int* which is set to one of the values shown above for **KDSETMODE**.

KDMKTONE

Generate tone of specified length. The lower 16 bits of the unsigned long integer in *argp* specify the period in clock cycles, and the upper 16 bits give the duration in msec. If the duration is zero, the sound is turned off. Control returns immediately. For example, *argp* = (125<<16) + 0x637 would specify the beep normally associated with a ctrl-G. (Thus since Linux 0.99pl1; broken in Linux 2.1.49-50.)

KIOCSOUND
Start or stop sound generation. The lower 16 bits of *argp* specify the period in clock cycles (that is, *argp* = 1193180/frequency). *argp* = 0 turns sound off. In either case, control returns immediately.

GIO_CMAP
Get the current default color map from kernel. *argp* points to a 48-byte array. (Since Linux 1.3.3.)

PIO_CMAP
Change the default text-mode color map. *argp* points to a 48-byte array which contains, in order, the Red, Green, and Blue values for the 16 available screen colors: 0 is off, and 255 is full intensity. The default colors are, in order: black, dark red, dark green, brown, dark blue, dark purple, dark cyan, light grey, dark grey, bright red, bright green, yellow, bright blue, bright purple, bright cyan, and white. (Since Linux 1.3.3.)

GIO_FONT
Gets 256-character screen font in expanded form. *argp* points to an 8192-byte array. Fails with error code **EINVAL** if the currently loaded font is a 512-character font, or if the console is not in text mode.

GIO_FONTX
Gets screen font and associated information. *argp* points to a *struct consolefontdesc* (see **PIO_FONTX**). On call, the *charcount* field should be set to the maximum number of characters that would fit in the buffer pointed to by *chardata*. On return, the *charcount* and *charheight* are filled with the respective data for the currently loaded font, and the *chardata* array contains the font data if the initial value of *charcount* indicated enough space was available; otherwise the buffer is untouched and *errno* is set to **ENOMEM**. (Since Linux 1.3.1.)

PIO_FONT
Sets 256-character screen font. Load font into the EGA/VGA character generator. *argp* points to an 8192-byte map, with 32 bytes per character. Only the first *N* of them are used for an 8x*N* font (0 < *N* <= 32). This call also invalidates the Unicode mapping.

PIO_FONTX
Sets screen font and associated rendering information. *argp* points to a

```
struct consolefontdesc {
    unsigned short charcount;  /* characters in font
                                  (256 or 512) */
    unsigned short charheight; /* scan lines per
                                  character (1−32) */
    char          *chardata;   /* font data in
                                  expanded form */
};
```

If necessary, the screen will be appropriately resized, and **SIGWINCH** sent to the appropriate processes. This call also invalidates the Unicode mapping. (Since Linux 1.3.1.)

PIO_FONTRESET
Resets the screen font, size, and Unicode mapping to the bootup defaults. *argp* is unused, but should be set to NULL to ensure compatibility with future versions of Linux. (Since Linux 1.3.28.)

GIO_SCRNMAP
Get screen mapping from kernel. *argp* points to an area of size E_TABSZ, which is loaded with the font positions used to display each character. This call is likely to return useless information if the currently loaded font is more than 256 characters.

GIO_UNISCRNMAP
Get full Unicode screen mapping from kernel. *argp* points to an area of size *E_TABSZ*sizeof(unsigned short)*, which is loaded with the Unicodes each character represent. A special set of Unicodes, starting at U+F000, are used to represent "direct to font" mappings. (Since Linux 1.3.1.)

PIO_SCRNMAP
Loads the "user definable" (fourth) table in the kernel which maps bytes into console screen symbols. *argp* points to an area of size E_TABSZ.

PIO_UNISCRNMAP
Loads the "user definable" (fourth) table in the kernel which maps bytes into Unicodes, which are then translated into screen symbols according to the currently loaded Unicode-to-font map. Special Unicodes starting at U+F000 can be used to map directly to the font symbols. (Since Linux 1.3.1.)

GIO_UNIMAP
Get Unicode-to-font mapping from kernel. *argp* points to a

```
struct unimapdesc {
    unsigned short  entry_ct;
    struct unipair *entries;
};
```

where *entries* points to an array of

```
struct unipair {
    unsigned short unicode;
    unsigned short fontpos;
};
```

(Since Linux 1.1.92.)

PIO_UNIMAP
Put unicode-to-font mapping in kernel. *argp* points to a *struct unimapdesc*. (Since Linux 1.1.92)

PIO_UNIMAPCLR
Clear table, possibly advise hash algorithm. *argp* points to a

```
struct unimapinit {
    unsigned short advised_hashsize;  /* 0 if no opinion */
    unsigned short advised_hashstep;  /* 0 if no opinion */
    unsigned short advised_hashlevel; /* 0 if no opinion */
};
```

(Since Linux 1.1.92.)

KDGKBMODE
Gets current keyboard mode. *argp* points to a *long* which is set to one of these:

```
K_RAW        0x00  /* Raw (scancode) mode */
K_XLATE      0x01  /* Translate keycodes using keymap */
K_MEDIUMRAW  0x02  /* Medium raw (scancode) mode */
K_UNICODE    0x03  /* Unicode mode */
K_OFF        0x04  /* Disabled mode; since Linux 2.6.39 */
```

KDSKBMODE
Sets current keyboard mode. *argp* is a *long* equal to one of the values shown for **KDGKBMODE**.

KDGKBMETA
Gets meta key handling mode. *argp* points to a *long* which is set to one of these:

```
K_METABIT    0x03    set high order bit
K_ESCPREFIX  0x04    escape prefix
```

KDSKBMETA
Sets meta key handling mode. *argp* is a *long* equal to one of the values shown above for **KDGKBMETA**.

KDGKBENT
Gets one entry in key translation table (keycode to action code). *argp* points to a

```
struct kbentry {
    unsigned char  kb_table;
```

```
    unsigned char  kb_index;
    unsigned short kb_value;
};
```

with the first two members filled in: *kb_table* selects the key table (0 <= *kb_table* < MAX_NR_KEYMAPS), and *kb_index* is the keycode (0 <= *kb_index* < NR_KEYS). *kb_value* is set to the corresponding action code, or K_HOLE if there is no such key, or K_NOSUCHMAP if *kb_table* is invalid.

KDSKBENT
Sets one entry in translation table. *argp* points to a *struct kbentry*.

KDGKBSENT
Gets one function key string. *argp* points to a

```
struct kbsentry {
    unsigned char kb_func;
    unsigned char kb_string[512];
};
```

kb_string is set to the (null-terminated) string corresponding to the *kb_func*th function key action code.

KDSKBSENT
Sets one function key string entry. *argp* points to a *struct kbsentry*.

KDGKBDIACR
Read kernel accent table. *argp* points to a

```
struct kbdiacrs {
    unsigned int   kb_cnt;
    struct kbdiacr kbdiacr[256];
};
```

where *kb_cnt* is the number of entries in the array, each of which is a

```
struct kbdiacr {
    unsigned char diacr;
    unsigned char base;
    unsigned char result;
};
```

KDGETKEYCODE
Read kernel keycode table entry (scan code to keycode). *argp* points to a

```
struct kbkeycode {
    unsigned int scancode;
    unsigned int keycode;
};
```

keycode is set to correspond to the given *scancode*. (89 <= *scancode* <= 255 only. For 1 <= *scancode* <= 88, *keycode*==*scancode*.) (Since Linux 1.1.63.)

KDSETKEYCODE
Write kernel keycode table entry. *argp* points to a *struct kbkeycode*. (Since Linux 1.1.63.)

KDSIGACCEPT
The calling process indicates its willingness to accept the signal *argp* when it is generated by pressing an appropriate key combination. (1 <= *argp* <= NSIG). (See *spawn_console*() in *linux/drivers/char/keyboard.c*.)

VT_OPENQRY
Returns the first available (non-opened) console. *argp* points to an *int* which is set to the number of the vt (1 <= **argp* <= MAX_NR_CONSOLES).

VT_GETMODE
Get mode of active vt. *argp* points to a

```
struct vt_mode {
    char  mode;    /* vt mode */
    char  waitv;   /* if set, hang on writes if not active */
    short relsig;  /* signal to raise on release op */
    short acqsig;  /* signal to raise on acquisition */
    short frsig;   /* unused (set to 0) */
};
```

which is set to the mode of the active vt. *mode* is set to one of these values:

VT_AUTO	auto vt switching
VT_PROCESS	process controls switching
VT_ACKACQ	acknowledge switch

VT_SETMODE
Set mode of active vt. *argp* points to a *struct vt_mode*.

VT_GETSTATE
Get global vt state info. *argp* points to a

```
struct vt_stat {
    unsigned short v_active;  /* active vt */
    unsigned short v_signal;  /* signal to send */
    unsigned short v_state;   /* vt bit mask */
};
```

For each vt in use, the corresponding bit in the *v_state* member is set. (Linux 1.0 through Linux 1.1.92.)

VT_RELDISP
Release a display.

VT_ACTIVATE
Switch to vt *argp* (1 <= *argp* <= MAX_NR_CONSOLES).

VT_WAITACTIVE
Wait until vt *argp* has been activated.

VT_DISALLOCATE
Deallocate the memory associated with vt *argp*. (Since Linux 1.1.54.)

VT_RESIZE
Set the kernel's idea of screensize. *argp* points to a

```
struct vt_sizes {
    unsigned short v_rows;       /* # rows */
    unsigned short v_cols;       /* # columns */
    unsigned short v_scrollsize; /* no longer used */
};
```

Note that this does not change the videomode. See *resizecons*(8)(Since Linux 1.1.54.)

VT_RESIZEX
Set the kernel's idea of various screen parameters. *argp* points to a

```
struct vt_consize {
    unsigned short v_rows;  /* number of rows */
    unsigned short v_cols;  /* number of columns */
    unsigned short v_vlin;  /* number of pixel rows
                               on screen */
    unsigned short v_clin;  /* number of pixel rows
                               per character */
    unsigned short v_vcol;  /* number of pixel columns
                               on screen */
    unsigned short v_ccol;  /* number of pixel columns
                               per character */
};
```

Any parameter may be set to zero, indicating "no change", but if multiple parameters are set, they must be self-consistent. Note that this does not change the videomode. See *resizecons*(8)(Since Linux 1.3.3.)

The action of the following ioctls depends on the first byte in the struct pointed to by *argp*, referred to here as the *subcode*. These are legal only for the superuser or the owner of the current terminal. Symbolic *subcode*s are available in <*linux/tiocl.h*> since Linux 2.5.71.

TIOCLINUX, **subcode**=**0**
Dump the screen. Disappeared in Linux 1.1.92. (With Linux 1.1.92 or later, read from */dev/vcsN* or */dev/vcsaN* instead.)

TIOCLINUX, **subcode**=**1**
Get task information. Disappeared in Linux 1.1.92.

TIOCLINUX, **subcode**=**TIOCL_SETSEL**
Set selection. *argp* points to a

```
struct {
    char  subcode;
    short xs, ys, xe, ye;
    short sel_mode;
};
```

xs and *ys* are the starting column and row. *xe* and *ye* are the ending column and row. (Upper left corner is row=column=1.) *sel_mode* is 0 for character-by-character selection, 1 for word-by-word selection, or 2 for line-by-line selection. The indicated screen characters are highlighted and saved in a kernel buffer.

Since Linux 6.7, using this subcode requires the **CAP_SYS_ADMIN** capability.

TIOCLINUX, **subcode**=**TIOCL_PASTESEL**
Paste selection. The characters in the selection buffer are written to *fd*.

Since Linux 6.7, using this subcode requires the **CAP_SYS_ADMIN** capability.

TIOCLINUX, **subcode**=**TIOCL_UNBLANKSCREEN**
Unblank the screen.

TIOCLINUX, **subcode**=**TIOCL_SELLOADLUT**
Sets contents of a 256-bit look up table defining characters in a "word", for word-by-word selection. (Since Linux 1.1.32.)

Since Linux 6.7, using this subcode requires the **CAP_SYS_ADMIN** capability.

TIOCLINUX, **subcode**=**TIOCL_GETSHIFTSTATE**
argp points to a char which is set to the value of the kernel variable *shift_state*. (Since Linux 1.1.32.)

TIOCLINUX, **subcode**=**TIOCL_GETMOUSEREPORTING**
argp points to a char which is set to the value of the kernel variable *report_mouse*. (Since Linux 1.1.33.)

TIOCLINUX, **subcode**=**8**
Dump screen width and height, cursor position, and all the character-attribute pairs. (Linux 1.1.67 through Linux 1.1.91 only. With Linux 1.1.92 or later, read from */dev/vcsa** instead.)

TIOCLINUX, **subcode**=**9**
Restore screen width and height, cursor position, and all the character-attribute pairs. (Linux 1.1.67 through Linux 1.1.91 only. With Linux 1.1.92 or later, write to */dev/vcsa** instead.)

TIOCLINUX, **subcode**=**TIOCL_SETVESABLANK**
Handles the Power Saving feature of the new generation of monitors. VESA screen blanking mode is set to *argp[1]*, which governs what screen blanking does:

0 Screen blanking is disabled.

1 The current video adapter register settings are saved, then the controller is programmed to turn off the vertical synchronization pulses. This puts the monitor into "standby" mode. If your monitor has an Off_Mode timer, then it will eventually

power down by itself.

2 The current settings are saved, then both the vertical and horizontal synchronization pulses are turned off. This puts the monitor into "off" mode. If your monitor has no Off_Mode timer, or if you want your monitor to power down immediately when the blank_timer times out, then you choose this option. (*Caution:* Powering down frequently will damage the monitor.) (Since Linux 1.1.76.)

TIOCLINUX, **subcode**=**TIOCL_SETKMSGREDIRECT**
Change target of kernel messages ("console"): by default, and if this is set to **0**, messages are written to the currently active VT. The VT to write to is a single byte following **subcode**. (Since Linux 2.5.36.)

TIOCLINUX, **subcode**=**TIOCL_GETFGCONSOLE**
Returns the number of VT currently in foreground. (Since Linux 2.5.36.)

TIOCLINUX, **subcode**=**TIOCL_SCROLLCONSOLE**
Scroll the foreground VT by the specified amount of *lines* down, or half the screen if **0**. *lines* is *(((int32_t *)&subcode) + 1). (Since Linux 2.5.67.)

TIOCLINUX, **subcode**=**TIOCL_BLANKSCREEN**
Blank the foreground VT, ignoring "pokes" (typing): can only be unblanked explicitly (by switching VTs, to text mode, etc.). (Since Linux 2.5.71.)

TIOCLINUX, **subcode**=**TIOCL_BLANKEDSCREEN**
Returns the number of VT currently blanked, **0** if none. (Since Linux 2.5.71.)

TIOCLINUX, **subcode**=**16**
Never used.

TIOCLINUX, **subcode**=**TIOCL_GETKMSGREDIRECT**
Returns target of kernel messages. (Since Linux 2.6.17.)

RETURN VALUE

On success, 0 is returned (except where indicated). On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
The file descriptor is invalid.

EINVAL
The file descriptor or *argp* is invalid.

ENOTTY
The file descriptor is not associated with a character special device, or the specified operation does not apply to it.

EPERM
Insufficient permission.

NOTES

Warning: Do not regard this man page as documentation of the Linux console ioctls. This is provided for the curious only, as an alternative to reading the source. Ioctl's are undocumented Linux internals, liable to be changed without warning. (And indeed, this page more or less describes the situation as of kernel version 1.1.94; there are many minor and not-so-minor differences with earlier versions.)

Very often, ioctls are introduced for communication between the kernel and one particular well-known program (fdisk, hdparm, setserial, tunelp, loadkeys, selection, setfont, etc.), and their behavior will be changed when required by this particular program.

Programs using these ioctls will not be portable to other versions of UNIX, will not work on older versions of Linux, and will not work on future versions of Linux.

Use POSIX functions.

SEE ALSO

dumpkeys(1), *kbd_mode*(1), *loadkeys*(1), *mknod*(1), *setleds*(1), *setmetamode*(1), *execve*(2), *fcntl*(2), *ioctl_tty*(2), *ioperm*(2), *termios*(3), *console_codes*(4), *mt*(4), *sd*(4), *tty*(4), *ttyS*(4), *vcs*(4), *vcsa*(4),

charsets(7), *mapscrn*(8), *resizecons*(8), *setfont*(8)
/usr/include/linux/kd.h, */usr/include/linux/vt.h*

NAME

ioctl_fat − manipulating the FAT filesystem

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/msdos_fs.h>     /* Definition of [V]FAT_* and
                                   ATTR_* constants*/"
#include <sys/ioctl.h>

int ioctl(int fd, FAT_IOCTL_GET_ATTRIBUTES, uint32_t *attr);
int ioctl(int fd, FAT_IOCTL_SET_ATTRIBUTES, uint32_t *attr);
int ioctl(int fd, FAT_IOCTL_GET_VOLUME_ID, uint32_t *id);
int ioctl(int fd, VFAT_IOCTL_READDIR_BOTH,
        struct __fat_dirent entry[2]);
int ioctl(int fd, VFAT_IOCTL_READDIR_SHORT,
        struct __fat_dirent entry[2]);
```

DESCRIPTION

The *ioctl*(2) system call can be used to read and write metadata of FAT filesystems that are not accessible using other system calls.

Reading and setting file attributes

Files and directories in the FAT filesystem possess an attribute bit mask that can be read with **FAT_IOCTL_GET_ATTRIBUTES** and written with **FAT_IOCTL_SET_ATTRIBUTES**.

The *fd* argument contains a file descriptor for a file or directory. It is sufficient to create the file descriptor by calling *open*(2) with the **O_RDONLY** flag.

The *attr* argument contains a pointer to a bit mask. The bits of the bit mask are:

ATTR_RO
: This bit specifies that the file or directory is read-only.

ATTR_HIDDEN
: This bit specifies that the file or directory is hidden.

ATTR_SYS
: This bit specifies that the file is a system file.

ATTR_VOLUME
: This bit specifies that the file is a volume label. This attribute is read-only.

ATTR_DIR
: This bit specifies that this is a directory. This attribute is read-only.

ATTR_ARCH
: This bit indicates that this file or directory should be archived. It is set when a file is created or modified. It is reset by an archiving system.

The zero value **ATTR_NONE** can be used to indicate that no attribute bit is set.

Reading the volume ID

FAT filesystems are identified by a volume ID. The volume ID can be read with **FAT_IOCTL_GET_VOLUME_ID**.

The *fd* argument can be a file descriptor for any file or directory of the filesystem. It is sufficient to create the file descriptor by calling *open*(2) with the **O_RDONLY** flag.

The *id* argument is a pointer to the field that will be filled with the volume ID. Typically the volume ID is displayed to the user as a group of two 16-bit fields:

```
printf("Volume ID %04x-%04x\n", id >> 16, id & 0xFFFF);
```

Reading short filenames of a directory

A file or directory on a FAT filesystem always has a short filename consisting of up to 8 capital letters, optionally followed by a period and up to 3 capital letters for the file extension. If the actual filename does not fit into this scheme, it is stored as a long filename of up to 255 UTF-16 characters.

The short filenames in a directory can be read with **VFAT_IOCTL_READDIR_SHORT**.

VFAT_IOCTL_READDIR_BOTH reads both the short and the long filenames.

The *fd* argument must be a file descriptor for a directory. It is sufficient to create the file descriptor by calling *open*(2) with the **O_RDONLY** flag. The file descriptor can be used only once to iterate over the directory entries by calling *ioctl*(2) repeatedly.

The *entry* argument is a two-element array of the following structures:

```
struct __fat_dirent {
    long            d_ino;
    __kernel_off_t  d_off;
    uint32_t short  d_reclen;
    char            d_name[256];
};
```

The first entry in the array is for the short filename. The second entry is for the long filename.

The *d_ino* and *d_off* fields are filled only for long filenames. The *d_ino* field holds the inode number of the directory. The *d_off* field holds the offset of the file entry in the directory. As these values are not available for short filenames, the user code should simply ignore them.

The field *d_reclen* contains the length of the filename in the field *d_name*. To keep backward compatibility, a length of 0 for the short filename signals that the end of the directory has been reached. However, the preferred method for detecting the end of the directory is to test the *ioctl*(2) return value. If no long filename exists, field *d_reclen* is set to 0 and *d_name* is a character string of length 0 for the long filename.

RETURN VALUE

On error, −1 is returned, and *errno* is set to indicate the error.

For **VFAT_IOCTL_READDIR_BOTH** and **VFAT_IOCTL_READDIR_SHORT** a return value of 1 signals that a new directory entry has been read and a return value of 0 signals that the end of the directory has been reached.

ERRORS

ENOENT
: This error is returned by **VFAT_IOCTL_READDIR_BOTH** and **VFAT_IOCTL_READDIR_SHORT** if the file descriptor *fd* refers to a removed, but still open directory.

ENOTDIR
: This error is returned by **VFAT_IOCTL_READDIR_BOTH** and **VFAT_IOCTL_READDIR_SHORT** if the file descriptor *fd* does not refer to a directory.

ENOTTY
: The file descriptor *fd* does not refer to an object in a FAT filesystem.

For further error values, see *ioctl*(2).

STANDARDS

Linux.

HISTORY

VFAT_IOCTL_READDIR_BOTH
VFAT_IOCTL_READDIR_SHORT
: Linux 2.0.

FAT_IOCTL_GET_ATTRIBUTES
FAT_IOCTL_SET_ATTRIBUTES
: Linux 2.6.12.

FAT_IOCTL_GET_VOLUME_ID
: Linux 3.11.

EXAMPLES

Toggling the archive flag

The following program demonstrates the usage of *ioctl*(2) to manipulate file attributes. The program reads and displays the archive attribute of a file. After inverting the value of the attribute, the program reads and displays the attribute again.

The following was recorded when applying the program for the file */mnt/user/foo*:

```
# ./toggle_fat_archive_flag /mnt/user/foo
Archive flag is set
Toggling archive flag
Archive flag is not set
```

Program source (toggle_fat_archive_flag.c)

```
#include <fcntl.h>
#include <linux/msdos_fs.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>

/*
 * Read file attributes of a file on a FAT filesystem.
 * Output the state of the archive flag.
 */
static uint32_t
readattr(int fd)
{
    int       ret;
    uint32_t  attr;

    ret = ioctl(fd, FAT_IOCTL_GET_ATTRIBUTES, &attr);
    if (ret == -1) {
        perror("ioctl");
        exit(EXIT_FAILURE);
    }

    if (attr & ATTR_ARCH)
        printf("Archive flag is set\n");
    else
        printf("Archive flag is not set\n");

    return attr;
}

int
main(int argc, char *argv[])
{
    int       fd;
    int       ret;
    uint32_t  attr;

    if (argc != 2) {
        printf("Usage: %s FILENAME\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /*
```

```
         * Read and display the FAT file attributes.
         */
        attr = readattr(fd);

        /*
         * Invert archive attribute.
         */
        printf("Toggling archive flag\n");
        attr ^= ATTR_ARCH;

        /*
         * Write the changed FAT file attributes.
         */
        ret = ioctl(fd, FAT_IOCTL_SET_ATTRIBUTES, &attr);
        if (ret == -1) {
            perror("ioctl");
            exit(EXIT_FAILURE);
        }

        /*
         * Read and display the FAT file attributes.
         */
        readattr(fd);

        close(fd);

        exit(EXIT_SUCCESS);
    }
```

Reading the volume ID

The following program demonstrates the use of *ioctl*(2) to display the volume ID of a FAT filesystem.

The following output was recorded when applying the program for directory */mnt/user*:

```
$ ./display_fat_volume_id /mnt/user
Volume ID 6443-6241
```

Program source (display_fat_volume_id.c)

```
#include <fcntl.h>
#include <linux/msdos_fs.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int       fd;
    int       ret;
    uint32_t  id;

    if (argc != 2) {
        printf("Usage: %s FILENAME\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
```

```
            perror("open");
            exit(EXIT_FAILURE);
        }

        /*
         * Read volume ID.
         */
        ret = ioctl(fd, FAT_IOCTL_GET_VOLUME_ID, &id);
        if (ret == -1) {
            perror("ioctl");
            exit(EXIT_FAILURE);
        }

        /*
         * Format the output as two groups of 16 bits each.
         */
        printf("Volume ID %04x-%04x\n", id >> 16, id & 0xFFFF);

        close(fd);

        exit(EXIT_SUCCESS);
    }
```

Listing a directory

The following program demonstrates the use of *ioctl*(2) to list a directory.

The following was recorded when applying the program to the directory */mnt/user*:

```
$ ./fat_dir /mnt/user
. -> ''
.. -> ''
ALONGF~1.TXT -> 'a long filename.txt'
UPPER.TXT -> ''
LOWER.TXT -> 'lower.txt'
```

Program source

```
#include <fcntl.h>
#include <linux/msdos_fs.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int                 fd;
    int                 ret;
    struct __fat_dirent  entry[2];

    if (argc != 2) {
        printf("Usage: %s DIRECTORY\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /*
     * Open file descriptor for the directory.
     */
    fd = open(argv[1], O_RDONLY | O_DIRECTORY);
    if (fd == -1) {
        perror("open");
```

```
            exit(EXIT_FAILURE);
        }

        for (;;) {

            /*
             * Read next directory entry.
             */
            ret = ioctl(fd, VFAT_IOCTL_READDIR_BOTH, entry);

            /*
             * If an error occurs, the return value is -1.
             * If the end of the directory list has been reached,
             * the return value is 0.
             * For backward compatibility the end of the directory
             * list is also signaled by d_reclen == 0.
             */
            if (ret < 1)
                break;

            /*
             * Write both the short name and the long name.
             */
            printf("%s -> '%s'\n", entry[0].d_name, entry[1].d_name);
        }

        if (ret == -1) {
            perror("VFAT_IOCTL_READDIR_BOTH");
            exit(EXIT_FAILURE);
        }

        /*
         * Close the file descriptor.
         */
        close(fd);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ioctl(2)

NAME

ioctl_ficlonerange, ioctl_ficlone − share some the data of one file with another file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fs.h> /* Definition of **FICLONE*** constants */
#include <sys/ioctl.h>

int ioctl(int *dest_fd***, FICLONERANGE, struct file_clone_range ***arg***);**
int ioctl(int *dest_fd***, FICLONE, int** *src_fd***);**

DESCRIPTION

If a filesystem supports files sharing physical storage between multiple files ("reflink"), this *ioctl*(2) operation can be used to make some of the data in the *src_fd* file appear in the *dest_fd* file by sharing the underlying storage, which is faster than making a separate physical copy of the data. Both files must reside within the same filesystem. If a file write should occur to a shared region, the filesystem must ensure that the changes remain private to the file being written. This behavior is commonly referred to as "copy on write".

This ioctl reflinks up to *src_length* bytes from file descriptor *src_fd* at offset *src_offset* into the file *dest_fd* at offset *dest_offset*, provided that both are files. If *src_length* is zero, the ioctl reflinks to the end of the source file. This information is conveyed in a structure of the following form:

```
struct file_clone_range {
    __s64 src_fd;
    __u64 src_offset;
    __u64 src_length;
    __u64 dest_offset;
};
```

Clones are atomic with regards to concurrent writes, so no locks need to be taken to obtain a consistent cloned copy.

The **FICLONE** ioctl clones entire files.

RETURN VALUE

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Error codes can be one of, but are not limited to, the following:

EBADF
src_fd is not open for reading; *dest_fd* is not open for writing or is open for append-only writes; or the filesystem which *src_fd* resides on does not support reflink.

EINVAL
The filesystem does not support reflinking the ranges of the given files. This error can also appear if either file descriptor represents a device, FIFO, or socket. Disk filesystems generally require the offset and length arguments to be aligned to the fundamental block size. XFS and Btrfs do not support overlapping reflink ranges in the same file.

EISDIR
One of the files is a directory and the filesystem does not support shared regions in directories.

EOPNOTSUPP
This can appear if the filesystem does not support reflinking either file descriptor, or if either file descriptor refers to special inodes.

EPERM
dest_fd is immutable.

ETXTBSY
One of the files is a swap file. Swap files cannot share storage.

EXDEV
dest_fd and *src_fd* are not on the same mounted filesystem.

STANDARDS

Linux.

HISTORY

Linux 4.5.

They were previously known as **BTRFS_IOC_CLONE** and **BTRFS_IOC_CLONE_RANGE**, and were private to Btrfs.

NOTES

Because a copy-on-write operation requires the allocation of new storage, the *fallocate*(2) operation may unshare shared blocks to guarantee that subsequent writes will not fail because of lack of disk space.

SEE ALSO

ioctl(2)

NAME

ioctl_fideduperange − share some the data of one file with another file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fs.h> /* Definition of **FIDEDUPERANGE** and
FILE_DEDUPE_* constants***/**
#include <sys/ioctl.h>

int ioctl(int *src_fd***, FIDEDUPERANGE, struct file_dedupe_range ****arg***);**

DESCRIPTION

If a filesystem supports files sharing physical storage between multiple files, this *ioctl*(2) operation can be used to make some of the data in the **src_fd** file appear in the **dest_fd** file by sharing the underlying storage if the file data is identical ("deduplication"). Both files must reside within the same filesystem. This reduces storage consumption by allowing the filesystem to store one shared copy of the data. If a file write should occur to a shared region, the filesystem must ensure that the changes remain private to the file being written. This behavior is commonly referred to as "copy on write".

This ioctl performs the "compare and share if identical" operation on up to *src_length* bytes from file descriptor *src_fd* at offset *src_offset*. This information is conveyed in a structure of the following form:

```
struct file_dedupe_range {
    __u64 src_offset;
    __u64 src_length;
    __u16 dest_count;
    __u16 reserved1;
    __u32 reserved2;
    struct file_dedupe_range_info info[0];
};
```

Deduplication is atomic with regards to concurrent writes, so no locks need to be taken to obtain a consistent deduplicated copy.

The fields *reserved1* and *reserved2* must be zero.

Destinations for the deduplication operation are conveyed in the array at the end of the structure. The number of destinations is given in *dest_count*, and the destination information is conveyed in the following form:

```
struct file_dedupe_range_info {
    __s64 dest_fd;
    __u64 dest_offset;
    __u64 bytes_deduped;
    __s32 status;
    __u32 reserved;
};
```

Each deduplication operation targets *src_length* bytes in file descriptor *dest_fd* at offset *dest_offset*. The field *reserved* must be zero. During the call, *src_fd* must be open for reading and *dest_fd* must be open for writing. The combined size of the struct *file_dedupe_range* and the struct *file_dedupe_range_info* array must not exceed the system page size. The maximum size of *src_length* is filesystem dependent and is typically 16 MiB. This limit will be enforced silently by the filesystem. By convention, the storage used by *src_fd* is mapped into *dest_fd* and the previous contents in *dest_fd* are freed.

Upon successful completion of this ioctl, the number of bytes successfully deduplicated is returned in *bytes_deduped* and a status code for the deduplication operation is returned in *status*. If even a single byte in the range does not match, the deduplication operation request will be ignored and *status* set to **FILE_DEDUPE_RANGE_DIFFERS**. The *status* code is set to **FILE_DEDUPE_RANGE_SAME** for success, a negative error code in case of error, or **FILE_DEDUPE_RANGE_DIFFERS** if the data did not match.

RETURN VALUE

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Possible errors include (but are not limited to) the following:

EBADF
src_fd is not open for reading; *dest_fd* is not open for writing or is open for append-only writes; or the filesystem which *src_fd* resides on does not support deduplication.

EINVAL
The filesystem does not support deduplicating the ranges of the given files. This error can also appear if either file descriptor represents a device, FIFO, or socket. Disk filesystems generally require the offset and length arguments to be aligned to the fundamental block size. Neither Btrfs nor XFS support overlapping deduplication ranges in the same file.

EISDIR
One of the files is a directory and the filesystem does not support shared regions in directories.

ENOMEM
The kernel was unable to allocate sufficient memory to perform the operation or *dest_count* is so large that the input argument description spans more than a single page of memory.

EOPNOTSUPP
This can appear if the filesystem does not support deduplicating either file descriptor, or if either file descriptor refers to special inodes.

EPERM
dest_fd is immutable.

ETXTBSY
One of the files is a swap file. Swap files cannot share storage.

EXDEV
dest_fd and *src_fd* are not on the same mounted filesystem.

VERSIONS

Some filesystems may limit the amount of data that can be deduplicated in a single call.

STANDARDS

Linux.

HISTORY

Linux 4.5.

It was previously known as **BTRFS_IOC_FILE_EXTENT_SAME** and was private to Btrfs.

NOTES

Because a copy-on-write operation requires the allocation of new storage, the *fallocate*(2) operation may unshare shared blocks to guarantee that subsequent writes will not fail because of lack of disk space.

SEE ALSO

ioctl(2)

NAME

ioctl_fslabel − get or set a filesystem label

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fs.h> /* Definition of ***FSLABEL*** constants */
#include <sys/ioctl.h>

int ioctl(int *fd***, FS_IOC_GETFSLABEL, char** *label***[FSLABEL_MAX]);**
int ioctl(int *fd***, FS_IOC_SETFSLABEL, char** *label***[FSLABEL_MAX]);**

DESCRIPTION

If a filesystem supports online label manipulation, these *ioctl*(2) operations can be used to get or set the filesystem label for the filesystem on which *fd* resides. The **FS_IOC_SETFSLABEL** operation requires privilege (**CAP_SYS_ADMIN**).

RETURN VALUE

On success zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Possible errors include (but are not limited to) the following:

EFAULT
label references an inaccessible memory area.

EINVAL
The specified label exceeds the maximum label length for the filesystem.

ENOTTY
This can appear if the filesystem does not support online label manipulation.

EPERM
The calling process does not have sufficient permissions to set the label.

STANDARDS

Linux.

HISTORY

Linux 4.18.

They were previously known as **BTRFS_IOC_GET_FSLABEL** and **BTRFS_IOC_SET_FSLABEL** and were private to Btrfs.

NOTES

The maximum string length for this interface is **FSLABEL_MAX**, including the terminating null byte ('\0'). Filesystems have differing maximum label lengths, which may or may not include the terminating null. The string provided to **FS_IOC_SETFSLABEL** must always be null-terminated, and the string returned by **FS_IOC_GETFSLABEL** will always be null-terminated.

SEE ALSO

ioctl(2), *blkid*(8)

NAME

ioctl_getfsmap − retrieve the physical layout of the filesystem

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fsmap.h> /* Definition of **FS_IOC_GETFSMAP**,
FM?_OF_*, and ***FMR_OWN_*** constants */
#include <sys/ioctl.h>

int ioctl(int *fd***, FS_IOC_GETFSMAP, struct fsmap_head *** *arg***);**

DESCRIPTION

This *ioctl*(2) operation retrieves physical extent mappings for a filesystem. This information can be used to discover which files are mapped to a physical block, examine free space, or find known bad blocks, among other things.

The sole argument to this operation should be a pointer to a single *struct fsmap_head*:

```
struct fsmap {
    __u32 fmr_device;      /* Device ID */
    __u32 fmr_flags;       /* Mapping flags */
    __u64 fmr_physical;    /* Device offset of segment */
    __u64 fmr_owner;       /* Owner ID */
    __u64 fmr_offset;      /* File offset of segment */
    __u64 fmr_length;      /* Length of segment */
    __u64 fmr_reserved[3]; /* Must be zero */
};

struct fsmap_head {
    __u32 fmh_iflags;       /* Control flags */
    __u32 fmh_oflags;       /* Output flags */
    __u32 fmh_count;        /* # of entries in array incl. input */
    __u32 fmh_entries;      /* # of entries filled in (output) */
    __u64 fmh_reserved[6];  /* Must be zero */

    struct fsmap fmh_keys[2];  /* Low and high keys for
                                  the mapping search */
    struct fsmap fmh_recs[];   /* Returned records */
};
```

The two *fmh_keys* array elements specify the lowest and highest reverse-mapping key for which the application would like physical mapping information. A reverse mapping key consists of the tuple (device, block, owner, offset). The owner and offset fields are part of the key because some filesystems support sharing physical blocks between multiple files and therefore may return multiple mappings for a given physical block.

Filesystem mappings are copied into the *fmh_recs* array, which immediately follows the header data.

Fields of struct fsmap_head

The *fmh_iflags* field is a bit mask passed to the kernel to alter the output. No flags are currently defined, so the caller must set this value to zero.

The *fmh_oflags* field is a bit mask of flags set by the kernel concerning the returned mappings. If **FMH_OF_DEV_T** is set, then the *fmr_device* field represents a *dev_t* structure containing the major and minor numbers of the block device.

The *fmh_count* field contains the number of elements in the array being passed to the kernel. If this value is 0, *fmh_entries* will be set to the number of records that would have been returned had the array been large enough; no mapping information will be returned.

The *fmh_entries* field contains the number of elements in the *fmh_recs* array that contain useful information.

The *fmh_reserved* fields must be set to zero.

Keys

The two key records in *fsmap_head.fmh_keys* specify the lowest and highest extent records in the key-space that the caller wants returned. A filesystem that can share blocks between files likely requires the tuple (*device*, *physical*, *owner*, *offset*, *flags*) to uniquely index any filesystem mapping record. Classic non-sharing filesystems might be able to identify any record with only (*device*, *physical*, *flags*). For example, if the low key is set to (8:0, 36864, 0, 0, 0), the filesystem will only return records for extents starting at or above 36 KiB on disk. If the high key is set to (8:0, 1048576, 0, 0, 0), only records below 1 MiB will be returned. The format of *fmr_device* in the keys must match the format of the same field in the output records, as defined below. By convention, the field *fsmap_head.fmh_keys[0]* must contain the low key and *fsmap_head.fmh_keys[1]* must contain the high key for the operation.

For convenience, if *fmr_length* is set in the low key, it will be added to *fmr_block* or *fmr_offset* as appropriate. The caller can take advantage of this subtlety to set up subsequent calls by copying *fsmap_head.fmh_recs[fsmap_head.fmh_entries − 1]* into the low key. The function *fsmap_advance* (defined in *linux/fsmap.h*) provides this functionality.

Fields of struct fsmap

The *fmr_device* field uniquely identifies the underlying storage device. If the **FMH_OF_DEV_T** flag is set in the header's *fmh_oflags* field, this field contains a *dev_t* from which major and minor numbers can be extracted. If the flag is not set, this field contains a value that must be unique for each unique storage device.

The *fmr_physical* field contains the disk address of the extent in bytes.

The *fmr_owner* field contains the owner of the extent. This is an inode number unless **FMR_OF_SPECIAL_OWNER** is set in the *fmr_flags* field, in which case the value is determined by the filesystem. See the section below about owner values for more details.

The *fmr_offset* field contains the logical address in the mapping record in bytes. This field has no meaning if the **FMR_OF_SPECIAL_OWNER** or **FMR_OF_EXTENT_MAP** flags are set in *fmr_flags*.

The *fmr_length* field contains the length of the extent in bytes.

The *fmr_flags* field is a bit mask of extent state flags. The bits are:

FMR_OF_PREALLOC
The extent is allocated but not yet written.

FMR_OF_ATTR_FORK
This extent contains extended attribute data.

FMR_OF_EXTENT_MAP
This extent contains extent map information for the owner.

FMR_OF_SHARED
Parts of this extent may be shared.

FMR_OF_SPECIAL_OWNER
The *fmr_owner* field contains a special value instead of an inode number.

FMR_OF_LAST
This is the last record in the data set.

The *fmr_reserved* field will be set to zero.

Owner values

Generally, the value of the *fmr_owner* field for non-metadata extents should be an inode number. However, filesystems are under no obligation to report inode numbers; they may instead report **FMR_OWN_UNKNOWN** if the inode number cannot easily be retrieved, if the caller lacks sufficient privilege, if the filesystem does not support stable inode numbers, or for any other reason. If a filesystem wishes to condition the reporting of inode numbers based on process capabilities, it is strongly urged that the **CAP_SYS_ADMIN** capability be used for this purpose.

The following special owner values are generic to all filesystems:

FMR_OWN_FREE
Free space.

FMR_OWN_UNKNOWN
This extent is in use but its owner is not known or not easily retrieved.

FMR_OWN_METADATA
This extent is filesystem metadata.

XFS can return the following special owner values:

XFS_FMR_OWN_FREE
Free space.

XFS_FMR_OWN_UNKNOWN
This extent is in use but its owner is not known or not easily retrieved.

XFS_FMR_OWN_FS
Static filesystem metadata which exists at a fixed address. These are the AG superblock, the AGF, the AGFL, and the AGI headers.

XFS_FMR_OWN_LOG
The filesystem journal.

XFS_FMR_OWN_AG
Allocation group metadata, such as the free space btrees and the reverse mapping btrees.

XFS_FMR_OWN_INOBT
The inode and free inode btrees.

XFS_FMR_OWN_INODES
Inode records.

XFS_FMR_OWN_REFC
Reference count information.

XFS_FMR_OWN_COW
This extent is being used to stage a copy-on-write.

XFS_FMR_OWN_DEFECTIVE:
This extent has been marked defective either by the filesystem or the underlying device.

ext4 can return the following special owner values:

EXT4_FMR_OWN_FREE
Free space.

EXT4_FMR_OWN_UNKNOWN
This extent is in use but its owner is not known or not easily retrieved.

EXT4_FMR_OWN_FS
Static filesystem metadata which exists at a fixed address. This is the superblock and the group descriptors.

EXT4_FMR_OWN_LOG
The filesystem journal.

EXT4_FMR_OWN_INODES
Inode records.

EXT4_FMR_OWN_BLKBM
Block bit map.

EXT4_FMR_OWN_INOBM
Inode bit map.

RETURN VALUE

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The error placed in *errno* can be one of, but is not limited to, the following:

EBADF
fd is not open for reading.

EBADMSG
The filesystem has detected a checksum error in the metadata.

EFAULT
The pointer passed in was not mapped to a valid memory address.

EINVAL
The array is not long enough, the keys do not point to a valid part of the filesystem, the low key points to a higher point in the filesystem's physical storage address space than the high key, or a nonzero value was passed in one of the fields that must be zero.

ENOMEM
Insufficient memory to process the operation.

EOPNOTSUPP
The filesystem does not support this operation.

EUCLEAN
The filesystem metadata is corrupt and needs repair.

STANDARDS
Linux.

Not all filesystems support it.

HISTORY
Linux 4.12.

EXAMPLES
See *io/fsmap.c* in the *xfsprogs* distribution for a sample program.

SEE ALSO
ioctl(2)

NAME

ioctl_iflags − ioctl() operations for inode flags

DESCRIPTION

Various Linux filesystems support the notion of *inode flags*—attributes that modify the semantics of files and directories. These flags can be retrieved and modified using two *ioctl*(2) operations:

```
int attr;
fd = open("pathname", ...);

ioctl(fd, FS_IOC_GETFLAGS, &attr);  /* Place current flags
                                       in 'attr' */
attr |= FS_NOATIME_FL;              /* Tweak returned bit mask */
ioctl(fd, FS_IOC_SETFLAGS, &attr);  /* Update flags for inode
                                       referred to by 'fd' */
```

The *lsattr*(1) and *chattr*(1) shell commands provide interfaces to these two operations, allowing a user to view and modify the inode flags associated with a file.

The following flags are supported (shown along with the corresponding letter used to indicate the flag by *lsattr*(1) and *chattr*(1)):

FS_APPEND_FL 'a'
: The file can be opened only with the **O_APPEND** flag. (This restriction applies even to the superuser.) Only a privileged process (**CAP_LINUX_IMMUTABLE**) can set or clear this attribute.

FS_COMPR_FL 'c'
: Store the file in a compressed format on disk. This flag is *not* supported by most of the mainstream filesystem implementations; one exception is *btrfs*(5)

FS_DIRSYNC_FL 'D' (since Linux 2.6.0)
: Write directory changes synchronously to disk. This flag provides semantics equivalent to the *mount*(2) **MS_DIRSYNC** option, but on a per-directory basis. This flag can be applied only to directories.

FS_IMMUTABLE_FL 'i'
: The file is immutable: no changes are permitted to the file contents or metadata (permissions, timestamps, ownership, link count, and so on). (This restriction applies even to the superuser.) Only a privileged process (**CAP_LINUX_IMMUTABLE**) can set or clear this attribute.

FS_JOURNAL_DATA_FL 'j'
: Enable journaling of file data on *ext3*(5) and *ext4*(5) filesystems. On a filesystem that is journaling in *ordered* or *writeback* mode, a privileged (**CAP_SYS_RESOURCE**) process can set this flag to enable journaling of data updates on a per-file basis.

FS_NOATIME_FL 'A'
: Don’t update the file last access time when the file is accessed. This can provide I/O performance benefits for applications that do not care about the accuracy of this timestamp. This flag provides functionality similar to the *mount*(2) **MS_NOATIME** flag, but on a per-file basis.

FS_NOCOW_FL 'C' (since Linux 2.6.39)
: The file will not be subject to copy-on-write updates. This flag has an effect only on filesystems that support copy-on-write semantics, such as Btrfs. See *chattr*(1) and *btrfs*(5)

FS_NODUMP_FL 'd'
: Don’t include this file in backups made using *dump*(8)

FS_NOTAIL_FL 't'
: This flag is supported only on Reiserfs. It disables the Reiserfs tail-packing feature, which tries to pack small files (and the final fragment of larger files) into the same disk block as the file metadata.

FS_PROJINHERIT_FL 'P' (since Linux 4.5)
: Inherit the quota project ID. Files and subdirectories will inherit the project ID of the directory. This flag can be applied only to directories.

FS_SECRM_FL 's'
Mark the file for secure deletion. This feature is not implemented by any filesystem, since the task of securely erasing a file from a recording medium is surprisingly difficult.

FS_SYNC_FL 'S'
Make file updates synchronous. For files, this makes all writes synchronous (as though all opens of the file were with the **O_SYNC** flag). For directories, this has the same effect as the **FS_DIRSYNC_FL** flag.

FS_TOPDIR_FL 'T'
Mark a directory for special treatment under the Orlov block-allocation strategy. See *chattr*(1) for details. This flag can be applied only to directories and has an effect only for ext2, ext3, and ext4.

FS_UNRM_FL 'u'
Allow the file to be undeleted if it is deleted. This feature is not implemented by any filesystem, since it is possible to implement file-recovery mechanisms outside the kernel.

In most cases, when any of the above flags is set on a directory, the flag is inherited by files and subdirectories created inside that directory. Exceptions include **FS_TOPDIR_FL**, which is not inheritable, and **FS_DIRSYNC_FL**, which is inherited only by subdirectories.

STANDARDS
Linux.

NOTES
In order to change the inode flags of a file using the **FS_IOC_SETFLAGS** operation, the effective user ID of the caller must match the owner of the file, or the caller must have the **CAP_FOWNER** capability.

The type of the argument given to the **FS_IOC_GETFLAGS** and **FS_IOC_SETFLAGS** operations is *int **, notwithstanding the implication in the kernel source file *include/uapi/linux/fs.h* that the argument is *long **.

SEE ALSO
chattr(1), *lsattr*(1), *mount*(2), *btrfs*(5), *ext4*(5), *xfs*(5), *xattr*(7), *mount*(8)

NAME

ioctl_ns − ioctl() operations for Linux namespaces

DESCRIPTION

Discovering namespace relationships

The following *ioctl*(2) operations are provided to allow discovery of namespace relationships (see *user_namespaces*(7) and *pid_namespaces*(7)). The form of the calls is:

```
new_fd = ioctl(fd, op);
```

In each case, *fd* refers to a */proc/*pid*/ns/** file. Both operations return a new file descriptor on success.

NS_GET_USERNS (since Linux 4.9)
: Returns a file descriptor that refers to the owning user namespace for the namespace referred to by *fd*.

NS_GET_PARENT (since Linux 4.9)
: Returns a file descriptor that refers to the parent namespace of the namespace referred to by *fd*. This operation is valid only for hierarchical namespaces (i.e., PID and user namespaces). For user namespaces, **NS_GET_PARENT** is synonymous with **NS_GET_USERNS**.

The new file descriptor returned by these operations is opened with the **O_RDONLY** and **O_CLOEXEC** (close-on-exec; see *fcntl*(2)) flags.

By applying *fstat*(2) to the returned file descriptor, one obtains a *stat* structure whose *st_dev* (resident device) and *st_ino* (inode number) fields together identify the owning/parent namespace. This inode number can be matched with the inode number of another */proc/*pid*/ns/*{*pid*,*user*} file to determine whether that is the owning/parent namespace.

Either of these *ioctl*(2) operations can fail with the following errors:

EPERM
: The requested namespace is outside of the caller's namespace scope. This error can occur if, for example, the owning user namespace is an ancestor of the caller's current user namespace. It can also occur on attempts to obtain the parent of the initial user or PID namespace.

ENOTTY
: The operation is not supported by this kernel version.

Additionally, the **NS_GET_PARENT** operation can fail with the following error:

EINVAL
: *fd* refers to a nonhierarchical namespace.

See the EXAMPLES section for an example of the use of these operations.

Discovering the namespace type

The **NS_GET_NSTYPE** operation (available since Linux 4.11) can be used to discover the type of namespace referred to by the file descriptor *fd*:

```
nstype = ioctl(fd, NS_GET_NSTYPE);
```

fd refers to a */proc/*pid*/ns/** file.

The return value is one of the **CLONE_NEW*** values that can be specified to *clone*(2) or *unshare*(2) in order to create a namespace.

Discovering the owner of a user namespace

The **NS_GET_OWNER_UID** operation (available since Linux 4.11) can be used to discover the owner user ID of a user namespace (i.e., the effective user ID of the process that created the user namespace). The form of the call is:

```
uid_t uid;
ioctl(fd, NS_GET_OWNER_UID, &uid);
```

fd refers to a */proc/*pid*/ns/user* file.

The owner user ID is returned in the *uid_t* pointed to by the third argument.

This operation can fail with the following error:

EINVAL
fd does not refer to a user namespace.

ERRORS

Any of the above **ioctl**() operations can return the following errors:

ENOTTY
fd does not refer to a */proc/*pid*/ns/** file.

STANDARDS

Linux.

EXAMPLES

The example shown below uses the *ioctl*(2) operations described above to perform simple discovery of namespace relationships. The following shell sessions show various examples of the use of this program.

Trying to get the parent of the initial user namespace fails, since it has no parent:

```
$ ./ns_show /proc/self/ns/user p
The parent namespace is outside your namespace scope
```

Create a process running *sleep*(1) that resides in new user and UTS namespaces, and show that the new UTS namespace is associated with the new user namespace:

```
$ unshare -Uu sleep 1000 &
[1] 23235
$ ./ns_show /proc/23235/ns/uts u
Device/Inode of owning user namespace is: [0,3] / 4026532448
$ readlink /proc/23235/ns/user
user:[4026532448]
```

Then show that the parent of the new user namespace in the preceding example is the initial user namespace:

```
$ readlink /proc/self/ns/user
user:[4026531837]
$ ./ns_show /proc/23235/ns/user p
Device/Inode of parent namespace is: [0,3] / 4026531837
```

Start a shell in a new user namespace, and show that from within this shell, the parent user namespace can't be discovered. Similarly, the UTS namespace (which is associated with the initial user namespace) can't be discovered.

```
$ PS1="sh2$ " unshare -U bash
sh2$ ./ns_show /proc/self/ns/user p
The parent namespace is outside your namespace scope
sh2$ ./ns_show /proc/self/ns/uts u
The owning user namespace is outside your namespace scope
```

Program source

```
/* ns_show.c

   Licensed under the GNU General Public License v2 or later.
*/
#include <errno.h>
#include <fcntl.h>
#include <linux/nsfs.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
```

```
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int          fd, userns_fd, parent_fd;
    struct stat  sb;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s /proc/[pid]/ns/[file] [p|u]\n",
                argv[0]);
        fprintf(stderr, "\nDisplay the result of one or both "
                "of NS_GET_USERNS (u) or NS_GET_PARENT (p)\n"
                "for the specified /proc/[pid]/ns/[file]. If neither "
                "'p' nor 'u' is specified,\n"
                "NS_GET_USERNS is the default.\n");
        exit(EXIT_FAILURE);
    }

    /* Obtain a file descriptor for the 'ns' file specified
       in argv[1]. */

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Obtain a file descriptor for the owning user namespace and
       then obtain and display the inode number of that namespace. */

    if (argc < 3 || strchr(argv[2], 'u')) {
        userns_fd = ioctl(fd, NS_GET_USERNS);

        if (userns_fd == -1) {
            if (errno == EPERM)
                printf("The owning user namespace is outside "
                       "your namespace scope\n");
            else
               perror("ioctl-NS_GET_USERNS");
            exit(EXIT_FAILURE);
         }

        if (fstat(userns_fd, &sb) == -1) {
            perror("fstat-userns");
            exit(EXIT_FAILURE);
        }
        printf("Device/Inode of owning user namespace is: "
               "[%x,%x] / %ju\n",
               major(sb.st_dev),
               minor(sb.st_dev),
               (uintmax_t) sb.st_ino);

        close(userns_fd);
    }

    /* Obtain a file descriptor for the parent namespace and
       then obtain and display the inode number of that namespace. */
```

```
        if (argc > 2 && strchr(argv[2], 'p')) {
            parent_fd = ioctl(fd, NS_GET_PARENT);

            if (parent_fd == -1) {
                if (errno == EINVAL)
                    printf("Can' get parent namespace of a "
                           "nonhierarchical namespace\n");
                else if (errno == EPERM)
                    printf("The parent namespace is outside "
                           "your namespace scope\n");
                else
                    perror("ioctl-NS_GET_PARENT");
                exit(EXIT_FAILURE);
            }

            if (fstat(parent_fd, &sb) == -1) {
                perror("fstat-parentns");
                exit(EXIT_FAILURE);
            }
            printf("Device/Inode of parent namespace is: [%x,%x] / %ju\n",
                   major(sb.st_dev),
                   minor(sb.st_dev),
                   (uintmax_t) sb.st_ino);

            close(parent_fd);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fstat(2), *ioctl*(2), *proc*(5), *namespaces*(7)

NAME

ioctl_pagemap_scan − get and/or clear page flags

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fs.h> /* Definition of **struct pm_scan_arg**,
struct page_region, and **PAGE_IS_*** constants */
#include <sys/ioctl.h>

int ioctl(int *pagemap_fd***, PAGEMAP_SCAN, struct pm_scan_arg *****arg***);**

DESCRIPTION

This *ioctl*(2) is used to get and optionally clear some specific flags from page table entries. The information is returned with **PAGE_SIZE** granularity.

To start tracking the written state (flag) of a page or range of memory, the **UFFD_FEATURE_WP_ASYNC** must be enabled by **UFFDIO_API** *ioctl*(2) on **userfaultfd** and memory range must be registered with **UFFDIO_REGISTER** *ioctl*(2) in **UFFDIO_REGISTER_MODE_WP** mode.

Supported page flags

The following page table entry flags are supported:

PAGE_IS_WPALLOWED
The page has asynchronous write-protection enabled.

PAGE_IS_WRITTEN
The page has been written to from the time it was write protected.

PAGE_IS_FILE
The page is file backed.

PAGE_IS_PRESENT
The page is present in the memory.

PAGE_IS_SWAPPED
The page is swapped.

PAGE_IS_PFNZERO
The page has zero PFN.

PAGE_IS_HUGE
The page is THP or Hugetlb backed.

Supported operations

The get operation is always performed if the output buffer is specified. The other operations are as following:

PM_SCAN_WP_MATCHING
Write protect the matched pages.

PM_SCAN_CHECK_WPASYNC
Abort the scan when a page is found which doesn't have the Userfaultfd Asynchronous Write protection enabled.

The *struct pm_scan_arg* argument

```
struct pm_scan_arg {
    __u64  size;
    __u64  flags;
    __u64  start;
    __u64  end;
    __u64  walk_end;
    __u64  vec;
    __u64  vec_len;
    __u64  max_pages
    __u64  category_inverted;
    __u64  category_mask;
    __u64  category_anyof_mask
```

```
    __u64  return_mask;
};
```

size This field should be set to the size of the structure in bytes, as in *sizeof(struct pm_scan_arg)*.

flags The operations to be performed are specified in it.

start The starting address of the scan is specified in it.

end The ending address of the scan is specified in it.

walk_end
The kernel returns the scan's ending address in it. The *walk_end* equal to *end* means that scan has completed on the entire range.

vec The address of *page_region* array for output.

```
struct page_region {
    __u64  start;
    __u64  end;
    __u64  categories;
};
```

vec_len
The length of the *page_region* struct array.

max_pages
It is the optional limit for the number of output pages required.

category_inverted
PAGE_IS_* categories which values match if 0 instead of 1.

category_mask
Skip pages for which any **PAGE_IS_*** category doesn't match.

category_anyof_mask
Skip pages for which no **PAGE_IS_*** category matches.

return_mask
PAGE_IS_* categories that are to be reported in *page_region*.

RETURN VALUE
On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS
Error codes can be one of, but are not limited to, the following:

EINVAL
Invalid arguments i.e., invalid *size* of the argument, invalid *flags*, invalid *categories*, the *start* address isn't aligned with **PAGE_SIZE**, or *vec_len* is specified when *vec* is NULL.

EFAULT
Invalid *arg* pointer, invalid *vec* pointer, or invalid address range specified by *start* and *end*.

ENOMEM
No memory is available.

EINTR
Fetal signal is pending.

STANDARDS
Linux.

HISTORY
Linux 6.7.

SEE ALSO
ioctl(2)

NAME

ioctl_pipe − ioctl() operations for General notification mechanism

SYNOPSIS

#include <linux/watch_queue.h> /* Definition of **IOC_WATCH_QUEUE_*** */
#include <sys/ioctl.h>

int ioctl(int *pipefd***[1], IOC_WATCH_QUEUE_SET_SIZE, int** *size***);**
int ioctl(int *pipefd***[1], IOC_WATCH_QUEUE_SET_FILTER,**
struct watch_notification_filter * *filter***);**

DESCRIPTION

The following *ioctl*(2) operations are provided to set up general notification queue parameters. The notification queue is built on the top of a *pipe*(2) opened with the **O_NOTIFICATION_PIPE** flag.

IOC_WATCH_QUEUE_SET_SIZE (since Linux 5.8)
Preallocates the pipe buffer memory so that it can fit *size* notification messages. Currently, *size* must be between 1 and 512.

IOC_WATCH_QUEUE_SET_FILTER (since Linux 5.8)
Watch queue filter can limit events that are received. Filters are passed in a *struct watch_notification_filter* and each filter is described by a *struct watch_notification_type_filter* structure.

```
struct watch_notification_filter {
        __u32   nr_filters;
        __u32   __reserved;
        struct watch_notification_type_filter filters[];
};

struct watch_notification_type_filter {
        __u32   type;
        __u32   info_filter;
        __u32   info_mask;
        __u32   subtype_filter[8];
};
```

SEE ALSO

pipe(2), *ioctl*(2)

NAME

ioctl_tty − ioctls for terminals and serial lines

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/ioctl.h>
#include <asm/termbits.h>  /* Definition of struct termios,
                              struct termios2, and
                              Bnnn, BOTHER, CBAUD, CLOCAL,
                              TC*{FLUSH,ON,OFF} and other constants */

int ioctl(int fd, int op, ...);
```

DESCRIPTION

The *ioctl*(2) call for terminals and serial ports accepts many possible operation arguments. Most require a third argument, of varying type, here called *argp* or *arg*.

Use of **ioctl**() makes for nonportable programs. Use the POSIX interface described in *termios*(3) whenever possible.

Please note that **struct termios** from <*asm/termbits.h*> is different and incompatible with **struct termios** from <*termios.h*>. These ioctl calls require **struct termios** from <*asm/termbits.h*>.

Get and set terminal attributes

TCGETS

Argument: **struct termios *****argp*

Equivalent to *tcgetattr(fd, argp)*.

Get the current serial port settings.

TCSETS

Argument: **const struct termios *****argp*

Equivalent to *tcsetattr(fd, TCSANOW, argp)*.

Set the current serial port settings.

TCSETSW

Argument: **const struct termios *****argp*

Equivalent to *tcsetattr(fd, TCSADRAIN, argp)*.

Allow the output buffer to drain, and set the current serial port settings.

TCSETSF

Argument: **const struct termios *****argp*

Equivalent to *tcsetattr(fd, TCSAFLUSH, argp)*.

Allow the output buffer to drain, discard pending input, and set the current serial port settings.

The following four ioctls, added in Linux 2.6.20, are just like **TCGETS**, **TCSETS**, **TCSETSW**, **TCSETSF**, except that they take a *struct termios2 ** instead of a *struct termios **. If the structure member **c_cflag** contains the flag **BOTHER**, then the baud rate is stored in the structure members **c_ispeed** and **c_ospeed** as integer values. These ioctls are not supported on all architectures.

TCGETS2 **struct termios2 ***argp
TCSETS2 **const struct termios2 ***argp
TCSETSW2 **const struct termios2 ***argp
TCSETSF2 **const struct termios2 ***argp

The following four ioctls are just like **TCGETS**, **TCSETS**, **TCSETSW**, **TCSETSF**, except that they take a *struct termio ** instead of a *struct termios **.

TCGETA **struct termio ***argp
TCSETA **const struct termio ***argp
TCSETAW **const struct termio ***argp
TCSETAF **const struct termio ***argp

Locking the termios structure

The *termios* structure of a terminal can be locked. The lock is itself a *termios* structure, with nonzero bits or fields indicating a locked value.

TIOCGLCKTRMIOS

Argument: **struct termios ****argp*

Gets the locking status of the *termios* structure of the terminal.

TIOCSLCKTRMIOS

Argument: **const struct termios ****argp*

Sets the locking status of the *termios* structure of the terminal. Only a process with the **CAP_SYS_ADMIN** capability can do this.

Get and set window size

Window sizes are kept in the kernel, but not used by the kernel (except in the case of virtual consoles, where the kernel will update the window size when the size of the virtual console changes, for example, by loading a new font).

TIOCGWINSZ

Argument: **struct winsize ****argp*

Get window size.

TIOCSWINSZ

Argument: **const struct winsize ****argp*

Set window size.

The struct used by these ioctls is defined as

```
struct winsize {
    unsigned short ws_row;
    unsigned short ws_col;
    unsigned short ws_xpixel;   /* unused */
    unsigned short ws_ypixel;   /* unused */
};
```

When the window size changes, a **SIGWINCH** signal is sent to the foreground process group.

Sending a break

TCSBRK

Argument: **int** *arg*

Equivalent to *tcsendbreak(fd, arg)*.

If the terminal is using asynchronous serial data transmission, and *arg* is zero, then send a break (a stream of zero bits) for between 0.25 and 0.5 seconds. If the terminal is not using asynchronous serial data transmission, then either a break is sent, or the function returns without doing anything. When *arg* is nonzero, nobody knows what will happen.

(SVr4, UnixWare, Solaris, and Linux treat *tcsendbreak(fd,arg)* with nonzero *arg* like *tcdrain(fd)*. SunOS treats *arg* as a multiplier, and sends a stream of bits *arg* times as long as done for zero *arg*. DG/UX and AIX treat *arg* (when nonzero) as a time interval measured in milliseconds. HP-UX ignores *arg*.)

TCSBRKP

Argument: **int** *arg*

So-called "POSIX version" of **TCSBRK**. It treats nonzero *arg* as a time interval measured in deciseconds, and does nothing when the driver does not support breaks.

TIOCSBRK

Argument: **void**

Turn break on, that is, start sending zero bits.

TIOCCBRK

Argument: **void**

Turn break off, that is, stop sending zero bits.

Software flow control

TCXONC

Argument: **int** *arg*

Equivalent to *tcflow(fd, arg)*.

See *tcflow*(3) for the argument values **TCOOFF**, **TCOON**, **TCIOFF**, **TCION**.

Buffer count and flushing

FIONREAD

Argument: **int ***_argp_

Get the number of bytes in the input buffer.

TIOCINQ

Argument: **int ***_argp_

Same as **FIONREAD**.

TIOCOUTQ

Argument: **int ***_argp_

Get the number of bytes in the output buffer.

TCFLSH

Argument: **int** *arg*

Equivalent to *tcflush(fd, arg)*.

See *tcflush*(3) for the argument values **TCIFLUSH**, **TCOFLUSH**, **TCIOFLUSH**.

TIOCSERGETLSR

Argument: **int ***_argp_

Get line status register. Status register has **TIOCSER_TEMT** bit set when output buffer is empty and also hardware transmitter is physically empty.

Does not have to be supported by all serial tty drivers.

tcdrain(3) does not wait and returns immediately when **TIOCSER_TEMT** bit is set.

Faking input

TIOCSTI

Argument: **const char ***_argp_

Insert the given byte in the input queue.

Since Linux 6.2, this operation may require the **CAP_SYS_ADMIN** capability (if the *dev.tty.legacy_tiocsti* sysctl variable is set to false).

Redirecting console output

TIOCCONS

Argument: **void**

Redirect output that would have gone to */dev/console* or */dev/tty0* to the given terminal. If that was a pseudoterminal master, send it to the slave. Before Linux 2.6.10, anybody can do this as long as the output was not redirected yet; since Linux 2.6.10, only a process with the **CAP_SYS_ADMIN** capability may do this. If output was redirected already, then **EBUSY** is returned, but redirection can be stopped by using this ioctl with *fd* pointing at */dev/console* or */dev/tty0*.

Controlling terminal

TIOCSCTTY

Argument: **int** *arg*

Make the given terminal the controlling terminal of the calling process. The calling process must be a session leader and not have a controlling terminal already. For this case, *arg* should be specified as zero.

If this terminal is already the controlling terminal of a different session group, then the ioctl fails with **EPERM**, unless the caller has the **CAP_SYS_ADMIN** capability and *arg* equals 1,

in which case the terminal is stolen, and all processes that had it as controlling terminal lose it.

TIOCNOTTY

Argument: **void**

If the given terminal was the controlling terminal of the calling process, give up this controlling terminal. If the process was session leader, then send **SIGHUP** and **SIGCONT** to the foreground process group and all processes in the current session lose their controlling terminal.

Process group and session ID

TIOCGPGRP

Argument: **pid_t ****argp*

When successful, equivalent to **argp = tcgetpgrp(fd)*.

Get the process group ID of the foreground process group on this terminal.

TIOCSPGRP

Argument: **const pid_t ****argp*

Equivalent to *tcsetpgrp(fd, *argp)*.

Set the foreground process group ID of this terminal.

TIOCGSID

Argument: **pid_t ****argp*

When successful, equivalent to **argp = tcgetsid(fd)*.

Get the session ID of the given terminal. This fails with the error **ENOTTY** if the terminal is not a master pseudoterminal and not our controlling terminal. Strange.

Exclusive mode

TIOCEXCL

Argument: **void**

Put the terminal into exclusive mode. No further *open*(2) operations on the terminal are permitted. (They fail with **EBUSY**, except for a process with the **CAP_SYS_ADMIN** capability.)

TIOCGEXCL

Argument: **int ****argp*

(since Linux 3.8) If the terminal is currently in exclusive mode, place a nonzero value in the location pointed to by *argp*; otherwise, place zero in **argp*.

TIOCNXCL

Argument: **void**

Disable exclusive mode.

Line discipline

TIOCGETD

Argument: **int ****argp*

Get the line discipline of the terminal.

TIOCSETD

Argument: **const int ****argp*

Set the line discipline of the terminal.

Pseudoterminal ioctls

TIOCPKT

Argument: **const int ****argp*

Enable (when **argp* is nonzero) or disable packet mode. Can be applied to the master side of a pseudoterminal only (and will return **ENOTTY** otherwise). In packet mode, each subsequent *read*(2) will return a packet that either contains a single nonzero control byte, or has a single byte containing zero ('\0') followed by data written on the slave side of the pseudoterminal. If the first byte is not **TIOCPKT_DATA** (0), it is an OR of one or more of the following

bits:

TIOCPKT_FLUSHREAD	The read queue for the terminal is flushed.
TIOCPKT_FLUSHWRITE	The write queue for the terminal is flushed.
TIOCPKT_STOP	Output to the terminal is stopped.
TIOCPKT_START	Output to the terminal is restarted.
TIOCPKT_DOSTOP	The start and stop characters are **^S**/**^Q**.
TIOCPKT_NOSTOP	The start and stop characters are not **^S**/**^Q**.

While packet mode is in use, the presence of control status information to be read from the master side may be detected by a *select*(2) for exceptional conditions or a *poll*(2) for the **POLLPRI** event.

This mode is used by *rlogin*(1) and *rlogind*(8) to implement a remote-echoed, locally **^S**/**^Q** flow-controlled remote login.

TIOCGPKT

Argument: **const int ***_argp_

(since Linux 3.8) Return the current packet mode setting in the integer pointed to by *argp*.

TIOCSPTLCK

Argument: **int ***_argp_

Set (if **argp* is nonzero) or remove (if **argp* is zero) the lock on the pseudoterminal slave device. (See also *unlockpt*(3).)

TIOCGPTLCK

Argument: **int ***_argp_

(since Linux 3.8) Place the current lock state of the pseudoterminal slave device in the location pointed to by *argp*.

TIOCGPTPEER

Argument: **int** *flags*

(since Linux 4.13) Given a file descriptor in *fd* that refers to a pseudoterminal master, open (with the given *open*(2)-style *flags*) and return a new file descriptor that refers to the peer pseudoterminal slave device. This operation can be performed regardless of whether the pathname of the slave device is accessible through the calling process's mount namespace.

Security-conscious programs interacting with namespaces may wish to use this operation rather than *open*(2) with the pathname returned by *ptsname*(3), and similar library functions that have insecure APIs. (For example, confusion can occur in some cases using *ptsname*(3) with a pathname where a devpts filesystem has been mounted in a different mount namespace.)

The BSD ioctls **TIOCSTOP**, **TIOCSTART**, **TIOCUCNTL**, and **TIOCREMOTE** have not been implemented under Linux.

Modem control

TIOCMGET

Argument: **int ***_argp_

Get the status of modem bits.

TIOCMSET

Argument: **const int ***_argp_

Set the status of modem bits.

TIOCMBIC

Argument: **const int ***_argp_

Clear the indicated modem bits.

TIOCMBIS
Argument: **const int ****argp*

Set the indicated modem bits.

The following bits are used by the above ioctls:

TIOCM_LE	DSR (data set ready/line enable)
TIOCM_DTR	DTR (data terminal ready)
TIOCM_RTS	RTS (request to send)
TIOCM_ST	Secondary TXD (transmit)
TIOCM_SR	Secondary RXD (receive)
TIOCM_CTS	CTS (clear to send)
TIOCM_CAR	DCD (data carrier detect)
TIOCM_CD	see TIOCM_CAR
TIOCM_RNG	RNG (ring)
TIOCM_RI	see TIOCM_RNG
TIOCM_DSR	DSR (data set ready)

TIOCMIWAIT
Argument: **int** *arg*

Wait for any of the 4 modem bits (DCD, RI, DSR, CTS) to change. The bits of interest are specified as a bit mask in *arg*, by ORing together any of the bit values, **TIOCM_RNG**, **TIOCM_DSR**, **TIOCM_CD**, and **TIOCM_CTS**. The caller should use **TIOCGICOUNT** to see which bit has changed.

TIOCGICOUNT
Argument: **struct serial_icounter_struct ****argp*

Get counts of input serial line interrupts (DCD, RI, DSR, CTS). The counts are written to the *serial_icounter_struct* structure pointed to by *argp*.

Note: both 1->0 and 0->1 transitions are counted, except for RI, where only 0->1 transitions are counted.

Marking a line as local

TIOCGSOFTCAR
Argument: **int ****argp*

("Get software carrier flag") Get the status of the CLOCAL flag in the c_cflag field of the *termios* structure.

TIOCSSOFTCAR
Argument: **const int ****argp*

("Set software carrier flag") Set the CLOCAL flag in the *termios* structure when **argp* is nonzero, and clear it otherwise.

If the **CLOCAL** flag for a line is off, the hardware carrier detect (DCD) signal is significant, and an *open*(2) of the corresponding terminal will block until DCD is asserted, unless the **O_NONBLOCK** flag is given. If **CLOCAL** is set, the line behaves as if DCD is always asserted. The software carrier flag is usually turned on for local devices, and is off for lines with modems.

Linux-specific

For the **TIOCLINUX** ioctl, see *ioctl_console*(2).

Kernel debugging

#include <linux/tty.h>

TIOCTTYGSTRUCT
Argument: **struct tty_struct ****argp*

Get the *tty_struct* corresponding to *fd*. This operation was removed in Linux 2.5.67.

RETURN VALUE

The *ioctl*(2) system call returns 0 on success. On error, it returns −1 and sets *errno* to indicate the error.

ERRORS

EINVAL
Invalid operation parameter.

ENOIOCTLCMD
Unknown operation.

ENOTTY
Inappropriate *fd*.

EPERM
Insufficient permission.

EXAMPLES

Check the condition of DTR on the serial port.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

int
main(void)
{
    int fd, serial;

    fd = open("/dev/ttyS0", O_RDONLY);
    ioctl(fd, TIOCMGET, &serial);
    if (serial & TIOCM_DTR)
        puts("TIOCM_DTR is set");
    else
        puts("TIOCM_DTR is not set");
    close(fd);
}
```

Get or set arbitrary baudrate on the serial port.

```
/* SPDX-License-Identifier: GPL-2.0-or-later */

#include <asm/termbits.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
#if !defined BOTHER
    fprintf(stderr, "BOTHER is unsupported\n");
    /* Program may fallback to TCGETS/TCSETS with Bnnn constants */
    exit(EXIT_FAILURE);
#else
    /* Declare tio structure, its type depends on supported ioctl */
# if defined TCGETS2
    struct termios2 tio;
# else
    struct termios tio;
# endif
    int fd, rc;

    if (argc != 2 && argc != 3 && argc != 4) {
```

```
            fprintf(stderr, "Usage: %s device [output [input] ]\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        fd = open(argv[1], O_RDWR | O_NONBLOCK | O_NOCTTY);
        if (fd < 0) {
            perror("open");
            exit(EXIT_FAILURE);
        }

        /* Get the current serial port settings via supported ioctl */
    # if defined TCGETS2
        rc = ioctl(fd, TCGETS2, &tio);
    # else
        rc = ioctl(fd, TCGETS, &tio);
    # endif
        if (rc) {
            perror("TCGETS");
            close(fd);
            exit(EXIT_FAILURE);
        }

        /* Change baud rate when more arguments were provided */
        if (argc == 3 || argc == 4) {
            /* Clear the current output baud rate and fill a new value */
            tio.c_cflag &= ~CBAUD;
            tio.c_cflag |= BOTHER;
            tio.c_ospeed = atoi(argv[2]);

            /* Clear the current input baud rate and fill a new value */
            tio.c_cflag &= ~(CBAUD << IBSHIFT);
            tio.c_cflag |= BOTHER << IBSHIFT;
            /* When 4th argument is not provided reuse output baud rate */
            tio.c_ispeed = (argc == 4) ? atoi(argv[3]) : atoi(argv[2]);

            /* Set new serial port settings via supported ioctl */
    # if defined TCSETS2
            rc = ioctl(fd, TCSETS2, &tio);
    # else
            rc = ioctl(fd, TCSETS, &tio);
    # endif
            if (rc) {
                perror("TCSETS");
                close(fd);
                exit(EXIT_FAILURE);
            }

            /* And get new values which were really configured */
    # if defined TCGETS2
            rc = ioctl(fd, TCGETS2, &tio);
    # else
            rc = ioctl(fd, TCGETS, &tio);
    # endif
            if (rc) {
                perror("TCGETS");
                close(fd);
                exit(EXIT_FAILURE);
            }
        }
```

```
        close(fd);

        printf("output baud rate: %u\n", tio.c_ospeed);
        printf("input baud rate: %u\n", tio.c_ispeed);

        exit(EXIT_SUCCESS);
    #endif
    }
```

SEE ALSO

ldattach(8), *ioctl*(2), *ioctl_console*(2), *termios*(3), *pty*(7)

NAME

ioctl_userfaultfd − create a file descriptor for handling page faults in user space

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/userfaultfd.h> /* Definition of **UFFD*** constants */
#include <sys/ioctl.h>

int ioctl(int *fd***, int** *op***, ...);**

DESCRIPTION

Various *ioctl*(2) operations can be performed on a userfaultfd object (created by a call to *userfaultfd*(2)) using calls of the form:

```
ioctl(fd, op, argp);
```

In the above, *fd* is a file descriptor referring to a userfaultfd object, *op* is one of the operations listed below, and *argp* is a pointer to a data structure that is specific to *op*.

The various *ioctl*(2) operations are described below. The **UFFDIO_API**, **UFFDIO_REGISTER**, and **UFFDIO_UNREGISTER** operations are used to *configure* userfaultfd behavior. These operations allow the caller to choose what features will be enabled and what kinds of events will be delivered to the application. The remaining operations are *range* operations. These operations enable the calling application to resolve page-fault events.

UFFDIO_API

(Since Linux 4.3.) Enable operation of the userfaultfd and perform API handshake.

The *argp* argument is a pointer to a *uffdio_api* structure, defined as:

```
struct uffdio_api {
    __u64 api;        /* Requested API version (input) */
    __u64 features;   /* Requested features (input/output) */
    __u64 ioctls;     /* Available ioctl() operations (output) */
};
```

The *api* field denotes the API version requested by the application. The kernel verifies that it can support the requested API version, and sets the *features* and *ioctls* fields to bit masks representing all the available features and the generic *ioctl*(2) operations available.

Since Linux 4.11, applications should use the *features* field to perform a two-step handshake. First, **UFFDIO_API** is called with the *features* field set to zero. The kernel responds by setting all supported feature bits.

Applications which do not require any specific features can begin using the userfaultfd immediately. Applications which do need specific features should call **UFFDIO_API** again with a subset of the reported feature bits set to enable those features.

Before Linux 4.11, the *features* field must be initialized to zero before the call to **UFFDIO_API**, and zero (i.e., no feature bits) is placed in the *features* field by the kernel upon return from *ioctl*(2).

If the application sets unsupported feature bits, the kernel will zero out the returned *uffdio_api* structure and return **EINVAL**.

The following feature bits may be set:

UFFD_FEATURE_EVENT_FORK (since Linux 4.11)
: When this feature is enabled, the userfaultfd objects associated with a parent process are duplicated into the child process during *fork*(2) and a **UFFD_EVENT_FORK** event is delivered to the userfaultfd monitor

UFFD_FEATURE_EVENT_REMAP (since Linux 4.11)
: If this feature is enabled, when the faulting process invokes *mremap*(2), the userfaultfd monitor will receive an event of type **UFFD_EVENT_REMAP**.

UFFD_FEATURE_EVENT_REMOVE (since Linux 4.11)
: If this feature is enabled, when the faulting process calls *madvise*(2) with the **MADV_DONTNEED** or **MADV_REMOVE** advice value to free a virtual memory area the userfaultfd

monitor will receive an event of type **UFFD_EVENT_REMOVE**.

UFFD_FEATURE_EVENT_UNMAP (since Linux 4.11)
If this feature is enabled, when the faulting process unmaps virtual memory either explicitly with *munmap*(2), or implicitly during either *mmap*(2) or *mremap*(2), the userfaultfd monitor will receive an event of type **UFFD_EVENT_UNMAP**.

UFFD_FEATURE_MISSING_HUGETLBFS (since Linux 4.11)
If this feature bit is set, the kernel supports registering userfaultfd ranges on hugetlbfs virtual memory areas

UFFD_FEATURE_MISSING_SHMEM (since Linux 4.11)
If this feature bit is set, the kernel supports registering userfaultfd ranges on shared memory areas. This includes all kernel shared memory APIs: System V shared memory, *tmpfs*(5), shared mappings of */dev/zero*, *mmap*(2) with the **MAP_SHARED** flag set, *memfd_create*(2), and so on.

UFFD_FEATURE_SIGBUS (since Linux 4.14)
If this feature bit is set, no page-fault events (**UFFD_EVENT_PAGEFAULT**) will be delivered. Instead, a **SIGBUS** signal will be sent to the faulting process. Applications using this feature will not require the use of a userfaultfd monitor for processing memory accesses to the regions registered with userfaultfd.

UFFD_FEATURE_THREAD_ID (since Linux 4.14)
If this feature bit is set, *uffd_msg.pagefault.feat.ptid* will be set to the faulted thread ID for each page-fault message.

UFFD_FEATURE_PAGEFAULT_FLAG_WP (since Linux 5.10)
If this feature bit is set, userfaultfd supports write-protect faults for anonymous memory. (Note that shmem / hugetlbfs support is indicated by a separate feature.)

UFFD_FEATURE_MINOR_HUGETLBFS (since Linux 5.13)
If this feature bit is set, the kernel supports registering userfaultfd ranges in minor mode on hugetlbfs-backed memory areas.

UFFD_FEATURE_MINOR_SHMEM (since Linux 5.14)
If this feature bit is set, the kernel supports registering userfaultfd ranges in minor mode on shmem-backed memory areas.

UFFD_FEATURE_EXACT_ADDRESS (since Linux 5.18)
If this feature bit is set, *uffd_msg.pagefault.address* will be set to the exact page-fault address that was reported by the hardware, and will not mask the offset within the page. Note that old Linux versions might indicate the exact address as well, even though the feature bit is not set.

UFFD_FEATURE_WP_HUGETLBFS_SHMEM (since Linux 5.19)
If this feature bit is set, userfaultfd supports write-protect faults for hugetlbfs and shmem / tmpfs memory.

UFFD_FEATURE_WP_UNPOPULATED (since Linux 6.4)
If this feature bit is set, the kernel will handle anonymous memory the same way as file memory, by allowing the user to write-protect unpopulated page table entries.

UFFD_FEATURE_POISON (since Linux 6.6)
If this feature bit is set, the kernel supports resolving faults with the **UFFDIO_POISON** ioctl.

UFFD_FEATURE_WP_ASYNC (since Linux 6.7)
If this feature bit is set, the write protection faults would be asynchronously resolved by the kernel.

The returned *ioctls* field can contain the following bits:

1 << _UFFDIO_API
The **UFFDIO_API** operation is supported.

1 << _UFFDIO_REGISTER
The **UFFDIO_REGISTER** operation is supported.

1 << _UFFDIO_UNREGISTER
The **UFFDIO_UNREGISTER** operation is supported.

This *ioctl*(2) operation returns 0 on success. On error, −1 is returned and *errno* is set to indicate the error. If an error occurs, the kernel may zero the provided *uffdio_api* structure. The caller should treat its contents as unspecified, and reinitialize it before re-attempting another **UFFDIO_API** call. Possible errors include:

EFAULT
argp refers to an address that is outside the calling process's accessible address space.

EINVAL
The API version requested in the *api* field is not supported by this kernel, or the *features* field passed to the kernel includes feature bits that are not supported by the current kernel version.

EINVAL
A previous **UFFDIO_API** call already enabled one or more features for this userfaultfd. Calling **UFFDIO_API** twice, the first time with no features set, is explicitly allowed as per the two-step feature detection handshake.

EPERM
The **UFFD_FEATURE_EVENT_FORK** feature was enabled, but the calling process doesn't have the **CAP_SYS_PTRACE** capability.

UFFDIO_REGISTER
(Since Linux 4.3.) Register a memory address range with the userfaultfd object. The pages in the range must be "compatible". Please refer to the list of register modes below for the compatible memory backends for each mode.

The *argp* argument is a pointer to a *uffdio_register* structure, defined as:

```
struct uffdio_range {
    __u64 start;    /* Start of range */
    __u64 len;      /* Length of range (bytes) */
};

struct uffdio_register {
    struct uffdio_range range;
    __u64 mode;     /* Desired mode of operation (input) */
    __u64 ioctls;   /* Available ioctl() operations (output) */
};
```

The *range* field defines a memory range starting at *start* and continuing for *len* bytes that should be handled by the userfaultfd.

The *mode* field defines the mode of operation desired for this memory region. The following values may be bitwise ORed to set the userfaultfd mode for the specified range:

UFFDIO_REGISTER_MODE_MISSING
Track page faults on missing pages. Since Linux 4.3, only private anonymous ranges are compatible. Since Linux 4.11, hugetlbfs and shared memory ranges are also compatible.

UFFDIO_REGISTER_MODE_WP
Track page faults on write-protected pages. Since Linux 5.7, only private anonymous ranges are compatible.

UFFDIO_REGISTER_MODE_MINOR
Track minor page faults. Since Linux 5.13, only hugetlbfs ranges are compatible. Since Linux 5.14, compatibility with shmem ranges was added.

If the operation is successful, the kernel modifies the *ioctls* bit-mask field to indicate which *ioctl*(2) operations are available for the specified range. This returned bit mask can contain the following bits:

1 << _UFFDIO_COPY
The **UFFDIO_COPY** operation is supported.

1 << _UFFDIO_WAKE
The **UFFDIO_WAKE** operation is supported.

1 << _UFFDIO_WRITEPROTECT
The **UFFDIO_WRITEPROTECT** operation is supported.

1 << _UFFDIO_ZEROPAGE
The **UFFDIO_ZEROPAGE** operation is supported.

1 << _UFFDIO_CONTINUE
The **UFFDIO_CONTINUE** operation is supported.

1 << _UFFDIO_POISON
The **UFFDIO_POISON** operation is supported.

This *ioctl*(2) operation returns 0 on success. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EBUSY
A mapping in the specified range is registered with another userfaultfd object.

EFAULT
argp refers to an address that is outside the calling process's accessible address space.

EINVAL
An invalid or unsupported bit was specified in the *mode* field; or the *mode* field was zero.

EINVAL
There is no mapping in the specified address range.

EINVAL
range.start or *range.len* is not a multiple of the system page size; or, *range.len* is zero; or these fields are otherwise invalid.

EINVAL
There as an incompatible mapping in the specified address range.

UFFDIO_UNREGISTER
(Since Linux 4.3.) Unregister a memory address range from userfaultfd. The pages in the range must be "compatible" (see the description of **UFFDIO_REGISTER**.)

The address range to unregister is specified in the *uffdio_range* structure pointed to by *argp*.

This *ioctl*(2) operation returns 0 on success. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EINVAL
Either the *start* or the *len* field of the *ufdio_range* structure was not a multiple of the system page size; or the *len* field was zero; or these fields were otherwise invalid.

EINVAL
There as an incompatible mapping in the specified address range.

EINVAL
There was no mapping in the specified address range.

UFFDIO_COPY
(Since Linux 4.3.) Atomically copy a continuous memory chunk into the userfault registered range and optionally wake up the blocked thread. The source and destination addresses and the number of bytes to copy are specified by the *src*, *dst*, and *len* fields of the *uffdio_copy* structure pointed to by *argp*:

```
struct uffdio_copy {
    __u64 dst;    /* Destination of copy */
    __u64 src;    /* Source of copy */
    __u64 len;    /* Number of bytes to copy */
    __u64 mode;   /* Flags controlling behavior of copy */
    __s64 copy;   /* Number of bytes copied, or negated error */
};
```

The following value may be bitwise ORed in *mode* to change the behavior of the **UFFDIO_COPY** operation:

UFFDIO_COPY_MODE_DONTWAKE
Do not wake up the thread that waits for page-fault resolution

UFFDIO_COPY_MODE_WP
Copy the page with read-only permission. This allows the user to trap the next write to the page, which will block and generate another write-protect userfault message. This is used only when both **UFFDIO_REGISTER_MODE_MISSING** and **UFFDIO_REGISTER_MODE_WP** modes are enabled for the registered range.

The *copy* field is used by the kernel to return the number of bytes that was actually copied, or an error (a negated *errno*-style value). If the value returned in *copy* doesn't match the value that was specified in *len*, the operation fails with the error **EAGAIN**. The *copy* field is output-only; it is not read by the **UFFDIO_COPY** operation.

This *ioctl*(2) operation returns 0 on success. In this case, the entire area was copied. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EAGAIN
The number of bytes copied (i.e., the value returned in the *copy* field) does not equal the value that was specified in the *len* field.

EINVAL
Either *dst* or *len* was not a multiple of the system page size, or the range specified by *src* and *len* or *dst* and *len* was invalid.

EINVAL
An invalid bit was specified in the *mode* field.

ENOENT (since Linux 4.11)
The faulting process has changed its virtual memory layout simultaneously with an outstanding **UFFDIO_COPY** operation.

ENOSPC (from Linux 4.11 until Linux 4.13)
The faulting process has exited at the time of a **UFFDIO_COPY** operation.

ESRCH (since Linux 4.13)
The faulting process has exited at the time of a **UFFDIO_COPY** operation.

UFFDIO_ZEROPAGE
(Since Linux 4.3.) Zero out a memory range registered with userfaultfd.

The requested range is specified by the *range* field of the *uffdio_zeropage* structure pointed to by *argp*:

```
struct uffdio_zeropage {
    struct uffdio_range range;
    __u64 mode;     /* Flags controlling behavior of copy */
    __s64 zeropage; /* Number of bytes zeroed, or negated error */
};
```

The following value may be bitwise ORed in *mode* to change the behavior of the **UFFDIO_ZEROPAGE** operation:

UFFDIO_ZEROPAGE_MODE_DONTWAKE
Do not wake up the thread that waits for page-fault resolution.

The *zeropage* field is used by the kernel to return the number of bytes that was actually zeroed, or an error in the same manner as **UFFDIO_COPY**. If the value returned in the *zeropage* field doesn't match the value that was specified in *range.len*, the operation fails with the error **EAGAIN**. The *zeropage* field is output-only; it is not read by the **UFFDIO_ZEROPAGE** operation.

This *ioctl*(2) operation returns 0 on success. In this case, the entire area was zeroed. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EAGAIN
The number of bytes zeroed (i.e., the value returned in the *zeropage* field) does not equal the value that was specified in the *range.len* field.

EINVAL
Either *range.start* or *range.len* was not a multiple of the system page size; or *range.len* was zero; or the range specified was invalid.

EINVAL
An invalid bit was specified in the *mode* field.

ESRCH (since Linux 4.13)
The faulting process has exited at the time of a **UFFDIO_ZEROPAGE** operation.

UFFDIO_WAKE
(Since Linux 4.3.) Wake up the thread waiting for page-fault resolution on a specified memory address range.

The **UFFDIO_WAKE** operation is used in conjunction with **UFFDIO_COPY** and **UFFDIO_ZEROPAGE** operations that have the **UFFDIO_COPY_MODE_DONTWAKE** or **UFFDIO_ZEROPAGE_MODE_DONTWAKE** bit set in the *mode* field. The userfault monitor can perform several **UFFDIO_COPY** and **UFFDIO_ZEROPAGE** operations in a batch and then explicitly wake up the faulting thread using **UFFDIO_WAKE**.

The *argp* argument is a pointer to a *uffdio_range* structure (shown above) that specifies the address range.

This *ioctl*(2) operation returns 0 on success. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EINVAL
The *start* or the *len* field of the *ufdio_range* structure was not a multiple of the system page size; or *len* was zero; or the specified range was otherwise invalid.

UFFDIO_WRITEPROTECT
(Since Linux 5.7.) Write-protect or write-unprotect a userfaultfd-registered memory range registered with mode **UFFDIO_REGISTER_MODE_WP**.

The *argp* argument is a pointer to a *uffdio_range* structure as shown below:

```
struct uffdio_writeprotect {
    struct uffdio_range range; /* Range to change write permission*/
    __u64 mode;                /* Mode to change write permission */
};
```

There are two mode bits that are supported in this structure:

UFFDIO_WRITEPROTECT_MODE_WP
When this mode bit is set, the ioctl will be a write-protect operation upon the memory range specified by *range*. Otherwise it will be a write-unprotect operation upon the specified range, which can be used to resolve a userfaultfd write-protect page fault.

UFFDIO_WRITEPROTECT_MODE_DONTWAKE
When this mode bit is set, do not wake up any thread that waits for page-fault resolution after the operation. This can be specified only if **UFFDIO_WRITEPROTECT_MODE_WP** is not specified.

This *ioctl*(2) operation returns 0 on success. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EINVAL
The *start* or the *len* field of the *ufdio_range* structure was not a multiple of the system page size; or *len* was zero; or the specified range was otherwise invalid.

EAGAIN
The process was interrupted; retry this call.

ENOENT
The range specified in *range* is not valid. For example, the virtual address does not exist, or not registered with userfaultfd write-protect mode.

EFAULT
Encountered a generic fault during processing.

UFFDIO_CONTINUE
(Since Linux 5.13.) Resolve a minor page fault by installing page table entries for existing pages in the page cache.

The *argp* argument is a pointer to a *uffdio_continue* structure as shown below:

```
struct uffdio_continue {
    struct uffdio_range range;
                  /* Range to install PTEs for and continue */
    __u64 mode;   /* Flags controlling the behavior of continue */
    __s64 mapped; /* Number of bytes mapped, or negated error */
};
```

The following value may be bitwise ORed in *mode* to change the behavior of the **UFFDIO_CONTINUE** operation:

UFFDIO_CONTINUE_MODE_DONTWAKE
Do not wake up the thread that waits for page-fault resolution.

The *mapped* field is used by the kernel to return the number of bytes that were actually mapped, or an error in the same manner as **UFFDIO_COPY**. If the value returned in the *mapped* field doesn't match the value that was specified in *range.len*, the operation fails with the error **EAGAIN**. The *mapped* field is output-only; it is not read by the **UFFDIO_CONTINUE** operation.

This *ioctl*(2) operation returns 0 on success. In this case, the entire area was mapped. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EAGAIN
The number of bytes mapped (i.e., the value returned in the *mapped* field) does not equal the value that was specified in the *range.len* field.

EEXIST
One or more pages were already mapped in the given range.

EFAULT
No existing page could be found in the page cache for the given range.

EINVAL
Either *range.start* or *range.len* was not a multiple of the system page size; or *range.len* was zero; or the range specified was invalid.

EINVAL
An invalid bit was specified in the *mode* field.

ENOENT
The faulting process has changed its virtual memory layout simultaneously with an outstanding **UFFDIO_CONTINUE** operation.

ENOMEM
Allocating memory needed to setup the page table mappings failed.

ESRCH
The faulting process has exited at the time of a **UFFDIO_CONTINUE** operation.

UFFDIO_POISON
(Since Linux 6.6.) Mark an address range as "poisoned". Future accesses to these addresses will raise a **SIGBUS** signal. Unlike **MADV_HWPOISON** this works by installing page table entries, rather than "really" poisoning the underlying physical pages. This means it only affects this particular address space.

The *argp* argument is a pointer to a *uffdio_poison* structure as shown below:

```
struct uffdio_poison {
      struct uffdio_range range;
                    /* Range to install poison PTE markers in */
      __u64 mode;   /* Flags controlling the behavior of poison */
```

```
        __s64 updated;  /* Number of bytes poisoned, or negated error */
};
```

The following value may be bitwise ORed in *mode* to change the behavior of the **UFFDIO_POISON** operation:

UFFDIO_POISON_MODE_DONTWAKE
Do not wake up the thread that waits for page-fault resolution.

The *updated* field is used by the kernel to return the number of bytes that were actually poisoned, or an error in the same manner as **UFFDIO_COPY**. If the value returned in the *updated* field doesn't match the value that was specified in *range.len*, the operation fails with the error **EAGAIN**. The *updated* field is output-only; it is not read by the **UFFDIO_POISON** operation.

This *ioctl*(2) operation returns 0 on success. In this case, the entire area was poisoned. On error, −1 is returned and *errno* is set to indicate the error. Possible errors include:

EAGAIN
The number of bytes mapped (i.e., the value returned in the *updated* field) does not equal the value that was specified in the *range.len* field.

EINVAL
Either *range.start* or *range.len* was not a multiple of the system page size; or *range.len* was zero; or the range specified was invalid.

EINVAL
An invalid bit was specified in the *mode* field.

EEXIST
One or more pages were already mapped in the given range.

ENOENT
The faulting process has changed its virtual memory layout simultaneously with an outstanding **UFFDIO_POISON** operation.

ENOMEM
Allocating memory for page table entries failed.

ESRCH
The faulting process has exited at the time of a **UFFDIO_POISON** operation.

RETURN VALUE

See descriptions of the individual operations, above.

ERRORS

See descriptions of the individual operations, above. In addition, the following general errors can occur for all of the operations described above:

EFAULT
argp does not point to a valid memory address.

EINVAL
(For all operations except **UFFDIO_API**.) The userfaultfd object has not yet been enabled (via the **UFFDIO_API** operation).

STANDARDS

Linux.

BUGS

In order to detect available userfault features and enable some subset of those features the userfaultfd file descriptor must be closed after the first **UFFDIO_API** operation that queries features availability and reopened before the second **UFFDIO_API** operation that actually enables the desired features.

EXAMPLES

See *userfaultfd*(2).

SEE ALSO

ioctl(2), *mmap*(2), *userfaultfd*(2)

Documentation/admin−guide/mm/userfaultfd.rst in the Linux kernel source tree

NAME

ioperm − set port input/output permissions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/io.h>

int ioperm(unsigned long *from***, unsigned long** *num***, int** *turn_on***);**

DESCRIPTION

ioperm() sets the port access permission bits for the calling thread for *num* bits starting from port address *from*. If *turn_on* is nonzero, then permission for the specified bits is enabled; otherwise it is disabled. If *turn_on* is nonzero, the calling thread must be privileged (**CAP_SYS_RAWIO**).

Before Linux 2.6.8, only the first 0x3ff I/O ports could be specified in this manner. For more ports, the *iopl*(2) system call had to be used (with a *level* argument of 3). Since Linux 2.6.8, 65,536 I/O ports can be specified.

Permissions are inherited by the child created by *fork*(2) (but see NOTES). Permissions are preserved across *execve*(2); this is useful for giving port access permissions to unprivileged programs.

This call is mostly for the i386 architecture. On many other architectures it does not exist or will always return an error.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
Invalid values for *from* or *num*.

EIO (on PowerPC) This call is not supported.

ENOMEM
Out of memory.

EPERM
The calling thread has insufficient privilege.

VERSIONS

glibc has an **ioperm**() prototype both in <*sys/io.h*> and in <*sys/perm.h*>. Avoid the latter, it is available on i386 only.

STANDARDS

Linux.

HISTORY

Before Linux 2.4, permissions were not inherited by a child created by *fork*(2).

NOTES

The */proc/ioports* file shows the I/O ports that are currently allocated on the system.

SEE ALSO

iopl(2), *outb*(2), *capabilities*(7)

NAME

iopl − change I/O privilege level

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/io.h>

[[deprecated]] int iopl(int *level***);**

DESCRIPTION

iopl() changes the I/O privilege level of the calling thread, as specified by the two least significant bits in *level*.

The I/O privilege level for a normal thread is 0. Permissions are inherited from parents to children.

This call is deprecated, is significantly slower than *ioperm*(2), and is only provided for older X servers which require access to all 65536 I/O ports. It is mostly for the i386 architecture. On many other architectures it does not exist or will always return an error.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: *level* is greater than 3.

ENOSYS
: This call is unimplemented.

EPERM
: The calling thread has insufficient privilege to call **iopl**(); the **CAP_SYS_RAWIO** capability is required to raise the I/O privilege level above its current value.

VERSIONS

glibc2 has a prototype both in <*sys/io.h*> and in <*sys/perm.h*>. Avoid the latter, it is available on i386 only.

STANDARDS

Linux.

HISTORY

Prior to Linux 5.5 **iopl**() allowed the thread to disable interrupts while running at a higher I/O privilege level. This will probably crash the system, and is not recommended.

Prior to Linux 3.7, on some architectures (such as i386), permissions *were* inherited by the child produced by *fork*(2) and were preserved across *execve*(2). This behavior was inadvertently changed in Linux 3.7, and won't be reinstated.

SEE ALSO

ioperm(2), *outb*(2), *capabilities*(7)

NAME

ioprio_get, ioprio_set − get/set I/O scheduling class and priority

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/ioprio.h>    /* Definition of IOPRIO_* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_ioprio_get, int which, int who);
int syscall(SYS_ioprio_set, int which, int who, int ioprio);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

The **ioprio_get**() and **ioprio_set**() system calls get and set the I/O scheduling class and priority of one or more threads.

The *which* and *who* arguments identify the thread(s) on which the system calls operate. The *which* argument determines how *who* is interpreted, and has one of the following values:

IOPRIO_WHO_PROCESS
: *who* is a process ID or thread ID identifying a single process or thread. If *who* is 0, then operate on the calling thread.

IOPRIO_WHO_PGRP
: *who* is a process group ID identifying all the members of a process group. If *who* is 0, then operate on the process group of which the caller is a member.

IOPRIO_WHO_USER
: *who* is a user ID identifying all of the processes that have a matching real UID.

If *which* is specified as **IOPRIO_WHO_PGRP** or **IOPRIO_WHO_USER** when calling **ioprio_get**(), and more than one process matches *who*, then the returned priority will be the highest one found among all of the matching processes. One priority is said to be higher than another one if it belongs to a higher priority class (**IOPRIO_CLASS_RT** is the highest priority class; **IOPRIO_CLASS_IDLE** is the lowest) or if it belongs to the same priority class as the other process but has a higher priority level (a lower priority number means a higher priority level).

The *ioprio* argument given to **ioprio_set**() is a bit mask that specifies both the scheduling class and the priority to be assigned to the target process(es). The following macros are used for assembling and dissecting *ioprio* values:

IOPRIO_PRIO_VALUE(*class***,** *data***)**
: Given a scheduling *class* and priority (*data*), this macro combines the two values to produce an *ioprio* value, which is returned as the result of the macro.

IOPRIO_PRIO_CLASS(*mask***)**
: Given *mask* (an *ioprio* value), this macro returns its I/O class component, that is, one of the values **IOPRIO_CLASS_RT**, **IOPRIO_CLASS_BE**, or **IOPRIO_CLASS_IDLE**.

IOPRIO_PRIO_DATA(*mask***)**
: Given *mask* (an *ioprio* value), this macro returns its priority (*data*) component.

See the NOTES section for more information on scheduling classes and priorities, as well as the meaning of specifying *ioprio* as 0.

I/O priorities are supported for reads and for synchronous (**O_DIRECT**, **O_SYNC**) writes. I/O priorities are not supported for asynchronous writes because they are issued outside the context of the program dirtying the memory, and thus program-specific priorities do not apply.

RETURN VALUE

On success, **ioprio_get**() returns the *ioprio* value of the process with highest I/O priority of any of the processes that match the criteria specified in *which* and *who*. On error, −1 is returned, and *errno* is set to indicate the error.

On success, **ioprio_set**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
Invalid value for *which* or *ioprio*. Refer to the NOTES section for available scheduler classes and priority levels for *ioprio*.

EPERM
The calling process does not have the privilege needed to assign this *ioprio* to the specified process(es). See the NOTES section for more information on required privileges for **ioprio_set**().

ESRCH
No process(es) could be found that matched the specification in *which* and *who*.

STANDARDS

Linux.

HISTORY

Linux 2.6.13.

NOTES

Two or more processes or threads can share an I/O context. This will be the case when *clone*(2) was called with the **CLONE_IO** flag. However, by default, the distinct threads of a process will **not** share the same I/O context. This means that if you want to change the I/O priority of all threads in a process, you may need to call **ioprio_set**() on each of the threads. The thread ID that you would need for this operation is the one that is returned by *gettid*(2) or *clone*(2).

These system calls have an effect only when used in conjunction with an I/O scheduler that supports I/O priorities. As at kernel 2.6.17 the only such scheduler is the Completely Fair Queuing (CFQ) I/O scheduler.

If no I/O scheduler has been set for a thread, then by default the I/O priority will follow the CPU nice value (**setpriority**(2)). Before Linux 2.6.24, once an I/O priority had been set using **ioprio_set**(), there was no way to reset the I/O scheduling behavior to the default. Since Linux 2.6.24, specifying *ioprio* as 0 can be used to reset to the default I/O scheduling behavior.

Selecting an I/O scheduler

I/O schedulers are selected on a per-device basis via the special file */sys/block/*device*/queue/scheduler*.

One can view the current I/O scheduler via the */sys* filesystem. For example, the following command displays a list of all schedulers currently loaded in the kernel:

```
$ cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

The scheduler surrounded by brackets is the one actually in use for the device (*sda* in the example). Setting another scheduler is done by writing the name of the new scheduler to this file. For example, the following command will set the scheduler for the *sda* device to *cfq*:

```
$ su
Password:
# echo cfq > /sys/block/sda/queue/scheduler
```

The Completely Fair Queuing (CFQ) I/O scheduler

Since version 3 (also known as CFQ Time Sliced), CFQ implements I/O nice levels similar to those of CPU scheduling. These nice levels are grouped into three scheduling classes, each one containing one or more priority levels:

IOPRIO_CLASS_RT (1)
This is the real-time I/O class. This scheduling class is given higher priority than any other class: processes from this class are given first access to the disk every time. Thus, this I/O class needs to be used with some care: one I/O real-time process can starve the entire system. Within the real-time class, there are 8 levels of class data (priority) that determine exactly how much time this process needs the disk for on each service. The highest real-time priority level is 0; the lowest is 7. In the future, this might change to be more directly mappable to performance, by passing in a desired data rate instead.

IOPRIO_CLASS_BE (2)
: This is the best-effort scheduling class, which is the default for any process that hasn't set a specific I/O priority. The class data (priority) determines how much I/O bandwidth the process will get. Best-effort priority levels are analogous to CPU nice values (see *getpriority*(2)). The priority level determines a priority relative to other processes in the best-effort scheduling class. Priority levels range from 0 (highest) to 7 (lowest).

IOPRIO_CLASS_IDLE (3)
: This is the idle scheduling class. Processes running at this level get I/O time only when no one else needs the disk. The idle class has no class data. Attention is required when assigning this priority class to a process, since it may become starved if higher priority processes are constantly accessing the disk.

Refer to the kernel source file *Documentation/block/ioprio.txt* for more information on the CFQ I/O Scheduler and an example program.

Required permissions to set I/O priorities
: Permission to change a process's priority is granted or denied based on two criteria:

Process ownership
: An unprivileged process may set the I/O priority only for a process whose real UID matches the real or effective UID of the calling process. A process which has the **CAP_SYS_NICE** capability can change the priority of any process.

What is the desired priority
: Attempts to set very high priorities (**IOPRIO_CLASS_RT**) require the **CAP_SYS_ADMIN** capability. Up to Linux 2.6.24 also required **CAP_SYS_ADMIN** to set a very low priority (**IOPRIO_CLASS_IDLE**), but since Linux 2.6.25, this is no longer required.

A call to **ioprio_set**() must follow both rules, or the call will fail with the error **EPERM**.

BUGS

glibc does not yet provide a suitable header file defining the function prototypes and macros described on this page. Suitable definitions can be found in *linux/ioprio.h*.

SEE ALSO

ionice(1), *getpriority*(2), *open*(2), *capabilities*(7), *cgroups*(7)

Documentation/block/ioprio.txt in the Linux kernel source tree

NAME

ipc − System V IPC system calls

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/ipc.h>      /* Definition of needed constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_ipc, unsigned int call, int  first,
            unsigned long second, unsigned long third, void *ptr,
            long  fifth);
```

Note: glibc provides no wrapper for **ipc**(), necessitating the use of *syscall*(2).

DESCRIPTION

ipc() is a common kernel entry point for the System V IPC calls for messages, semaphores, and shared memory. *call* determines which IPC function to invoke; the other arguments are passed through to the appropriate call.

User-space programs should call the appropriate functions by their usual names. Only standard library implementors and kernel hackers need to know about **ipc**().

VERSIONS

On some architectures—for example x86-64 and ARM—there is no **ipc**() system call; instead, *msgctl*(2), *semctl*(2), *shmctl*(2), and so on really are implemented as separate system calls.

STANDARDS

Linux.

SEE ALSO

msgctl(2), *msgget*(2), *msgrcv*(2), *msgsnd*(2), *semctl*(2), *semget*(2), *semop*(2), *semtimedop*(2), *shmat*(2), *shmctl*(2), *shmdt*(2), *shmget*(2), *sysvipc*(7)

NAME

kcmp − compare two processes to determine if they share a kernel resource

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/kcmp.h>      /* Definition of KCMP_* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_kcmp, pid_t pid1, pid_t pid2, int type,
            unsigned long idx1, unsigned long idx2);
```

Note: glibc provides no wrapper for **kcmp**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **kcmp**() system call can be used to check whether the two processes identified by *pid1* and *pid2* share a kernel resource such as virtual memory, file descriptors, and so on.

Permission to employ **kcmp**() is governed by ptrace access mode **PTRACE_MODE_READ_REALCREDS** checks against both *pid1* and *pid2*; see *ptrace*(2).

The *type* argument specifies which resource is to be compared in the two processes. It has one of the following values:

KCMP_FILE
: Check whether a file descriptor *idx1* in the process *pid1* refers to the same open file description (see *open*(2)) as file descriptor *idx2* in the process *pid2*. The existence of two file descriptors that refer to the same open file description can occur as a result of *dup*(2) (and similar) *fork*(2), or passing file descriptors via a domain socket (see *unix*(7)).

KCMP_FILES
: Check whether the processes share the same set of open file descriptors. The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_FILES** flag in *clone*(2).

KCMP_FS
: Check whether the processes share the same filesystem information (i.e., file mode creation mask, working directory, and filesystem root). The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_FS** flag in *clone*(2).

KCMP_IO
: Check whether the processes share I/O context. The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_IO** flag in *clone*(2).

KCMP_SIGHAND
: Check whether the processes share the same table of signal dispositions. The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_SIGHAND** flag in *clone*(2).

KCMP_SYSVSEM
: Check whether the processes share the same list of System V semaphore undo operations. The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_SYSVSEM** flag in *clone*(2).

KCMP_VM
: Check whether the processes share the same address space. The arguments *idx1* and *idx2* are ignored. See the discussion of the **CLONE_VM** flag in *clone*(2).

KCMP_EPOLL_TFD (since Linux 4.13)
: Check whether the file descriptor *idx1* of the process *pid1* is present in the *epoll*(7) instance described by *idx2* of the process *pid2*. The argument *idx2* is a pointer to a structure where the target file is described. This structure has the form:

```
struct kcmp_epoll_slot {
    __u32 efd;
    __u32 tfd;
    __u64 toff;
};
```

Within this structure, *efd* is an epoll file descriptor returned from *epoll_create*(2), *tfd* is a target file descriptor number, and *toff* is a target file offset counted from zero. Several different targets may be registered with the same file descriptor number and setting a specific offset helps to investigate each of them.

Note the **kcmp**() is not protected against false positives which may occur if the processes are currently running. One should stop the processes by sending **SIGSTOP** (see *signal*(7)) prior to inspection with this system call to obtain meaningful results.

RETURN VALUE

The return value of a successful call to **kcmp**() is simply the result of arithmetic comparison of kernel pointers (when the kernel compares resources, it uses their memory addresses).

The easiest way to explain is to consider an example. Suppose that *v1* and *v2* are the addresses of appropriate resources, then the return value is one of the following:

0 *v1* is equal to *v2*; in other words, the two processes share the resource.

1 *v1* is less than *v2*.

2 *v1* is greater than *v2*.

3 *v1* is not equal to *v2*, but ordering information is unavailable.

On error, −1 is returned, and *errno* is set to indicate the error.

kcmp() was designed to return values suitable for sorting. This is particularly handy if one needs to compare a large number of file descriptors.

ERRORS

EBADF
type is **KCMP_FILE** and *fd1* or *fd2* is not an open file descriptor.

EFAULT
The epoll slot addressed by *idx2* is outside of the user's address space.

EINVAL
type is invalid.

ENOENT
The target file is not present in *epoll*(7) instance.

EPERM
Insufficient permission to inspect process resources. The **CAP_SYS_PTRACE** capability is required to inspect processes that you do not own. Other ptrace limitations may also apply, such as **CONFIG_SECURITY_YAMA**, which, when */proc/sys/kernel/yama/ptrace_scope* is 2, limits **kcmp**() to child processes; see *ptrace*(2).

ESRCH
Process *pid1* or *pid2* does not exist.

STANDARDS

Linux.

HISTORY

Linux 3.5.

Before Linux 5.12, this system call is available only if the kernel is configured with **CONFIG_CHECKPOINT_RESTORE**, since the original purpose of the system call was for the checkpoint/restore in user space (CRIU) feature. (The alternative to this system call would have been to expose suitable process information via the *proc*(5) filesystem; this was deemed to be unsuitable for security reasons.) Since Linux 5.12, this system call is also available if the kernel is configured with **CONFIG_KCMP**.

NOTES

See *clone*(2) for some background information on the shared resources referred to on this page.

EXAMPLES

The program below uses **kcmp**() to test whether pairs of file descriptors refer to the same open file description. The program tests different cases for the file descriptor pairs, as described in the program output. An example run of the program is as follows:

```
$ ./a.out
Parent PID is 1144
Parent opened file on FD 3

PID of child of fork() is 1145
      Compare duplicate FDs from different processes:
            kcmp(1145, 1144, KCMP_FILE, 3, 3) ==> same
Child opened file on FD 4
      Compare FDs from distinct open()s in same process:
            kcmp(1145, 1145, KCMP_FILE, 3, 4) ==> different
Child duplicated FD 3 to create FD 5
      Compare duplicated FDs in same process:
            kcmp(1145, 1145, KCMP_FILE, 3, 5) ==> same
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <fcntl.h>
#include <linux/kcmp.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <unistd.h>

static int
kcmp(pid_t pid1, pid_t pid2, int type,
     unsigned long idx1, unsigned long idx2)
{
    return syscall(SYS_kcmp, pid1, pid2, type, idx1, idx2);
}

static void
test_kcmp(char *msg, pid_t pid1, pid_t pid2, int fd_a, int fd_b)
{
    printf("\t%s\n", msg);
    printf("\t\tkcmp(%jd, %jd, KCMP_FILE, %d, %d) ==> %s\n",
           (intmax_t) pid1, (intmax_t) pid2, fd_a, fd_b,
           (kcmp(pid1, pid2, KCMP_FILE, fd_a, fd_b) == 0) ?
                    "same" : "different");
}

int
main(void)
{
    int                fd1, fd2, fd3;
    static const char  pathname[] = "/tmp/kcmp.test";

    fd1 = open(pathname, O_CREAT | O_RDWR, 0600);
    if (fd1 == -1)
        err(EXIT_FAILURE, "open");

    printf("Parent PID is %jd\n", (intmax_t) getpid());
    printf("Parent opened file on FD %d\n\n", fd1);

    switch (fork()) {
    case -1:
```

```
            err(EXIT_FAILURE, "fork");

        case 0:
            printf("PID of child of fork() is %jd\n", (intmax_t) getpid());

            test_kcmp("Compare duplicate FDs from different processes:",
                      getpid(), getppid(), fd1, fd1);

            fd2 = open(pathname, O_CREAT | O_RDWR, 0600);
            if (fd2 == -1)
                err(EXIT_FAILURE, "open");
            printf("Child opened file on FD %d\n", fd2);

            test_kcmp("Compare FDs from distinct open()s in same process:",
                      getpid(), getpid(), fd1, fd2);

            fd3 = dup(fd1);
            if (fd3 == -1)
                err(EXIT_FAILURE, "dup");
            printf("Child duplicated FD %d to create FD %d\n", fd1, fd3);

            test_kcmp("Compare duplicated FDs in same process:",
                      getpid(), getpid(), fd1, fd3);
            break;

        default:
            wait(NULL);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

clone(2), *unshare*(2)

NAME

kexec_load, kexec_file_load − load a new kernel for later execution

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/kexec.h>      /* Definition of KEXEC_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_kexec_load, unsigned long entry,
             unsigned long nr_segments, struct kexec_segment *segments,
             unsigned long flags);
long syscall(SYS_kexec_file_load, int kernel_fd, int initrd_fd,
             unsigned long cmdline_len, const char *cmdline,
             unsigned long flags);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

The **kexec_load**() system call loads a new kernel that can be executed later by *reboot*(2).

The *flags* argument is a bit mask that controls the operation of the call. The following values can be specified in *flags*:

KEXEC_ON_CRASH (since Linux 2.6.13)
: Execute the new kernel automatically on a system crash. This "crash kernel" is loaded into an area of reserved memory that is determined at boot time using the *crashkernel* kernel command-line parameter. The location of this reserved memory is exported to user space via the */proc/iomem* file, in an entry labeled "Crash kernel". A user-space application can parse this file and prepare a list of segments (see below) that specify this reserved memory as destination. If this flag is specified, the kernel checks that the target segments specified in *segments* fall within the reserved region.

KEXEC_PRESERVE_CONTEXT (since Linux 2.6.27)
: Preserve the system hardware and software states before executing the new kernel. This could be used for system suspend. This flag is available only if the kernel was configured with **CONFIG_KEXEC_JUMP**, and is effective only if *nr_segments* is greater than 0.

The high-order bits (corresponding to the mask 0xffff0000) of *flags* contain the architecture of the to-be-executed kernel. Specify (OR) the constant **KEXEC_ARCH_DEFAULT** to use the current architecture, or one of the following architecture constants **KEXEC_ARCH_386**, **KEXEC_ARCH_68K**, **KEXEC_ARCH_X86_64**, **KEXEC_ARCH_PPC**, **KEXEC_ARCH_PPC64**, **KEXEC_ARCH_IA_64**, **KEXEC_ARCH_ARM**, **KEXEC_ARCH_S390**, **KEXEC_ARCH_SH**, **KEXEC_ARCH_MIPS**, and **KEXEC_ARCH_MIPS_LE**. The architecture must be executable on the CPU of the system.

The *entry* argument is the physical entry address in the kernel image. The *nr_segments* argument is the number of segments pointed to by the *segments* pointer; the kernel imposes an (arbitrary) limit of 16 on the number of segments. The *segments* argument is an array of *kexec_segment* structures which define the kernel layout:

```
struct kexec_segment {
    void   *buf;        /* Buffer in user space */
    size_t  bufsz;      /* Buffer length in user space */
    void   *mem;        /* Physical address of kernel */
    size_t  memsz;      /* Physical address length */
};
```

The kernel image defined by *segments* is copied from the calling process into the kernel either in regular memory or in reserved memory (if **KEXEC_ON_CRASH** is set). The kernel first performs various sanity checks on the information passed in *segments*. If these checks pass, the kernel copies the segment data to kernel memory. Each segment specified in *segments* is copied as follows:

- *buf* and *bufsz* identify a memory region in the caller's virtual address space that is the source of the copy. The value in *bufsz* may not exceed the value in the *memsz* field.
- *mem* and *memsz* specify a physical address range that is the target of the copy. The values specified in both fields must be multiples of the system page size.
- *bufsz* bytes are copied from the source buffer to the target kernel buffer. If *bufsz* is less than *memsz*, then the excess bytes in the kernel buffer are zeroed out.

In case of a normal kexec (i.e., the **KEXEC_ON_CRASH** flag is not set), the segment data is loaded in any available memory and is moved to the final destination at kexec reboot time (e.g., when the *kexec*(8) command is executed with the *−e* option).

In case of kexec on panic (i.e., the **KEXEC_ON_CRASH** flag is set), the segment data is loaded to reserved memory at the time of the call, and, after a crash, the kexec mechanism simply passes control to that kernel.

The **kexec_load**() system call is available only if the kernel was configured with **CONFIG_KEXEC**.

kexec_file_load()

The **kexec_file_load**() system call is similar to **kexec_load**(), but it takes a different set of arguments. It reads the kernel to be loaded from the file referred to by the file descriptor *kernel_fd*, and the initrd (initial RAM disk) to be loaded from file referred to by the file descriptor *initrd_fd*. The *cmdline* argument is a pointer to a buffer containing the command line for the new kernel. The *cmdline_len* argument specifies size of the buffer. The last byte in the buffer must be a null byte ('\0').

The *flags* argument is a bit mask which modifies the behavior of the call. The following values can be specified in *flags*:

KEXEC_FILE_UNLOAD
: Unload the currently loaded kernel.

KEXEC_FILE_ON_CRASH
: Load the new kernel in the memory region reserved for the crash kernel (as for **KEXEC_ON_CRASH**). This kernel is booted if the currently running kernel crashes.

KEXEC_FILE_NO_INITRAMFS
: Loading initrd/initramfs is optional. Specify this flag if no initramfs is being loaded. If this flag is set, the value passed in *initrd_fd* is ignored.

The **kexec_file_load**() system call was added to provide support for systems where "kexec" loading should be restricted to only kernels that are signed. This system call is available only if the kernel was configured with **CONFIG_KEXEC_FILE**.

RETURN VALUE

On success, these system calls returns 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EADDRNOTAVAIL
: The **KEXEC_ON_CRASH** flags was specified, but the region specified by the *mem* and *memsz* fields of one of the *segments* entries lies outside the range of memory reserved for the crash kernel.

EADDRNOTAVAIL
: The value in a *mem* or *memsz* field in one of the *segments* entries is not a multiple of the system page size.

EBADF
: *kernel_fd* or *initrd_fd* is not a valid file descriptor.

EBUSY
: Another crash kernel is already being loaded or a crash kernel is already in use.

EINVAL
: *flags* is invalid.

EINVAL
: The value of a *bufsz* field in one of the *segments* entries exceeds the value in the corresponding *memsz* field.

EINVAL
nr_segments exceeds **KEXEC_SEGMENT_MAX** (16).

EINVAL
Two or more of the kernel target buffers overlap.

EINVAL
The value in *cmdline[cmdline_len−1]* is not '\0'.

EINVAL
The file referred to by *kernel_fd* or *initrd_fd* is empty (length zero).

ENOEXEC
kernel_fd does not refer to an open file, or the kernel can't load this file. Currently, the file must be a bzImage and contain an x86 kernel that is loadable above 4 GiB in memory (see the kernel source file *Documentation/x86/boot.txt*).

ENOMEM
Could not allocate memory.

EPERM
The caller does not have the **CAP_SYS_BOOT** capability.

STANDARDS
Linux.

HISTORY
kexec_load()
Linux 2.6.13.

kexec_file_load()
Linux 3.17.

SEE ALSO
reboot(2), *syscall*(2), *kexec*(8)

The kernel source files *Documentation/kdump/kdump.txt* and *Documentation/admin−guide/kernel−parameters.txt*

NAME

keyctl − manipulate the kernel's key management facility

LIBRARY

Standard C library (*libc*, *−lc*)

Alternatively, Linux Key Management Utilities (*libkeyutils*, *−lkeyutils*); see VERSIONS.

SYNOPSIS

```
#include <linux/keyctl.h>     /* Definition of KEY* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_keyctl, int operation, unsigned long arg2,
             unsigned long arg3, unsigned long arg4,
             unsigned long arg5);
```

Note: glibc provides no wrapper for **keyctl**(), necessitating the use of *syscall*(2).

DESCRIPTION

keyctl() allows user-space programs to perform key manipulation.

The operation performed by **keyctl**() is determined by the value of the *operation* argument. Each of these operations is wrapped by the *libkeyutils* library (provided by the *keyutils* package) into individual functions (noted below) to permit the compiler to check types.

The permitted values for *operation* are:

KEYCTL_GET_KEYRING_ID (since Linux 2.6.10)
Map a special key ID to a real key ID for this process.

This operation looks up the special key whose ID is provided in *arg2* (cast to *key_serial_t*). If the special key is found, the ID of the corresponding real key is returned as the function result. The following values may be specified in *arg2*:

KEY_SPEC_THREAD_KEYRING
This specifies the calling thread's thread-specific keyring. See *thread-keyring*(7).

KEY_SPEC_PROCESS_KEYRING
This specifies the caller's process-specific keyring. See *process-keyring*(7).

KEY_SPEC_SESSION_KEYRING
This specifies the caller's session-specific keyring. See *session-keyring*(7).

KEY_SPEC_USER_KEYRING
This specifies the caller's UID-specific keyring. See *user-keyring*(7).

KEY_SPEC_USER_SESSION_KEYRING
This specifies the caller's UID-session keyring. See *user-session-keyring*(7).

KEY_SPEC_REQKEY_AUTH_KEY (since Linux 2.6.16)
This specifies the authorization key created by *request_key*(2) and passed to the process it spawns to generate a key. This key is available only in a *request-key*(8)-style program that was passed an authorization key by the kernel and ceases to be available once the requested key has been instantiated; see *request_key*(2).

KEY_SPEC_REQUESTOR_KEYRING (since Linux 2.6.29)
This specifies the key ID for the *request_key*(2) destination keyring. This keyring is available only in a *request-key*(8)-style program that was passed an authorization key by the kernel and ceases to be available once the requested key has been instantiated; see *request_key*(2).

The behavior if the key specified in *arg2* does not exist depends on the value of *arg3* (cast to *int*). If *arg3* contains a nonzero value, then—if it is appropriate to do so (e.g., when looking up the user, user-session, or session key)—a new key is created and its real key ID returned as the function result. Otherwise, the operation fails with the error **ENOKEY**.

If a valid key ID is specified in *arg2*, and the key exists, then this operation simply returns the key ID. If the key does not exist, the call fails with error **ENOKEY**.

The caller must have *search* permission on a keyring in order for it to be found.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_get_keyring_ID*(3)

KEYCTL_JOIN_SESSION_KEYRING (since Linux 2.6.10)
Replace the session keyring this process subscribes to with a new session keyring.

If *arg2* is NULL, an anonymous keyring with the description "_ses" is created and the process is subscribed to that keyring as its session keyring, displacing the previous session keyring.

Otherwise, *arg2* (cast to *char* *) is treated as the description (name) of a keyring, and the behavior is as follows:

- If a keyring with a matching description exists, the process will attempt to subscribe to that keyring as its session keyring if possible; if that is not possible, an error is returned. In order to subscribe to the keyring, the caller must have *search* permission on the keyring.
- If a keyring with a matching description does not exist, then a new keyring with the specified description is created, and the process is subscribed to that keyring as its session keyring.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_join_session_keyring*(3)

KEYCTL_UPDATE (since Linux 2.6.10)
Update a key's data payload.

The *arg2* argument (cast to *key_serial_t*) specifies the ID of the key to be updated. The *arg3* argument (cast to *void* *) points to the new payload and *arg4* (cast to *size_t*) contains the new payload size in bytes.

The caller must have *write* permission on the key specified and the key type must support updating.

A negatively instantiated key (see the description of **KEYCTL_REJECT**) can be positively instantiated with this operation.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_update*(3)

KEYCTL_REVOKE (since Linux 2.6.10)
Revoke the key with the ID provided in *arg2* (cast to *key_serial_t*). The key is scheduled for garbage collection; it will no longer be findable, and will be unavailable for further operations. Further attempts to use the key will fail with the error **EKEYREVOKED**.

The caller must have *write* or *setattr* permission on the key.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_revoke*(3)

KEYCTL_CHOWN (since Linux 2.6.10)
Change the ownership (user and group ID) of a key.

The *arg2* argument (cast to *key_serial_t*) contains the key ID. The *arg3* argument (cast to *uid_t*) contains the new user ID (or −1 in case the user ID shouldn't be changed). The *arg4* argument (cast to *gid_t*) contains the new group ID (or −1 in case the group ID shouldn't be changed).

The key must grant the caller *setattr* permission.

For the UID to be changed, or for the GID to be changed to a group the caller is not a member of, the caller must have the **CAP_SYS_ADMIN** capability (see *capabilities*(7)).

If the UID is to be changed, the new user must have sufficient quota to accept the key. The quota deduction will be removed from the old user to the new user should the UID be changed.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_chown*(3)

KEYCTL_SETPERM (since Linux 2.6.10)

Change the permissions of the key with the ID provided in the *arg2* argument (cast to *key_serial_t*) to the permissions provided in the *arg3* argument (cast to *key_perm_t*).

If the caller doesn't have the **CAP_SYS_ADMIN** capability, it can change permissions only for the keys it owns. (More precisely: the caller's filesystem UID must match the UID of the key.)

The key must grant *setattr* permission to the caller *regardless* of the caller's capabilities.

The permissions in *arg3* specify masks of available operations for each of the following user categories:

possessor (since Linux 2.6.14)
: This is the permission granted to a process that possesses the key (has it attached searchably to one of the process's keyrings); see *keyrings*(7).

user
: This is the permission granted to a process whose filesystem UID matches the UID of the key.

group
: This is the permission granted to a process whose filesystem GID or any of its supplementary GIDs matches the GID of the key.

other
: This is the permission granted to other processes that do not match the *user* and *group* categories.

The *user*, *group*, and *other* categories are exclusive: if a process matches the *user* category, it will not receive permissions granted in the *group* category; if a process matches the *user* or *group* category, then it will not receive permissions granted in the *other* category.

The *possessor* category grants permissions that are cumulative with the grants from the *user*, *group*, or *other* category.

Each permission mask is eight bits in size, with only six bits currently used. The available permissions are:

view
: This permission allows reading attributes of a key.

  This permission is required for the **KEYCTL_DESCRIBE** operation.

  The permission bits for each category are **KEY_POS_VIEW**, **KEY_USR_VIEW**, **KEY_GRP_VIEW**, and **KEY_OTH_VIEW**.

read
: This permission allows reading a key's payload.

  This permission is required for the **KEYCTL_READ** operation.

  The permission bits for each category are **KEY_POS_READ**, **KEY_USR_READ**, **KEY_GRP_READ**, and **KEY_OTH_READ**.

write
: This permission allows update or instantiation of a key's payload. For a keyring, it allows keys to be linked and unlinked from the keyring,

  This permission is required for the **KEYCTL_UPDATE**, **KEYCTL_REVOKE**, **KEYCTL_CLEAR**, **KEYCTL_LINK**, and **KEYCTL_UNLINK** operations.

  The permission bits for each category are **KEY_POS_WRITE**, **KEY_USR_WRITE**, **KEY_GRP_WRITE**, and **KEY_OTH_WRITE**.

search
: This permission allows keyrings to be searched and keys to be found. Searches can recurse only into nested keyrings that have *search* permission set.

  This permission is required for the **KEYCTL_GET_KEYRING_ID**, **KEYCTL_JOIN_SESSION_KEYRING**, **KEYCTL_SEARCH**, and **KEYCTL_INVALIDATE** operations.

  The permission bits for each category are **KEY_POS_SEARCH**, **KEY_USR_SEARCH**, **KEY_GRP_SEARCH**, and **KEY_OTH_SEARCH**.

link This permission allows a key or keyring to be linked to.

This permission is required for the **KEYCTL_LINK** and **KEYCTL_SESSION_TO_PARENT** operations.

The permission bits for each category are **KEY_POS_LINK**, **KEY_USR_LINK**, **KEY_GRP_LINK**, and **KEY_OTH_LINK**.

setattr (since Linux 2.6.15).
This permission allows a key's UID, GID, and permissions mask to be changed.

This permission is required for the **KEYCTL_REVOKE**, **KEYCTL_CHOWN**, and **KEYCTL_SETPERM** operations.

The permission bits for each category are **KEY_POS_SETATTR**, **KEY_USR_SETATTR**, **KEY_GRP_SETATTR**, and **KEY_OTH_SETATTR**.

As a convenience, the following macros are defined as masks for all of the permission bits in each of the user categories: **KEY_POS_ALL**, **KEY_USR_ALL**, **KEY_GRP_ALL**, and **KEY_OTH_ALL**.

The *arg4* and *arg5* arguments are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_setperm*(3)

KEYCTL_DESCRIBE (since Linux 2.6.10)
Obtain a string describing the attributes of a specified key.

The ID of the key to be described is specified in *arg2* (cast to *key_serial_t*). The descriptive string is returned in the buffer pointed to by *arg3* (cast to *char **); *arg4* (cast to *size_t*) specifies the size of that buffer in bytes.

The key must grant the caller *view* permission.

The returned string is null-terminated and contains the following information about the key:

type;*uid*;*gid*;*perm*;*description*

In the above, *type* and *description* are strings, *uid* and *gid* are decimal strings, and *perm* is a hexadecimal permissions mask. The descriptive string is written with the following format:

```
%s;%d;%d;%08x;%s
```

Note: the intention is that the descriptive string should be extensible in future kernel versions. In particular, the *description* field will not contain semicolons; it should be parsed by working backwards from the end of the string to find the last semicolon. This allows future semicolon-delimited fields to be inserted in the descriptive string in the future.

Writing to the buffer is attempted only when *arg3* is non-NULL and the specified buffer size is large enough to accept the descriptive string (including the terminating null byte). In order to determine whether the buffer size was too small, check to see if the return value of the operation is greater than *arg4*.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_describe*(3)

KEYCTL_CLEAR
Clear the contents of (i.e., unlink all keys from) a keyring.

The ID of the key (which must be of keyring type) is provided in *arg2* (cast to *key_serial_t*).

The caller must have *write* permission on the keyring.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_clear*(3)

KEYCTL_LINK (since Linux 2.6.10)
Create a link from a keyring to a key.

The key to be linked is specified in *arg2* (cast to *key_serial_t*); the keyring is specified in *arg3* (cast to *key_serial_t*).

If a key with the same type and description is already linked in the keyring, then that key is displaced from the keyring.

Before creating the link, the kernel checks the nesting of the keyrings and returns appropriate errors if the link would produce a cycle or if the nesting of keyrings would be too deep (The limit on the nesting of keyrings is determined by the kernel constant **KEYRING_SEARCH_MAX_DEPTH**, defined with the value 6, and is necessary to prevent overflows on the kernel stack when recursively searching keyrings).

The caller must have *link* permission on the key being added and *write* permission on the keyring.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_link*(3)

KEYCTL_UNLINK (since Linux 2.6.10)
Unlink a key from a keyring.

The ID of the key to be unlinked is specified in *arg2* (cast to *key_serial_t*); the ID of the keyring from which it is to be unlinked is specified in *arg3* (cast to *key_serial_t*).

If the key is not currently linked into the keyring, an error results.

The caller must have *write* permission on the keyring from which the key is being removed.

If the last link to a key is removed, then that key will be scheduled for destruction.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_unlink*(3)

KEYCTL_SEARCH (since Linux 2.6.10)
Search for a key in a keyring tree, returning its ID and optionally linking it to a specified keyring.

The tree to be searched is specified by passing the ID of the head keyring in *arg2* (cast to *key_serial_t*). The search is performed breadth-first and recursively.

The *arg3* and *arg4* arguments specify the key to be searched for: *arg3* (cast as *char* *) contains the key type (a null-terminated character string up to 32 bytes in size, including the terminating null byte), and *arg4* (cast as *char* *) contains the description of the key (a null-terminated character string up to 4096 bytes in size, including the terminating null byte).

The source keyring must grant *search* permission to the caller. When performing the recursive search, only keyrings that grant the caller *search* permission will be searched. Only keys with for which the caller has *search* permission can be found.

If the key is found, its ID is returned as the function result.

If the key is found and *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the key is linked into the keyring whose ID is specified in *arg5*. If the destination keyring specified in *arg5* already contains a link to a key that has the same type and description, then that link will be displaced by a link to the key found by this operation.

Instead of valid existing keyring IDs, the source (*arg2*) and destination (*arg5*) keyrings can be one of the special keyring IDs listed under **KEYCTL_GET_KEYRING_ID**.

This operation is exposed by *libkeyutils* via the function *keyctl_search*(3)

KEYCTL_READ (since Linux 2.6.10)
Read the payload data of a key.

The ID of the key whose payload is to be read is specified in *arg2* (cast to *key_serial_t*). This can be the ID of an existing key, or any of the special key IDs listed for **KEYCTL_GET_KEYRING_ID**.

The payload is placed in the buffer pointed by *arg3* (cast to *char* *); the size of that buffer must be specified in *arg4* (cast to *size_t*).

The returned data will be processed for presentation according to the key type. For example, a keyring will return an array of *key_serial_t* entries representing the IDs of all the keys that are linked to it. The *user* key type will return its data as is. If a key type does not implement this function, the operation fails with the error **EOPNOTSUPP**.

If *arg3* is not NULL, as much of the payload data as will fit is copied into the buffer. On a successful return, the return value is always the total size of the payload data. To determine whether the buffer was of sufficient size, check to see that the return value is less than or equal to the value supplied in *arg4*.

The key must either grant the caller *read* permission, or grant the caller *search* permission when searched for from the process keyrings (i.e., the key is possessed).

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_read*(3)

KEYCTL_INSTANTIATE (since Linux 2.6.10)
(Positively) instantiate an uninstantiated key with a specified payload.

The ID of the key to be instantiated is provided in *arg2* (cast to *key_serial_t*).

The key payload is specified in the buffer pointed to by *arg3* (cast to *void* *); the size of that buffer is specified in *arg4* (cast to *size_t*).

The payload may be a null pointer and the buffer size may be 0 if this is supported by the key type (e.g., it is a keyring).

The operation may be fail if the payload data is in the wrong format or is otherwise invalid.

If *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the instantiated key is linked into the keyring whose ID specified in *arg5*.

The caller must have the appropriate authorization key, and once the uninstantiated key has been instantiated, the authorization key is revoked. In other words, this operation is available only from a *request-key*(8)-style program. See *request_key*(2) for an explanation of uninstantiated keys and key instantiation.

This operation is exposed by *libkeyutils* via the function *keyctl_instantiate*(3)

KEYCTL_NEGATE (since Linux 2.6.10)
Negatively instantiate an uninstantiated key.

This operation is equivalent to the call:

```
keyctl(KEYCTL_REJECT, arg2, arg3, ENOKEY, arg4);
```

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_negate*(3)

KEYCTL_SET_REQKEY_KEYRING (since Linux 2.6.13)
Set the default keyring to which implicitly requested keys will be linked for this thread, and return the previous setting. Implicit key requests are those made by internal kernel components, such as can occur when, for example, opening files on an AFS or NFS filesystem. Setting the default keyring also has an effect when requesting a key from user space; see *request_key*(2) for details.

The *arg2* argument (cast to *int*) should contain one of the following values, to specify the new default keyring:

KEY_REQKEY_DEFL_NO_CHANGE
Don't change the default keyring. This can be used to discover the current default keyring (without changing it).

KEY_REQKEY_DEFL_DEFAULT
This selects the default behaviour, which is to use the thread-specific keyring if there is one, otherwise the process-specific keyring if there is one, otherwise the session keyring if there is one, otherwise the UID-specific session keyring, otherwise the user-specific keyring.

KEY_REQKEY_DEFL_THREAD_KEYRING
Use the thread-specific keyring (**thread−keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_PROCESS_KEYRING
Use the process-specific keyring (**process−keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_SESSION_KEYRING
Use the session-specific keyring (**session−keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_USER_KEYRING
Use the UID-specific keyring (**user−keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_USER_SESSION_KEYRING
Use the UID-specific session keyring (**user−session−keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_REQUESTOR_KEYRING (since Linux 2.6.29)
Use the requestor keyring.

All other values are invalid.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

The setting controlled by this operation is inherited by the child of *fork*(2) and preserved across *execve*(2).

This operation is exposed by *libkeyutils* via the function *keyctl_set_reqkey_keyring*(3)

KEYCTL_SET_TIMEOUT (since Linux 2.6.16)
Set a timeout on a key.

The ID of the key is specified in *arg2* (cast to *key_serial_t*). The timeout value, in seconds from the current time, is specified in *arg3* (cast to *unsigned int*). The timeout is measured against the realtime clock.

Specifying the timeout value as 0 clears any existing timeout on the key.

The */proc/keys* file displays the remaining time until each key will expire. (This is the only method of discovering the timeout on a key.)

The caller must either have the *setattr* permission on the key or hold an instantiation authorization token for the key (see *request_key*(2)).

The key and any links to the key will be automatically garbage collected after the timeout expires. Subsequent attempts to access the key will then fail with the error **EKEYEXPIRED**.

This operation cannot be used to set timeouts on revoked, expired, or negatively instantiated keys.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_set_timeout*(3)

KEYCTL_ASSUME_AUTHORITY (since Linux 2.6.16)
Assume (or divest) the authority for the calling thread to instantiate a key.

The *arg2* argument (cast to *key_serial_t*) specifies either a nonzero key ID to assume authority, or the value 0 to divest authority.

If *arg2* is nonzero, then it specifies the ID of an uninstantiated key for which authority is to be assumed. That key can then be instantiated using one of **KEYCTL_INSTANTIATE**, **KEYCTL_INSTANTIATE_IOV**, **KEYCTL_REJECT**, or **KEYCTL_NEGATE**. Once the key has been instantiated, the thread is automatically divested of authority to instantiate the key.

Authority over a key can be assumed only if the calling thread has present in its keyrings the authorization key that is associated with the specified key. (In other words, the **KEYCTL_ASSUME_AUTHORITY** operation is available only from a *request-key*(8)-style program; see *request_key*(2) for an explanation of how this operation is used.) The caller must have *search* permission on the authorization key.

If the specified key has a matching authorization key, then the ID of that key is returned. The authorization key can be read (**KEYCTL_READ**) to obtain the callout information passed to *request_key*(2).

If the ID given in *arg2* is 0, then the currently assumed authority is cleared (divested), and the value 0 is returned.

The **KEYCTL_ASSUME_AUTHORITY** mechanism allows a program such as *request-key*(8) to assume the necessary authority to instantiate a new uninstantiated key that was created as a consequence of a call to *request_key*(2). For further information, see *request_key*(2) and the kernel source file *Documentation/security/keys−request−key.txt*.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_assume_authority*(3)

KEYCTL_GET_SECURITY (since Linux 2.6.26)
Get the LSM (Linux Security Module) security label of the specified key.

The ID of the key whose security label is to be fetched is specified in *arg2* (cast to *key_serial_t*). The security label (terminated by a null byte) will be placed in the buffer pointed to by *arg3* argument (cast to *char **); the size of the buffer must be provided in *arg4* (cast to *size_t*).

If *arg3* is specified as NULL or the buffer size specified in *arg4* is too small, the full size of the security label string (including the terminating null byte) is returned as the function result, and nothing is copied to the buffer.

The caller must have *view* permission on the specified key.

The returned security label string will be rendered in a form appropriate to the LSM in force. For example, with SELinux, it may look like:

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

If no LSM is currently in force, then an empty string is placed in the buffer.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the functions *keyctl_get_security*(3) and *keyctl_get_security_alloc*(3)

KEYCTL_SESSION_TO_PARENT (since Linux 2.6.32)
Replace the session keyring to which the *parent* of the calling process subscribes with the session keyring of the calling process.

The keyring will be replaced in the parent process at the point where the parent next transitions from kernel space to user space.

The keyring must exist and must grant the caller *link* permission. The parent process must be single-threaded and have the same effective ownership as this process and must not be set-user-ID or set-group-ID. The UID of the parent process's existing session keyring (f it has one), as well as the UID of the caller's session keyring much match the caller's effective UID.

The fact that it is the parent process that is affected by this operation allows a program such as the shell to start a child process that uses this operation to change the shell's session keyring. (This is what the *keyctl*(1) **new_session** command does.)

The arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_session_to_parent*(3)

KEYCTL_REJECT (since Linux 2.6.39)
Mark a key as negatively instantiated and set an expiration timer on the key. This operation provides a superset of the functionality of the earlier **KEYCTL_NEGATE** operation.

The ID of the key that is to be negatively instantiated is specified in *arg2* (cast to *key_serial_t*). The *arg3* (cast to *unsigned int*) argument specifies the lifetime of the key, in seconds. The *arg4* argument (cast to *unsigned int*) specifies the error to be returned when a search hits this key; typically, this is one of **EKEYREJECTED**, **EKEYREVOKED**, or **EKEYEXPIRED**.

If *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the negatively instantiated key is linked into the keyring whose ID is specified in *arg5*.

The caller must have the appropriate authorization key. In other words, this operation is available only from a *request-key*(8)-style program. See *request_key*(2).

The caller must have the appropriate authorization key, and once the uninstantiated key has been instantiated, the authorization key is revoked. In other words, this operation is available only from a *request-key*(8)-style program. See *request_key*(2) for an explanation of uninstantiated keys and key instantiation.

This operation is exposed by *libkeyutils* via the function *keyctl_reject*(3)

KEYCTL_INSTANTIATE_IOV (since Linux 2.6.39)
Instantiate an uninstantiated key with a payload specified via a vector of buffers.

This operation is the same as **KEYCTL_INSTANTIATE**, but the payload data is specified as an array of *iovec* structures (see *iovec*(3type)).

The pointer to the payload vector is specified in *arg3* (cast as *const struct iovec* *). The number of items in the vector is specified in *arg4* (cast as *unsigned int*).

The *arg2* (key ID) and *arg5* (keyring ID) are interpreted as for **KEYCTL_INSTANTIATE**.

This operation is exposed by *libkeyutils* via the function *keyctl_instantiate_iov*(3)

KEYCTL_INVALIDATE (since Linux 3.5)
Mark a key as invalid.

The ID of the key to be invalidated is specified in *arg2* (cast to *key_serial_t*).

To invalidate a key, the caller must have *search* permission on the key.

This operation marks the key as invalid and schedules immediate garbage collection. The garbage collector removes the invalidated key from all keyrings and deletes the key when its reference count reaches zero. After this operation, the key will be ignored by all searches, even if it is not yet deleted.

Keys that are marked invalid become invisible to normal key operations immediately, though they are still visible in */proc/keys* (marked with an 'i' flag) until they are actually removed.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_invalidate*(3)

KEYCTL_GET_PERSISTENT (since Linux 3.13)
Get the persistent keyring (**persistent−keyring**(7)) for a specified user and link it to a specified keyring.

The user ID is specified in *arg2* (cast to *uid_t*). If the value −1 is specified, the caller's real user ID is used. The ID of the destination keyring is specified in *arg3* (cast to *key_serial_t*).

The caller must have the **CAP_SETUID** capability in its user namespace in order to fetch the persistent keyring for a user ID that does not match either the real or effective user ID of the caller.

If the call is successful, a link to the persistent keyring is added to the keyring whose ID was specified in *arg3*.

The caller must have *write* permission on the keyring.

The persistent keyring will be created by the kernel if it does not yet exist.

Each time the **KEYCTL_GET_PERSISTENT** operation is performed, the persistent keyring will have its expiration timeout reset to the value in:

```
/proc/sys/kernel/keys/persistent_keyring_expiry
```

Should the timeout be reached, the persistent keyring will be removed and everything it pins can then be garbage collected.

Persistent keyrings were added in Linux 3.13.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function *keyctl_get_persistent*(3)

KEYCTL_DH_COMPUTE (since Linux 4.7)
Compute a Diffie-Hellman shared secret or public key, optionally applying key derivation function (KDF) to the result.

The *arg2* argument is a pointer to a set of parameters containing serial numbers for three *"user"* keys used in the Diffie-Hellman calculation, packaged in a structure of the following form:

```
struct keyctl_dh_params {
    int32_t private; /* The local private key */
    int32_t prime; /* The prime, known to both parties */
    int32_t base;  /* The base integer: either a shared
                      generator or the remote public key */
};
```

Each of the three keys specified in this structure must grant the caller *read* permission. The payloads of these keys are used to calculate the Diffie-Hellman result as:

```
base ^ private mod prime
```

If the base is the shared generator, the result is the local public key. If the base is the remote public key, the result is the shared secret.

The *arg3* argument (cast to *char* *) points to a buffer where the result of the calculation is placed. The size of that buffer is specified in *arg4* (cast to *size_t*).

The buffer must be large enough to accommodate the output data, otherwise an error is returned. If *arg4* is specified zero, in which case the buffer is not used and the operation returns the minimum required buffer size (i.e., the length of the prime).

Diffie-Hellman computations can be performed in user space, but require a multiple-precision integer (MPI) library. Moving the implementation into the kernel gives access to the kernel MPI implementation, and allows access to secure or acceleration hardware.

Adding support for DH computation to the **keyctl**() system call was considered a good fit due to the DH algorithm's use for deriving shared keys; it also allows the type of the key to determine which DH implementation (software or hardware) is appropriate.

If the *arg5* argument is **NULL**, then the DH result itself is returned. Otherwise (since Linux 4.12), it is a pointer to a structure which specifies parameters of the KDF operation to be applied:

```
struct keyctl_kdf_params {
    char *hashname;     /* Hash algorithm name */
    char *otherinfo;    /* SP800-56A OtherInfo */
    __u32 otherinfolen; /* Length of otherinfo data */
    __u32 __spare[8];   /* Reserved */
};
```

The *hashname* field is a null-terminated string which specifies a hash name (available in the kernel's crypto API; the list of the hashes available is rather tricky to observe; please refer to the "Kernel Crypto API Architecture" documentation for the information regarding how hash names are constructed and your kernel's source and configuration regarding what ciphers and templates with type **CRYPTO_ALG_TYPE_SHASH** are available) to be applied to DH result in KDF operation.

The *otherinfo* field is an *OtherInfo* data as described in SP800-56A section 5.8.1.2 and is algorithm-specific. This data is concatenated with the result of DH operation and is provided as an input to the KDF operation. Its size is provided in the *otherinfolen* field and is limited by **KEYCTL_KDF_MAX_OI_LEN** constant that defined in *security/keys/internal.h* to a value of 64.

The **__spare** field is currently unused. It was ignored until Linux 4.13 (but still should be user-addressable since it is copied to the kernel), and should contain zeros since Linux 4.13.

The KDF implementation complies with SP800-56A as well as with SP800-108 (the counter KDF).

This operation is exposed by *libkeyutils* (from *libkeyutils* 1.5.10 onwards) via the functions *keyctl_dh_compute*(3) and *keyctl_dh_compute_alloc*(3)

KEYCTL_RESTRICT_KEYRING (since Linux 4.12)
Apply a key-linking restriction to the keyring with the ID provided in *arg2* (cast to *key_serial_t*). The caller must have *setattr* permission on the key. If *arg3* is NULL, any attempt to add a key to the keyring is blocked; otherwise it contains a pointer to a string with a key type name and *arg4* contains a pointer to string that describes the type-specific restriction. As of Linux 4.12, only the type "asymmetric" has restrictions defined:

builtin_trusted
Allows only keys that are signed by a key linked to the built-in keyring (".builtin_trusted_keys").

builtin_and_secondary_trusted
Allows only keys that are signed by a key linked to the secondary keyring (".secondary_trusted_keys") or, by extension, a key in a built-in keyring, as the latter is linked to the former.

key_or_keyring:*key*
key_or_keyring:*key***:chain**
If *key* specifies the ID of a key of type "asymmetric", then only keys that are signed by this key are allowed.

If *key* specifies the ID of a keyring, then only keys that are signed by a key linked to this keyring are allowed.

If ":chain" is specified, keys that are signed by a keys linked to the destination keyring (that is, the keyring with the ID specified in the *arg2* argument) are also allowed.

Note that a restriction can be configured only once for the specified keyring; once a restriction is set, it can't be overridden.

The argument *arg5* is ignored.

RETURN VALUE

For a successful call, the return value depends on the operation:

KEYCTL_GET_KEYRING_ID
The ID of the requested keyring.

KEYCTL_JOIN_SESSION_KEYRING
The ID of the joined session keyring.

KEYCTL_DESCRIBE
The size of the description (including the terminating null byte), irrespective of the provided buffer size.

KEYCTL_SEARCH
The ID of the key that was found.

KEYCTL_READ
The amount of data that is available in the key, irrespective of the provided buffer size.

KEYCTL_SET_REQKEY_KEYRING
The ID of the previous default keyring to which implicitly requested keys were linked (one of **KEY_REQKEY_DEFL_USER_***).

KEYCTL_ASSUME_AUTHORITY
Either 0, if the ID given was 0, or the ID of the authorization key matching the specified key, if a nonzero key ID was provided.

KEYCTL_GET_SECURITY
The size of the LSM security label string (including the terminating null byte), irrespective of the provided buffer size.

KEYCTL_GET_PERSISTENT
The ID of the persistent keyring.

KEYCTL_DH_COMPUTE
The number of bytes copied to the buffer, or, if *arg4* is 0, the required buffer size.

All other operations
Zero.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
The requested operation wasn't permitted.

EAGAIN
operation was **KEYCTL_DH_COMPUTE** and there was an error during crypto module initialization.

EDEADLK
operation was **KEYCTL_LINK** and the requested link would result in a cycle.

EDEADLK
operation was **KEYCTL_RESTRICT_KEYRING** and the requested keyring restriction would result in a cycle.

EDQUOT
The key quota for the caller's user would be exceeded by creating a key or linking it to the keyring.

EEXIST
operation was **KEYCTL_RESTRICT_KEYRING** and keyring provided in *arg2* argument already has a restriction set.

EFAULT
operation was **KEYCTL_DH_COMPUTE** and one of the following has failed:

- copying of the *struct keyctl_dh_params*, provided in the *arg2* argument, from user space;
- copying of the *struct keyctl_kdf_params*, provided in the non-NULL *arg5* argument, from user space (in case kernel supports performing KDF operation on DH operation result);
- copying of data pointed by the *hashname* field of the *struct keyctl_kdf_params* from user space;
- copying of data pointed by the *otherinfo* field of the *struct keyctl_kdf_params* from user space if the *otherinfolen* field was nonzero;
- copying of the result to user space.

EINVAL
operation was **KEYCTL_SETPERM** and an invalid permission bit was specified in *arg3*.

EINVAL
operation was **KEYCTL_SEARCH** and the size of the description in *arg4* (including the terminating null byte) exceeded 4096 bytes.

EINVAL
size of the string (including the terminating null byte) specified in *arg3* (the key type) or *arg4* (the key description) exceeded the limit (32 bytes and 4096 bytes respectively).

EINVAL (before Linux 4.12)
operation was **KEYCTL_DH_COMPUTE**, argument *arg5* was non-NULL.

EINVAL
operation was **KEYCTL_DH_COMPUTE** And the digest size of the hashing algorithm supplied is zero.

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the buffer size provided is not enough to hold the result. Provide 0 as a buffer size in order to obtain the minimum buffer size.

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the hash name provided in the *hashname* field of the *struct keyctl_kdf_params* pointed by *arg5* argument is too big (the limit is implementation-specific and varies between kernel versions, but it is deemed big enough for all valid algorithm names).

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the *__spare* field of the *struct keyctl_kdf_params* provided in the *arg5* argument contains nonzero values.

EKEYEXPIRED
An expired key was found or specified.

EKEYREJECTED
A rejected key was found or specified.

EKEYREVOKED
A revoked key was found or specified.

ELOOP
operation was **KEYCTL_LINK** and the requested link would cause the maximum nesting depth for keyrings to be exceeded.

EMSGSIZE
operation was **KEYCTL_DH_COMPUTE** and the buffer length exceeds **KEYCTL_KDF_MAX_OUTPUT_LEN** (which is 1024 currently) or the *otherinfolen* field of the *struct keyctl_kdf_parms* passed in *arg5* exceeds **KEYCTL_KDF_MAX_OI_LEN** (which is 64 currently).

ENFILE (before Linux 3.13)
operation was **KEYCTL_LINK** and the keyring is full. (Before Linux 3.13, the available space for storing keyring links was limited to a single page of memory; since Linux 3.13, there is no fixed limit.)

ENOENT
operation was **KEYCTL_UNLINK** and the key to be unlinked isn't linked to the keyring.

ENOENT
operation was **KEYCTL_DH_COMPUTE** and the hashing algorithm specified in the *hashname* field of the *struct keyctl_kdf_params* pointed by *arg5* argument hasn't been found.

ENOENT
operation was **KEYCTL_RESTRICT_KEYRING** and the type provided in *arg3* argument doesn't support setting key linking restrictions.

ENOKEY
No matching key was found or an invalid key was specified.

ENOKEY
The value **KEYCTL_GET_KEYRING_ID** was specified in *operation*, the key specified in *arg2* did not exist, and *arg3* was zero (meaning don't create the key if it didn't exist).

ENOMEM
One of kernel memory allocation routines failed during the execution of the syscall.

ENOTDIR
A key of keyring type was expected but the ID of a key with a different type was provided.

EOPNOTSUPP
operation was **KEYCTL_READ** and the key type does not support reading (e.g., the type is *"login"*).

EOPNOTSUPP
operation was **KEYCTL_UPDATE** and the key type does not support updating.

EOPNOTSUPP
operation was **KEYCTL_RESTRICT_KEYRING**, the type provided in *arg3* argument was "asymmetric", and the key specified in the restriction specification provided in *arg4* has type other than "asymmetric" or "keyring".

EPERM
operation was **KEYCTL_GET_PERSISTENT**, *arg2* specified a UID other than the calling thread's real or effective UID, and the caller did not have the **CAP_SETUID** capability.

EPERM
operation was **KEYCTL_SESSION_TO_PARENT** and either: all of the UIDs (GIDs) of the parent process do not match the effective UID (GID) of the calling process; the UID of the parent's existing session keyring or the UID of the caller's session keyring did not match the effective UID of the caller; the parent process is not single-thread; or the parent process is *init*(1) or a kernel thread.

ETIMEDOUT
operation was **KEYCTL_DH_COMPUTE** and the initialization of crypto modules has timed out.

VERSIONS
A wrapper is provided in the *libkeyutils* library. (The accompanying package provides the <*keyutils.h*> header file.) However, rather than using this system call directly, you probably want to use the various library functions mentioned in the descriptions of individual operations above.

STANDARDS
Linux.

HISTORY
Linux 2.6.10.

EXAMPLES
The program below provide subset of the functionality of the *request-key*(8) program provided by the *keyutils* package. For informational purposes, the program records various information in a log file.

As described in *request_key*(2), the *request-key*(8) program is invoked with command-line arguments that describe a key that is to be instantiated. The example program fetches and logs these arguments. The program assumes authority to instantiate the requested key, and then instantiates that key.

The following shell session demonstrates the use of this program. In the session, we compile the program and then use it to temporarily replace the standard *request-key*(8) program. (Note that temporarily disabling the standard *request-key*(8) program may not be safe on some systems.) While our example program is installed, we use the example program shown in *request_key*(2) to request a key.

```
$ cc -o key_instantiate key_instantiate.c -lkeyutils
$ sudo mv /sbin/request-key /sbin/request-key.backup
$ sudo cp key_instantiate /sbin/request-key
$ ./t_request_key user mykey somepayloaddata
Key ID is 20d035bf
$ sudo mv /sbin/request-key.backup /sbin/request-key
```

Looking at the log file created by this program, we can see the command-line arguments supplied to our example program:

```
$ cat /tmp/key_instantiate.log
Time: Mon Nov  7 13:06:47 2016

Command line arguments:
  argv[0]:            /sbin/request-key
  operation:          create
  key_to_instantiate: 20d035bf
  UID:                1000
  GID:                1000
  thread_keyring:     0
  process_keyring:    0
  session_keyring:    256e6a6

Key description:      user;1000;1000;3f010000;mykey
Auth key payload:     somepayloaddata
Destination keyring:  256e6a6
```

```
Auth key description: .request_key_auth;1000;1000;0b010000;20d035bf
```

The last few lines of the above output show that the example program was able to fetch:

- the description of the key to be instantiated, which included the name of the key (*mykey*);
- the payload of the authorization key, which consisted of the data (*somepayloaddata*) passed to *request_key*(2);
- the destination keyring that was specified in the call to *request_key*(2); and
- the description of the authorization key, where we can see that the name of the authorization key matches the ID of the key that is to be instantiated (*20d035bf*).

The example program in *request_key*(2) specified the destination keyring as **KEY_SPEC_SESSION_KEYRING**. By examining the contents of */proc/keys*, we can see that this was translated to the ID of the destination keyring (*0256e6a6*) shown in the log output above; we can also see the newly created key with the name *mykey* and ID *20d035bf*.

```
$ cat /proc/keys | egrep 'mykey|256e6a6'
0256e6a6 I--Q---  194 perm 3f030000  1000  1000 keyring  _ses: 3
20d035bf I--Q---    1 perm 3f010000  1000  1000 user     mykey: 16
```

Program source

```
/* key_instantiate.c */

#include <errno.h>
#include <keyutils.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

#ifndef KEY_SPEC_REQUESTOR_KEYRING
#define KEY_SPEC_REQUESTOR_KEYRING      (-8)
#endif

int
main(int argc, char *argv[])
{
    int           akp_size;       /* Size of auth_key_payload */
    int           auth_key;
    char          dbuf[256];
    char          auth_key_payload[256];
    char          *operation;
    FILE          *fp;
    gid_t         gid;
    uid_t         uid;
    time_t        t;
    key_serial_t  key_to_instantiate, dest_keyring;
    key_serial_t  thread_keyring, process_keyring, session_keyring;

    if (argc != 8) {
        fprintf(stderr, "Usage: %s op key uid gid thread_keyring "
                        "process_keyring session_keyring\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fp = fopen("/tmp/key_instantiate.log", "w");
    if (fp == NULL)
```

```
    exit(EXIT_FAILURE);

setbuf(fp, NULL);

t = time(NULL);
fprintf(fp, "Time: %s\n", ctime(&t));

/*
 * The kernel passes a fixed set of arguments to the program
 * that it execs; fetch them.
 */
operation = argv[1];
key_to_instantiate = atoi(argv[2]);
uid = atoi(argv[3]);
gid = atoi(argv[4]);
thread_keyring = atoi(argv[5]);
process_keyring = atoi(argv[6]);
session_keyring = atoi(argv[7]);

fprintf(fp, "Command line arguments:\n");
fprintf(fp, "  argv[0]:            %s\n", argv[0]);
fprintf(fp, "  operation:          %s\n", operation);
fprintf(fp, "  key_to_instantiate: %jx\n",
        (uintmax_t) key_to_instantiate);
fprintf(fp, "  UID:                %jd\n", (intmax_t) uid);
fprintf(fp, "  GID:                %jd\n", (intmax_t) gid);
fprintf(fp, "  thread_keyring:     %jx\n",
        (uintmax_t) thread_keyring);
fprintf(fp, "  process_keyring:    %jx\n",
        (uintmax_t) process_keyring);
fprintf(fp, "  session_keyring:    %jx\n",
        (uintmax_t) session_keyring);
fprintf(fp, "\n");

/*
 * Assume the authority to instantiate the key named in argv[2].
 */
if (keyctl(KEYCTL_ASSUME_AUTHORITY, key_to_instantiate) == -1) {
    fprintf(fp, "KEYCTL_ASSUME_AUTHORITY failed: %s\n",
            strerror(errno));
    exit(EXIT_FAILURE);
}

/*
 * Fetch the description of the key that is to be instantiated.
 */
if (keyctl(KEYCTL_DESCRIBE, key_to_instantiate,
           dbuf, sizeof(dbuf)) == -1) {
    fprintf(fp, "KEYCTL_DESCRIBE failed: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}

fprintf(fp, "Key description:      %s\n", dbuf);

/*
 * Fetch the payload of the authorization key, which is
 * actually the callout data given to request_key().
 */
akp_size = keyctl(KEYCTL_READ, KEY_SPEC_REQKEY_AUTH_KEY,
```

```
                    auth_key_payload, sizeof(auth_key_payload));
    if (akp_size == -1) {
        fprintf(fp, "KEYCTL_READ failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    auth_key_payload[akp_size] = '\0';
    fprintf(fp, "Auth key payload:     %s\n", auth_key_payload);

    /*
     * For interest, get the ID of the authorization key and
     * display it.
     */
    auth_key = keyctl(KEYCTL_GET_KEYRING_ID,
                      KEY_SPEC_REQKEY_AUTH_KEY);
    if (auth_key == -1) {
        fprintf(fp, "KEYCTL_GET_KEYRING_ID failed: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Auth key ID:          %jx\n", (uintmax_t) auth_key);

    /*
     * Fetch key ID for the request_key(2) destination keyring.
     */
    dest_keyring = keyctl(KEYCTL_GET_KEYRING_ID,
                          KEY_SPEC_REQUESTOR_KEYRING);
    if (dest_keyring == -1) {
        fprintf(fp, "KEYCTL_GET_KEYRING_ID failed: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Destination keyring:  %jx\n", (uintmax_t) dest_keyring);

    /*
     * Fetch the description of the authorization key. This
     * allows us to see the key type, UID, GID, permissions,
     * and description (name) of the key. Among other things,
     * we will see that the name of the key is a hexadecimal
     * string representing the ID of the key to be instantiated.
     */
    if (keyctl(KEYCTL_DESCRIBE, KEY_SPEC_REQKEY_AUTH_KEY,
               dbuf, sizeof(dbuf)) == -1)
    {
        fprintf(fp, "KEYCTL_DESCRIBE failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Auth key description: %s\n", dbuf);

    /*
     * Instantiate the key using the callout data that was supplied
     * in the payload of the authorization key.
     */
    if (keyctl(KEYCTL_INSTANTIATE, key_to_instantiate,
               auth_key_payload, akp_size + 1, dest_keyring) == -1)
    {
```

```
            fprintf(fp, "KEYCTL_INSTANTIATE failed: %s\n",
                    strerror(errno));
            exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

keyctl(1), *add_key*(2), *request_key*(2), *keyctl*(3), *keyctl_assume_authority*(3), *keyctl_chown*(3), *keyctl_clear*(3), *keyctl_describe*(3), *keyctl_describe_alloc*(3), *keyctl_dh_compute*(3), *keyctl_dh_compute_alloc*(3), *keyctl_get_keyring_ID*(3), *keyctl_get_persistent*(3), *keyctl_get_security*(3), *keyctl_get_security_alloc*(3), *keyctl_instantiate*(3), *keyctl_instantiate_iov*(3), *keyctl_invalidate*(3), *keyctl_join_session_keyring*(3), *keyctl_link*(3), *keyctl_negate*(3), *keyctl_read*(3), *keyctl_read_alloc*(3), *keyctl_reject*(3), *keyctl_revoke*(3), *keyctl_search*(3), *keyctl_session_to_parent*(3), *keyctl_set_reqkey_keyring*(3), *keyctl_set_timeout*(3), *keyctl_setperm*(3), *keyctl_unlink*(3), *keyctl_update*(3), *recursive_key_scan*(3), *recursive_session_key_scan*(3), *capabilities*(7), *credentials*(7), *keyrings*(7), *keyutils*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user_namespaces*(7), *user-session-keyring*(7), *request-key*(8)

The kernel source files under *Documentation/security/keys/* (or, before Linux 4.13, in the file *Documentation/security/keys.txt*).

NAME

kill − send signal to a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

int kill(pid_t *pid***, int** *sig***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

kill():
_POSIX_C_SOURCE

DESCRIPTION

The **kill**() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than −1, then *sig* is sent to every process in the process group whose ID is *−pid*.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
An invalid signal was specified.

EPERM
The calling process does not have permission to send the signal to any of the target processes.

ESRCH
The target process or process group does not exist. Note that an existing process might be a zombie, a process that has terminated execution, but has not yet been *wait*(2)ed for.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

Linux notes

Across different kernel versions, Linux has enforced different rules for the permissions required for an unprivileged process to send a signal to another process. In Linux 1.0 to 1.2.2, a signal could be sent if the effective user ID of the sender matched effective user ID of the target, or the real user ID of the sender matched the real user ID of the target. From Linux 1.2.3 until 1.3.77, a signal could be sent if the effective user ID of the sender matched either the real or effective user ID of the target. The current rules, which conform to POSIX.1, were adopted in Linux 1.3.78.

NOTES

The only signals that can be sent to process ID 1, the *init* process, are those for which *init* has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.

POSIX.1 requires that *kill(−1,sig)* send *sig* to all processes that the calling process may send signals to,

except possibly for some implementation-defined system processes. Linux allows a process to signal itself, but on Linux the call *kill(−1,sig)* does not signal the calling process.

POSIX.1 requires that if a process sends a signal to itself, and the sending thread does not have the signal blocked, and no other thread has it unblocked or is waiting for it in *sigwait*(3), at least one unblocked signal must be delivered to the sending thread before the **kill**() returns.

BUGS

In Linux 2.6 up to and including Linux 2.6.7, there was a bug that meant that when sending signals to a process group, **kill**() failed with the error **EPERM** if the caller did not have permission to send the signal to *any* (rather than *all*) of the members of the process group. Notwithstanding this error return, the signal was still delivered to all of the processes for which the caller had permission to signal.

SEE ALSO

kill(1), *_exit*(2), *pidfd_send_signal*(2), *signal*(2), *tkill*(2), *exit*(3), *killpg*(3), *sigqueue*(3), *capabilities*(7), *credentials*(7), *signal*(7)

NAME

landlock_add_rule − add a new Landlock rule to a ruleset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/landlock.h> /* Definition of **LANDLOCK_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */

int syscall(SYS_landlock_add_rule, int *ruleset_fd***,**
enum landlock_rule_type *rule_type***,**
const void ***rule_attr,* **uint32_t** *flags***);**

DESCRIPTION

A Landlock rule describes an action on an object. An object is currently a file hierarchy, and the related filesystem actions are defined with a set of access rights. This **landlock_add_rule**() system call enables adding a new Landlock rule to an existing ruleset created with *landlock_create_ruleset*(2). See *landlock*(7) for a global overview.

ruleset_fd is a Landlock ruleset file descriptor obtained with *landlock_create_ruleset*(2).

rule_type identifies the structure type pointed to by *rule_attr*. Currently, Linux supports the following *rule_type* value:

LANDLOCK_RULE_PATH_BENEATH
This defines the object type as a file hierarchy. In this case, *rule_attr* points to the following structure:

```
struct landlock_path_beneath_attr {
    __u64 allowed_access;
    __s32 parent_fd;
} __attribute__((packed));
```

allowed_access contains a bitmask of allowed filesystem actions for this file hierarchy (see **Filesystem actions** in *landlock*(7)).

parent_fd is an opened file descriptor, preferably with the *O_PATH* flag, which identifies the parent directory of the file hierarchy or just a file.

flags must be 0.

RETURN VALUE

On success, **landlock_add_rule**() returns 0.

ERRORS

landlock_add_rule() can fail for the following reasons:

EOPNOTSUPP
Landlock is supported by the kernel but disabled at boot time.

EINVAL
flags is not 0, or the rule accesses are inconsistent (i.e., *rule_attr−>allowed_access* is not a subset of the ruleset handled accesses).

ENOMSG
Empty accesses (i.e., *rule_attr−>allowed_access* is 0).

EBADF
ruleset_fd is not a file descriptor for the current thread, or a member of *rule_attr* is not a file descriptor as expected.

EBADFD
ruleset_fd is not a ruleset file descriptor, or a member of *rule_attr* is not the expected file descriptor type.

EPERM
ruleset_fd has no write access to the underlying ruleset.

EFAULT
rule_attr was not a valid address.

STANDARDS
Linux.

HISTORY
Linux 5.13.

EXAMPLES
See *landlock*(7).

SEE ALSO
landlock_create_ruleset(2), *landlock_restrict_self*(2), *landlock*(7)

NAME

landlock_create_ruleset − create a new Landlock ruleset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/landlock.h> /* Definition of **LANDLOCK_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */

int syscall(SYS_landlock_create_ruleset,
const struct landlock_ruleset_attr ***attr*,**
size_t *size* **, uint32_t** *flags***);**

DESCRIPTION

A Landlock ruleset identifies a set of rules (i.e., actions on objects). This **landlock_create_ruleset**() system call enables creating a new file descriptor identifying a ruleset. This file descriptor can then be used by *landlock_add_rule*(2) and *landlock_restrict_self*(2). See *landlock*(7) for a global overview.

attr specifies the properties of the new ruleset. It points to the following structure:

```
struct landlock_ruleset_attr {
    __u64 handled_access_fs;
};
```

handled_access_fs is a bitmask of actions that is handled by this ruleset and should then be forbidden if no rule explicitly allows them (see **Filesystem actions** in *landlock*(7)). This enables simply restricting ambient rights (e.g., global filesystem access) and is needed for compatibility reasons.

size must be specified as *sizeof(struct landlock_ruleset_attr)* for compatibility reasons.

flags must be 0 if *attr* is used. Otherwise, *flags* can be set to:

LANDLOCK_CREATE_RULESET_VERSION
If *attr* is NULL and *size* is 0, then the returned value is the highest supported Landlock ABI version (starting at 1). This version can be used for a best-effort security approach, which is encouraged when user space is not pinned to a specific kernel version. All features documented in these man pages are available with the version 1.

RETURN VALUE

On success, **landlock_create_ruleset**() returns a new Landlock ruleset file descriptor, or a Landlock ABI version, according to *flags*.

ERRORS

landlock_create_ruleset() can fail for the following reasons:

EOPNOTSUPP
Landlock is supported by the kernel but disabled at boot time.

EINVAL
Unknown *flags*, or unknown access, or too small *size*.

E2BIG *size* is too big.

EFAULT
attr was not a valid address.

ENOMSG
Empty accesses (i.e., *attr−>handled_access_fs* is 0).

STANDARDS

Linux.

HISTORY

Linux 5.13.

EXAMPLES

See *landlock*(7).

SEE ALSO

landlock_add_rule(2), *landlock_restrict_self*(2), *landlock*(7)

NAME

landlock_restrict_self − enforce a Landlock ruleset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/landlock.h> /* Definition of **LANDLOCK_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */

int syscall(SYS_landlock_restrict_self, int *ruleset_fd***,**
uint32_t *flags***);**

DESCRIPTION

Once a Landlock ruleset is populated with the desired rules, the **landlock_restrict_self**() system call enables enforcing this ruleset on the calling thread. See *landlock*(7) for a global overview.

A thread can be restricted with multiple rulesets that are then composed together to form the thread's Landlock domain. This can be seen as a stack of rulesets but it is implemented in a more efficient way. A domain can only be updated in such a way that the constraints of each past and future composed rulesets will restrict the thread and its future children for their entire life. It is then possible to gradually enforce tailored access control policies with multiple independent rulesets coming from different sources (e.g., init system configuration, user session policy, built-in application policy). However, most applications should only need one call to **landlock_restrict_self**() and they should avoid arbitrary numbers of such calls because of the composed rulesets limit. Instead, developers are encouraged to build a tailored ruleset thanks to multiple calls to *landlock_add_rule*(2).

In order to enforce a ruleset, either the caller must have the **CAP_SYS_ADMIN** capability in its user namespace, or the thread must already have the *no_new_privs* bit set. As for *seccomp*(2), this avoids scenarios where unprivileged processes can affect the behavior of privileged children (e.g., because of set-user-ID binaries). If that bit was not already set by an ancestor of this thread, the thread must make the following call:

```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
```

ruleset_fd is a Landlock ruleset file descriptor obtained with *landlock_create_ruleset*(2) and fully populated with a set of calls to *landlock_add_rule*(2).

flags must be 0.

RETURN VALUE

On success, **landlock_restrict_self**() returns 0.

ERRORS

landlock_restrict_self() can fail for the following reasons:

EOPNOTSUPP
Landlock is supported by the kernel but disabled at boot time.

EINVAL
flags is not 0.

EBADF
ruleset_fd is not a file descriptor for the current thread.

EBADFD
ruleset_fd is not a ruleset file descriptor.

EPERM
ruleset_fd has no read access to the underlying ruleset, or the calling thread is not running with *no_new_privs*, or it doesn't have the **CAP_SYS_ADMIN** in its user namespace.

E2BIG The maximum number of composed rulesets is reached for the calling thread. This limit is currently 64.

STANDARDS

Linux.

HISTORY

Linux 5.13.

EXAMPLES

See *landlock*(7).

SEE ALSO

landlock_create_ruleset(2), *landlock_add_rule*(2), *landlock*(7)

NAME

link, linkat − make a new name for a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int linkat(int olddirfd, const char *oldpath,
           int newdirfd, const char *newpath, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

linkat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

link() creates a new link (also known as a hard link) to an existing file.

If *newpath* exists, it will *not* be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

linkat()

The **linkat**() system call operates in exactly the same way as **link**(), except for the differences described here.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by **link**() for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value **AT_FDCWD**, then *oldpath* is interpreted relative to the current working directory of the calling process (like *link*())

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor *newdirfd*.

The following values can be bitwise ORed in *flags*:

AT_EMPTY_PATH (since Linux 2.6.39)
If *oldpath* is an empty string, create a link to the file referenced by *olddirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *olddirfd* can refer to any type of file except a directory. This will generally not work if the file has a link count of zero (files created with **O_TMPFILE** and without **O_EXCL** are an exception). The caller must have the **CAP_DAC_READ_SEARCH** capability in order to use this flag. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_FOLLOW (since Linux 2.6.18)
By default, **linkat**(), does not dereference *oldpath* if it is a symbolic link (like *link*())The flag **AT_SYMLINK_FOLLOW** can be specified in *flags* to cause *oldpath* to be dereferenced if it is a symbolic link. If procfs is mounted, this can be used as an alternative to **AT_EMPTY_PATH**, like this:

```
linkat(AT_FDCWD, "/proc/self/fd/<fd>", newdirfd,
       newname, AT_SYMLINK_FOLLOW);
```

Before Linux 2.6.18, the *flags* argument was unused, and had to be specified as 0.

See *openat*(2) for an explanation of the need for **linkat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Write access to the directory containing *newpath* is denied, or search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*. (See also *path_resolution*(7).)

EDQUOT
The user's quota of disk blocks on the filesystem has been exhausted.

EEXIST
newpath already exists.

EFAULT
oldpath or *newpath* points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *oldpath* or *newpath*.

EMLINK
The file referred to by *oldpath* already has the maximum number of links to it. For example, on an *ext4*(5) filesystem that does not employ the *dir_index* feature, the limit on the number of hard links to a file is 65,000; on *btrfs*(5), the limit is 65,535 links.

ENAMETOOLONG
oldpath or *newpath* was too long.

ENOENT
A directory component in *oldpath* or *newpath* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing the file has no room for the new directory entry.

ENOTDIR
A component used as a directory in *oldpath* or *newpath* is not, in fact, a directory.

EPERM
oldpath is a directory.

EPERM
The filesystem containing *oldpath* and *newpath* does not support the creation of hard links.

EPERM (since Linux 3.6)
The caller does not have permission to create a hard link to this file (see the description of */proc/sys/fs/protected_hardlinks* in *proc*(5)).

EPERM
oldpath is marked immutable or append-only. (See *ioctl_iflags*(2).)

EROFS
The file is on a read-only filesystem.

EXDEV
oldpath and *newpath* are not on the same mounted filesystem. (Linux permits a filesystem to be mounted at multiple points, but **link**() does not work across different mounts, even if the same filesystem is mounted on both.)

The following additional errors can occur for **linkat**():

EBADF
oldpath (*newpath*) is relative but *olddirfd* (*newdirfd*) is neither **AT_FDCWD** nor a valid file descriptor.

EINVAL
An invalid flag value was specified in *flags*.

ENOENT
AT_EMPTY_PATH was specified in *flags*, but the caller did not have the **CAP_DAC_READ_SEARCH** capability.

ENOENT
An attempt was made to link to the */proc/self/fd/NN* file corresponding to a file descriptor created with

```
open(path, O_TMPFILE | O_EXCL, mode);
```

See *open*(2).

ENOENT
An attempt was made to link to a */proc/self/fd/NN* file corresponding to a file that has been deleted.

ENOENT
oldpath is a relative pathname and *olddirfd* refers to a directory that has been deleted, or *newpath* is a relative pathname and *newdirfd* refers to a directory that has been deleted.

ENOTDIR
oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

EPERM
AT_EMPTY_PATH was specified in *flags*, *oldpath* is an empty string, and *olddirfd* refers to a directory.

VERSIONS

POSIX.1-2001 says that **link**() should dereference *oldpath* if it is a symbolic link. However, since Linux 2.0, Linux does not do so: if *oldpath* is a symbolic link, then *newpath* is created as a (hard) link to the same symbolic link file (i.e., *newpath* becomes a symbolic link to the same file that *oldpath* refers to). Some other implementations behave in the same manner as Linux. POSIX.1-2008 changes the specification of **link**(), making it implementation-dependent whether or not *oldpath* is dereferenced if it is a symbolic link. For precise control over the treatment of symbolic links when creating a link, use **linkat**().

glibc
On older kernels where **linkat**() is unavailable, the glibc wrapper function falls back to the use of **link**(), unless the **AT_SYMLINK_FOLLOW** is specified. When *oldpath* and *newpath* are relative pathnames, glibc constructs pathnames based on the symbolic links in */proc/self/fd* that correspond to the *olddirfd* and *newdirfd* arguments.

STANDARDS

link() POSIX.1-2008.

HISTORY

link() SVr4, 4.3BSD, POSIX.1-2001 (but see VERSIONS).

linkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES

Hard links, as created by **link**(), cannot span filesystems. Use *symlink*(2) if this is required.

BUGS

On NFS filesystems, the return code may be wrong in case the NFS server performs the link creation and dies before it can say so. Use *stat*(2) to find out if the link got created.

SEE ALSO

ln(1), *open*(2), *rename*(2), *stat*(2), *symlink*(2), *unlink*(2), *path_resolution*(7), *symlink*(7)

NAME

listen − listen for connections on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

int listen(int *sockfd***, int** *backlog***);**

DESCRIPTION

listen() marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using *accept*(2).

The *sockfd* argument is a file descriptor that refers to a socket of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EADDRINUSE
Another socket is already listening on the same port.

EADDRINUSE
(Internet domain sockets) The socket referred to by *sockfd* had not previously been bound to an address and, upon attempting to bind it to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of */proc/sys/net/ipv4/ip_local_port_range* in *ip*(7).

EBADF
The argument *sockfd* is not a valid file descriptor.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

EOPNOTSUPP
The socket is not of a type that supports the **listen**() operation.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

NOTES

To accept connections, the following steps are performed:

(1) A socket is created with *socket*(2).

(2) The socket is bound to a local address using *bind*(2), so that other sockets may be *connect*(2)ed to it.

(3) A willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**().

(4) Connections are accepted with *accept*(2).

The behavior of the *backlog* argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for *completely* established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using */proc/sys/net/ipv4/tcp_max_syn_backlog*. When syncookies are enabled there is no logical maximum length and this setting is ignored. See *tcp*(7) for more information.

If the *backlog* argument is greater than the value in */proc/sys/net/core/somaxconn*, then it is silently

capped to that value. Since Linux 5.4, the default in this file is 4096; in earlier kernels, the default value is 128. Before Linux 2.4.25, this limit was a hard coded value, **SOMAXCONN**, with the value 128.

EXAMPLES

See *bind*(2).

SEE ALSO

accept(2), *bind*(2), *connect*(2), *socket*(2), *socket*(7)

NAME

listxattr, llistxattr, flistxattr − list extended attribute names

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/xattr.h>

ssize_t listxattr(const char *path, char *_Nullable list, size_t size);
ssize_t llistxattr(const char *path, char *_Nullable list, size_t size);
ssize_t flistxattr(int fd, char *_Nullable list, size_t size);
```

DESCRIPTION

Extended attributes are *name*:*value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the *stat*(2) data). A complete overview of extended attributes concepts can be found in *xattr*(7).

listxattr() retrieves the list of extended attribute names associated with the given *path* in the filesystem. The retrieved list is placed in *list*, a caller-allocated buffer whose size (in bytes) is specified in the argument *size*. The list is the set of (null-terminated) names, one after the other. Names of extended attributes to which the calling process does not have access may be omitted from the list. The length of the attribute name *list* is returned.

llistxattr() is identical to **listxattr**(), except in the case of a symbolic link, where the list of names of extended attributes associated with the link itself is retrieved, not the file that it refers to.

flistxattr() is identical to **listxattr**(), only the open file referred to by *fd* (as returned by *open*(2)) is interrogated in place of *path*.

A single extended attribute *name* is a null-terminated string. The name includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode.

If *size* is specified as zero, these calls return the current size of the list of extended attribute names (and leave *list* unchanged). This can be used to determine the size of the buffer that should be supplied in a subsequent call. (But, bear in mind that there is a possibility that the set of extended attributes may change between the two calls, so that it is still necessary to check the return status from the second call.)

Example

The *list* of names is returned as an unordered array of null-terminated character strings (attribute names are separated by null bytes ('\0')), like this:

```
user.name1\0system.name1\0user.name2\0
```

Filesystems that implement POSIX ACLs using extended attributes might return a *list* like this:

```
system.posix_acl_access\0system.posix_acl_default\0
```

RETURN VALUE

On success, a nonnegative number is returned indicating the size of the extended attribute name list. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

E2BIG The size of the list of extended attribute names is larger than the maximum size allowed; the list cannot be retrieved. This can happen on filesystems that support an unlimited number of extended attributes per file such as XFS, for example. See BUGS.

ENOTSUP
Extended attributes are not supported by the filesystem, or are disabled.

ERANGE
The *size* of the *list* buffer is too small to hold the result.

In addition, the errors documented in *stat*(2) can also occur.

STANDARDS

Linux.

HISTORY

Linux 2.4, glibc 2.3.

BUGS

As noted in *xattr*(7), the VFS imposes a limit of 64 kB on the size of the extended attribute name list returned by **listxattr**(). If the total size of attribute names attached to a file exceeds this limit, it is no longer possible to retrieve the list of attribute names.

EXAMPLES

The following program demonstrates the usage of **listxattr**() and *getxattr*(2). For the file whose pathname is provided as a command-line argument, it lists all extended file attributes and their values.

To keep the code simple, the program assumes that attribute keys and values are constant during the execution of the program. A production program should expect and handle changes during execution of the program. For example, the number of bytes required for attribute keys might increase between the two calls to **listxattr**(). An application could handle this possibility using a loop that retries the call (perhaps up to a predetermined maximum number of attempts) with a larger buffer each time it fails with the error **ERANGE**. Calls to *getxattr*(2) could be handled similarly.

The following output was recorded by first creating a file, setting some extended file attributes, and then listing the attributes with the example program.

Example output

```
$ touch /tmp/foo
$ setfattr -n user.fred -v chocolate /tmp/foo
$ setfattr -n user.frieda -v bar /tmp/foo
$ setfattr -n user.empty /tmp/foo
$ ./listxattr /tmp/foo
user.fred: chocolate
user.frieda: bar
user.empty: <no value>
```

Program source (listxattr.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/xattr.h>

int
main(int argc, char *argv[])
{
    char     *buf, *key, *val;
    ssize_t  buflen, keylen, vallen;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /*
     * Determine the length of the buffer needed.
     */
    buflen = listxattr(argv[1], NULL, 0);
    if (buflen == -1) {
        perror("listxattr");
        exit(EXIT_FAILURE);
    }
    if (buflen == 0) {
        printf("%s has no attributes.\n", argv[1]);
        exit(EXIT_SUCCESS);
    }
```

```
/*
 * Allocate the buffer.
 */
buf = malloc(buflen);
if (buf == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/*
 * Copy the list of attribute keys to the buffer.
 */
buflen = listxattr(argv[1], buf, buflen);
if (buflen == -1) {
    perror("listxattr");
    exit(EXIT_FAILURE);
}

/*
 * Loop over the list of zero terminated strings with the
 * attribute keys. Use the remaining buffer length to determine
 * the end of the list.
 */
key = buf;
while (buflen > 0) {

    /*
     * Output attribute key.
     */
    printf("%s: ", key);

    /*
     * Determine length of the value.
     */
    vallen = getxattr(argv[1], key, NULL, 0);
    if (vallen == -1)
        perror("getxattr");

    if (vallen > 0) {

        /*
         * Allocate value buffer.
         * One extra byte is needed to append 0x00.
         */
        val = malloc(vallen + 1);
        if (val == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        /*
         * Copy value to buffer.
         */
        vallen = getxattr(argv[1], key, val, vallen);
        if (vallen == -1) {
            perror("getxattr");
        } else {
            /*
             * Output attribute value.
```

```
                     */
                    val[vallen] = 0;
                    printf("%s", val);
                }

                free(val);
            } else if (vallen == 0) {
                printf("<no value>");
            }

            printf("\n");

            /*
             * Forward to next attribute key.
             */
            keylen = strlen(key) + 1;
            buflen -= keylen;
            key += keylen;
        }

        free(buf);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getfattr(1), *setfattr*(1), *getxattr*(2), *open*(2), *removexattr*(2), *setxattr*(2), *stat*(2), *symlink*(7), *xattr*(7)

NAME

_llseek − reposition read/write file offset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS__llseek, unsigned int  fd, unsigned long offset_high,
        unsigned long offset_low, loff_t *result,
        unsigned int whence);
```

Note: glibc provides no wrapper for **_llseek**(), necessitating the use of *syscall*(2).

DESCRIPTION

Note: for information about the *llseek*(3) library function, see *lseek64*(3).

The **_llseek**() system call repositions the offset of the open file description associated with the file descriptor *fd* to the value

```
(offset_high << 32) | offset_low
```

This new offset is a byte offset relative to the beginning of the file, the current file offset, or the end of the file, depending on whether *whence* is **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, respectively.

The new file offset is returned in the argument *result*. The type *loff_t* is a 64-bit signed type.

This system call exists on various 32-bit platforms to support seeking to large file offsets.

RETURN VALUE

Upon successful completion, **_llseek**() returns 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not an open file descriptor.

EFAULT
Problem with copying results to user space.

EINVAL
whence is invalid.

VERSIONS

You probably want to use the *lseek*(2) wrapper function instead.

STANDARDS

Linux.

SEE ALSO

lseek(2), *open*(2), *lseek64*(3)

NAME

lookup_dcookie − return a directory entry's path

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_lookup_dcookie, uint64_t *cookie***, char *****buffer***,**
size_t *len***);**

Note: glibc provides no wrapper for **lookup_dcookie**(), necessitating the use of *syscall*(2).

DESCRIPTION

Look up the full path of the directory entry specified by the value *cookie*. The cookie is an opaque identifier uniquely identifying a particular directory entry. The buffer given is filled in with the full path of the directory entry.

For **lookup_dcookie**() to return successfully, the kernel must still hold a cookie reference to the directory entry.

RETURN VALUE

On success, **lookup_dcookie**() returns the length of the path string copied into the buffer. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
The buffer was not valid.

EINVAL
The kernel has no registered cookie/directory entry mappings at the time of lookup, or the cookie does not refer to a valid directory entry.

ENAMETOOLONG
The name could not fit in the buffer.

ENOMEM
The kernel could not allocate memory for the temporary buffer holding the path.

EPERM
The process does not have the capability **CAP_SYS_ADMIN** required to look up cookie values.

ERANGE
The buffer was not large enough to hold the path of the directory entry.

STANDARDS

Linux.

HISTORY

Linux 2.5.43.

The **ENAMETOOLONG** error was added in Linux 2.5.70.

NOTES

lookup_dcookie() is a special-purpose system call, currently used only by the *oprofile*(1) profiler. It relies on a kernel driver to register cookies for directory entries.

The path returned may be suffixed by the string " (deleted)" if the directory entry has been removed.

SEE ALSO

oprofile(1)

NAME

lseek − reposition read/write file offset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

off_t lseek(int *fd***, off_t** *offset***, int** *whence***);**

DESCRIPTION

lseek() repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET
The file offset is set to *offset* bytes.

SEEK_CUR
The file offset is set to its current location plus *offset* bytes.

SEEK_END
The file offset is set to the size of the file plus *offset* bytes.

lseek() allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

Seeking file data and holes

Since Linux 3.1, Linux supports the following additional values for *whence*:

SEEK_DATA
Adjust the file offset to the next location in the file greater than or equal to *offset* containing data. If *offset* points to data, then the file offset is set to *offset*.

SEEK_HOLE
Adjust the file offset to the next hole in the file greater than or equal to *offset*. If *offset* points into the middle of a hole, then the file offset is set to *offset*. If there is no hole past *offset*, then the file offset is adjusted to the end of the file (i.e., there is an implicit hole at the end of any file).

In both of the above cases, **lseek**() fails if *offset* points past the end of the file.

These operations allow applications to map holes in a sparsely allocated file. This can be useful for applications such as file backup tools, which can save space when creating backups and preserve holes, if they have a mechanism for discovering holes.

For the purposes of these operations, a hole is a sequence of zeros that (normally) has not been allocated in the underlying file storage. However, a filesystem is not obliged to report holes, so these operations are not a guaranteed mechanism for mapping the storage space actually allocated to a file. (Furthermore, a sequence of zeros that actually has been written to the underlying storage may not be reported as a hole.) In the simplest implementation, a filesystem can support the operations by making **SEEK_HOLE** always return the offset of the end of the file, and making **SEEK_DATA** always return *offset* (i.e., even if the location referred to by *offset* is a hole, it can be considered to consist of data that is a sequence of zeros).

The **_GNU_SOURCE** feature test macro must be defined in order to obtain the definitions of **SEEK_DATA** and **SEEK_HOLE** from *<unistd.h>*.

The **SEEK_HOLE** and **SEEK_DATA** operations are supported for the following filesystems:

- Btrfs (since Linux 3.1)
- OCFS (since Linux 3.2)
- XFS (since Linux 3.5)
- ext4 (since Linux 3.8)

- *tmpfs*(5) (since Linux 3.8)
- NFS (since Linux 3.18)
- FUSE (since Linux 4.5)
- GFS2 (since Linux 4.15)

RETURN VALUE

Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value *(off_t) −1* is returned and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not an open file descriptor.

EINVAL
whence is not valid. Or: the resulting file offset would be negative, or beyond the end of a seekable device.

ENXIO
whence is **SEEK_DATA** or **SEEK_HOLE**, and *offset* is beyond the end of the file, or *whence* is **SEEK_DATA** and *offset* is within a hole at the end of the file.

EOVERFLOW
The resulting file offset cannot be represented in an *off_t*.

ESPIPE
fd is associated with a pipe, socket, or FIFO.

VERSIONS

On Linux, using **lseek**() on a terminal device fails with the error **ESPIPE**.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEEK_DATA and **SEEK_HOLE** are nonstandard extensions also present in Solaris, FreeBSD, and DragonFly BSD; they are proposed for inclusion in the next POSIX revision (Issue 8).

NOTES

See *open*(2) for a discussion of the relationship between file descriptors, open file descriptions, and files.

If the **O_APPEND** file status flag is set on the open file description, then a *write*(2) *always* moves the file offset to the end of the file, regardless of the use of **lseek**().

Some devices are incapable of seeking and POSIX does not specify which devices must support **lseek**().

SEE ALSO

dup(2), *fallocate*(2), *fork*(2), *open*(2), *fseek*(3), *lseek64*(3), *posix_fallocate*(3)

NAME

madvise − give advice about use of memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

int madvise(void *addr***[.***length***], size_t** *length***, int** *advice***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

madvise():
Since glibc 2.19:
_DEFAULT_SOURCE
Up to and including glibc 2.19:
_BSD_SOURCE

DESCRIPTION

The **madvise**() system call is used to give advice or directions to the kernel about the address range beginning at address *addr* and with size *length*. **madvise**() only operates on whole pages, therefore *addr* must be page-aligned. The value of *length* is rounded up to a multiple of page size. In most cases, the goal of such advice is to improve system or application performance.

Initially, the system call supported a set of "conventional" *advice* values, which are also available on several other implementations. (Note, though, that **madvise**() is not specified in POSIX.) Subsequently, a number of Linux-specific *advice* values have been added.

Conventional advice values

The *advice* values listed below allow an application to tell the kernel how it expects to use some mapped or shared memory areas, so that the kernel can choose appropriate read-ahead and caching techniques. These *advice* values do not influence the semantics of the application (except in the case of **MADV_DONTNEED**), but may influence its performance. All of the *advice* values listed here have analogs in the POSIX-specified *posix_madvise*(3) function, and the values have the same meanings, with the exception of **MADV_DONTNEED**.

The advice is indicated in the *advice* argument, which is one of the following:

MADV_NORMAL
No special treatment. This is the default.

MADV_RANDOM
Expect page references in random order. (Hence, read ahead may be less useful than normally.)

MADV_SEQUENTIAL
Expect page references in sequential order. (Hence, pages in the given range can be aggressively read ahead, and may be freed soon after they are accessed.)

MADV_WILLNEED
Expect access in the near future. (Hence, it might be a good idea to read some pages ahead.)

MADV_DONTNEED
Do not expect access in the near future. (For the time being, the application is finished with the given range, so the kernel can free resources associated with it.)

After a successful **MADV_DONTNEED** operation, the semantics of memory access in the specified region are changed: subsequent accesses of pages in the range will succeed, but will result in either repopulating the memory contents from the up-to-date contents of the underlying mapped file (for shared file mappings, shared anonymous mappings, and shmem-based techniques such as System V shared memory segments) or zero-fill-on-demand pages for anonymous private mappings.

Note that, when applied to shared mappings, **MADV_DONTNEED** might not lead to immediate freeing of the pages in the range. The kernel is free to delay freeing the pages until an appropriate moment. The resident set size (RSS) of the calling process will be immediately reduced however.

MADV_DONTNEED cannot be applied to locked pages, or **VM_PFNMAP** pages. (Pages marked with the kernel-internal **VM_PFNMAP** flag are special memory areas that are not managed by the virtual memory subsystem. Such pages are typically created by device drivers that map the pages into user space.)

Support for Huge TLB pages was added in Linux v5.18. Addresses within a mapping backed by Huge TLB pages must be aligned to the underlying Huge TLB page size, and the range length is rounded up to a multiple of the underlying Huge TLB page size.

Linux-specific advice values

The following Linux-specific *advice* values have no counterparts in the POSIX-specified *posix_madvise*(3), and may or may not have counterparts in the **madvise**() interface available on other implementations. Note that some of these operations change the semantics of memory accesses.

MADV_REMOVE (since Linux 2.6.16)

Free up a given range of pages and its associated backing store. This is equivalent to punching a hole in the corresponding range of the backing store (see *fallocate*(2)). Subsequent accesses in the specified address range will see data with a value of zero.

The specified address range must be mapped shared and writable. This flag cannot be applied to locked pages, or **VM_PFNMAP** pages.

In the initial implementation, only *tmpfs*(5) supported **MADV_REMOVE**; but since Linux 3.5, any filesystem which supports the *fallocate*(2) **FALLOC_FL_PUNCH_HOLE** mode also supports **MADV_REMOVE**. Filesystems which do not support **MADV_REMOVE** fail with the error **EOPNOTSUPP**.

Support for the Huge TLB filesystem was added in Linux v4.3.

MADV_DONTFORK (since Linux 2.6.16)

Do not make the pages in this range available to the child after a *fork*(2). This is useful to prevent copy-on-write semantics from changing the physical location of a page if the parent writes to it after a *fork*(2). (Such page relocations cause problems for hardware that DMAs into the page.)

MADV_DOFORK (since Linux 2.6.16)

Undo the effect of **MADV_DONTFORK**, restoring the default behavior, whereby a mapping is inherited across *fork*(2).

MADV_HWPOISON (since Linux 2.6.32)

Poison the pages in the range specified by *addr* and *length* and handle subsequent references to those pages like a hardware memory corruption. This operation is available only for privileged (**CAP_SYS_ADMIN**) processes. This operation may result in the calling process receiving a **SIGBUS** and the page being unmapped.

This feature is intended for testing of memory error-handling code; it is available only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

MADV_MERGEABLE (since Linux 2.6.32)

Enable Kernel Samepage Merging (KSM) for the pages in the range specified by *addr* and *length*. The kernel regularly scans those areas of user memory that have been marked as mergeable, looking for pages with identical content. These are replaced by a single write-protected page (which is automatically copied if a process later wants to update the content of the page). KSM merges only private anonymous pages (see *mmap*(2)).

The KSM feature is intended for applications that generate many instances of the same data (e.g., virtualization systems such as KVM). It can consume a lot of processing power; use with care. See the Linux kernel source file *Documentation/admin−guide/mm/ksm.rst* for more details.

The **MADV_MERGEABLE** and **MADV_UNMERGEABLE** operations are available only if the kernel was configured with **CONFIG_KSM**.

MADV_UNMERGEABLE (since Linux 2.6.32)

Undo the effect of an earlier **MADV_MERGEABLE** operation on the specified address range; KSM unmerges whatever pages it had merged in the address range specified by *addr* and *length*.

MADV_SOFT_OFFLINE (since Linux 2.6.33)
Soft offline the pages in the range specified by *addr* and *length*. The memory of each page in the specified range is preserved (i.e., when next accessed, the same content will be visible, but in a new physical page frame), and the original page is offlined (i.e., no longer used, and taken out of normal memory management). The effect of the **MADV_SOFT_OFFLINE** operation is invisible to (i.e., does not change the semantics of) the calling process.

This feature is intended for testing of memory error-handling code; it is available only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

MADV_HUGEPAGE (since Linux 2.6.38)
Enable Transparent Huge Pages (THP) for pages in the range specified by *addr* and *length*. The kernel will regularly scan the areas marked as huge page candidates to replace them with huge pages. The kernel will also allocate huge pages directly when the region is naturally aligned to the huge page size (see *posix_memalign*(2)).

This feature is primarily aimed at applications that use large mappings of data and access large regions of that memory at a time (e.g., virtualization systems such as QEMU). It can very easily waste memory (e.g., a 2 MB mapping that only ever accesses 1 byte will result in 2 MB of wired memory instead of one 4 KB page). See the Linux kernel source file *Documentation/admin−guide/mm/transhuge.rst* for more details.

Most common kernels configurations provide **MADV_HUGEPAGE**-style behavior by default, and thus **MADV_HUGEPAGE** is normally not necessary. It is mostly intended for embedded systems, where **MADV_HUGEPAGE**-style behavior may not be enabled by default in the kernel. On such systems, this flag can be used in order to selectively enable THP. Whenever **MADV_HUGEPAGE** is used, it should always be in regions of memory with an access pattern that the developer knows in advance won't risk to increase the memory footprint of the application when transparent hugepages are enabled.

Since Linux 5.4, automatic scan of eligible areas and replacement by huge pages works with private anonymous pages (see *mmap*(2)), shmem pages, and file-backed pages. For all memory types, memory may only be replaced by huge pages on hugepage-aligned boundaries. For file-mapped memory —including tmpfs (see *tmpfs*(2))— the mapping must also be naturally hugepage-aligned within the file. Additionally, for file-backed, non-tmpfs memory, the file must not be open for write and the mapping must be executable.

The VMA must not be marked **VM_NOHUGEPAGE**, **VM_HUGETLB**, **VM_IO**, **VM_DONTEXPAND**, **VM_MIXEDMAP**, or **VM_PFNMAP**, nor can it be stack memory or backed by a DAX-enabled device (unless the DAX device is hot-plugged as System RAM). The process must also not have **PR_SET_THP_DISABLE** set (see *prctl*(2)).

The **MADV_HUGEPAGE**, **MADV_NOHUGEPAGE**, and **MADV_COLLAPSE** operations are available only if the kernel was configured with **CONFIG_TRANSPARENT_HUGEPAGE** and file/shmem memory is only supported if the kernel was configured with **CONFIG_READ_ONLY_THP_FOR_FS**.

MADV_NOHUGEPAGE (since Linux 2.6.38)
Ensures that memory in the address range specified by *addr* and *length* will not be backed by transparent hugepages.

MADV_COLLAPSE (since Linux 6.1)
Perform a best-effort synchronous collapse of the native pages mapped by the memory range into Transparent Huge Pages (THPs). **MADV_COLLAPSE** operates on the current state of memory of the calling process and makes no persistent changes or guarantees on how pages will be mapped, constructed, or faulted in the future.

MADV_COLLAPSE supports private anonymous pages (see *mmap*(2)), shmem pages, and file-backed pages. See **MADV_HUGEPAGE** for general information on memory requirements for THP. If the range provided spans multiple VMAs, the semantics of the collapse over each VMA is independent from the others. If collapse of a given huge page-aligned/sized region fails, the operation may continue to attempt collapsing the remainder of the specified memory. **MADV_COLLAPSE** will automatically clamp the provided range to be hugepage-aligned.

All non-resident pages covered by the range will first be swapped/faulted-in, before being copied onto a freshly allocated hugepage. If the native pages compose the same PTE-mapped hugepage, and are suitably aligned, allocation of a new hugepage may be elided and collapse may happen in-place. Unmapped pages will have their data directly initialized to 0 in the new hugepage. However, for every eligible hugepage-aligned/sized region to be collapsed, at least one page must currently be backed by physical memory.

MADV_COLLAPSE is independent of any sysfs (see *sysfs*(5)) setting under */sys/kernel/mm/transparent_hugepage*, both in terms of determining THP eligibility, and allocation semantics. See Linux kernel source file *Documentation/admin−guide/mm/transhuge.rst* for more information. **MADV_COLLAPSE** also ignores **huge=** tmpfs mount when operating on tmpfs files. Allocation for the new hugepage may enter direct reclaim and/or compaction, regardless of VMA flags (though **VM_NOHUGEPAGE** is still respected).

When the system has multiple NUMA nodes, the hugepage will be allocated from the node providing the most native pages.

If all hugepage-sized/aligned regions covered by the provided range were either successfully collapsed, or were already PMD-mapped THPs, this operation will be deemed successful. Note that this doesn't guarantee anything about other possible mappings of the memory. In the event multiple hugepage-aligned/sized areas fail to collapse, only the most-recently–failed code will be set in *errno*.

MADV_DONTDUMP (since Linux 3.4)
Exclude from a core dump those pages in the range specified by *addr* and *length*. This is useful in applications that have large areas of memory that are known not to be useful in a core dump. The effect of **MADV_DONTDUMP** takes precedence over the bit mask that is set via the */proc/*pid*/coredump_filter* file (see *core*(5)).

MADV_DODUMP (since Linux 3.4)
Undo the effect of an earlier **MADV_DONTDUMP**.

MADV_FREE (since Linux 4.5)
The application no longer requires the pages in the range specified by *addr* and *len*. The kernel can thus free these pages, but the freeing could be delayed until memory pressure occurs. For each of the pages that has been marked to be freed but has not yet been freed, the free operation will be canceled if the caller writes into the page. After a successful **MADV_FREE** operation, any stale data (i.e., dirty, unwritten pages) will be lost when the kernel frees the pages. However, subsequent writes to pages in the range will succeed and then kernel cannot free those dirtied pages, so that the caller can always see just written data. If there is no subsequent write, the kernel can free the pages at any time. Once pages in the range have been freed, the caller will see zero-fill-on-demand pages upon subsequent page references.

The **MADV_FREE** operation can be applied only to private anonymous pages (see *mmap*(2)). Before Linux 4.12, when freeing pages on a swapless system, the pages in the given range are freed instantly, regardless of memory pressure.

MADV_WIPEONFORK (since Linux 4.14)
Present the child process with zero-filled memory in this range after a *fork*(2). This is useful in forking servers in order to ensure that sensitive per-process data (for example, PRNG seeds, cryptographic secrets, and so on) is not handed to child processes.

The **MADV_WIPEONFORK** operation can be applied only to private anonymous pages (see *mmap*(2)).

Within the child created by *fork*(2), the **MADV_WIPEONFORK** setting remains in place on the specified address range. This setting is cleared during *execve*(2).

MADV_KEEPONFORK (since Linux 4.14)
Undo the effect of an earlier **MADV_WIPEONFORK**.

MADV_COLD (since Linux 5.4)
Deactivate a given range of pages. This will make the pages a more probable reclaim target should there be a memory pressure. This is a nondestructive operation. The advice might be ignored for some pages in the range when it is not applicable.

MADV_PAGEOUT (since Linux 5.4)
Reclaim a given range of pages. This is done to free up memory occupied by these pages. If a page is anonymous, it will be swapped out. If a page is file-backed and dirty, it will be written back to the backing storage. The advice might be ignored for some pages in the range when it is not applicable.

MADV_POPULATE_READ (since Linux 5.14)
"Populate (prefault) page tables readable, faulting in all pages in the range just as if manually reading from each page; however, avoid the actual memory access that would have been performed after handling the fault.

In contrast to **MAP_POPULATE**, **MADV_POPULATE_READ** does not hide errors, can be applied to (parts of) existing mappings and will always populate (prefault) page tables readable. One example use case is prefaulting a file mapping, reading all file content from disk; however, pages won't be dirtied and consequently won't have to be written back to disk when evicting the pages from memory.

Depending on the underlying mapping, map the shared zeropage, preallocate memory or read the underlying file; files with holes might or might not preallocate blocks. If populating fails, a **SIGBUS** signal is not generated; instead, an error is returned.

If **MADV_POPULATE_READ** succeeds, all page tables have been populated (prefaulted) readable once. If **MADV_POPULATE_READ** fails, some page tables might have been populated.

MADV_POPULATE_READ cannot be applied to mappings without read permissions and special mappings, for example, mappings marked with kernel-internal flags such as **VM_PFNMAP** or **VM_IO**, or secret memory regions created using **memfd_secret(2)**.

Note that with **MADV_POPULATE_READ**, the process can be killed at any moment when the system runs out of memory.

MADV_POPULATE_WRITE (since Linux 5.14)
Populate (prefault) page tables writable, faulting in all pages in the range just as if manually writing to each each page; however, avoid the actual memory access that would have been performed after handling the fault.

In contrast to **MAP_POPULATE**, MADV_POPULATE_WRITE does not hide errors, can be applied to (parts of) existing mappings and will always populate (prefault) page tables writable. One example use case is preallocating memory, breaking any CoW (Copy on Write).

Depending on the underlying mapping, preallocate memory or read the underlying file; files with holes will preallocate blocks. If populating fails, a **SIGBUS** signal is not generated; instead, an error is returned.

If **MADV_POPULATE_WRITE** succeeds, all page tables have been populated (prefaulted) writable once. If **MADV_POPULATE_WRITE** fails, some page tables might have been populated.

MADV_POPULATE_WRITE cannot be applied to mappings without write permissions and special mappings, for example, mappings marked with kernel-internal flags such as **VM_PFNMAP** or **VM_IO**, or secret memory regions created using **memfd_secret(2)**.

Note that with **MADV_POPULATE_WRITE**, the process can be killed at any moment when the system runs out of memory.

RETURN VALUE
On success, **madvise**() returns zero. On error, it returns −1 and *errno* is set to indicate the error.

ERRORS
EACCES
advice is **MADV_REMOVE**, but the specified address range is not a shared writable mapping.

EAGAIN
A kernel resource was temporarily unavailable.

EBADF
The map exists, but the area maps something that isn't a file.

EBUSY
(for **MADV_COLLAPSE**) Could not charge hugepage to cgroup: cgroup limit exceeded.

EFAULT
advice is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because a **SIGBUS** would have been generated on actual memory access and the reason is not a HW poisoned page (HW poisoned pages can, for example, be created using the **MADV_HWPOISON** flag described elsewhere in this page).

EINVAL
addr is not page-aligned or *length* is negative.

EINVAL
advice is not a valid.

EINVAL
advice is **MADV_COLD** or **MADV_PAGEOUT** and the specified address range includes locked, Huge TLB pages, or **VM_PFNMAP** pages.

EINVAL
advice is **MADV_DONTNEED** or **MADV_REMOVE** and the specified address range includes locked, Huge TLB pages, or **VM_PFNMAP** pages.

EINVAL
advice is **MADV_MERGEABLE** or **MADV_UNMERGEABLE**, but the kernel was not configured with **CONFIG_KSM**.

EINVAL
advice is **MADV_FREE** or **MADV_WIPEONFORK** but the specified address range includes file, Huge TLB, **MAP_SHARED**, or **VM_PFNMAP** ranges.

EINVAL
advice is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, but the specified address range includes ranges with insufficient permissions or special mappings, for example, mappings marked with kernel-internal flags such a **VM_IO** or **VM_PFNMAP**, or secret memory regions created using **memfd_secret(2)**.

EIO (for **MADV_WILLNEED**) Paging in this area would exceed the process's maximum resident set size.

ENOMEM
(for **MADV_WILLNEED**) Not enough memory: paging in failed.

ENOMEM
(for **MADV_COLLAPSE**) Not enough memory: could not allocate hugepage.

ENOMEM
Addresses in the specified range are not currently mapped, or are outside the address space of the process.

ENOMEM
advice is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because there was not enough memory.

EPERM
advice is **MADV_HWPOISON**, but the caller does not have the **CAP_SYS_ADMIN** capability.

EHWPOISON
advice is **MADV_POPULATE_READ** or **MADV_POPULATE_WRITE**, and populating (prefaulting) page tables failed because a HW poisoned page (HW poisoned pages can, for example, be created using the **MADV_HWPOISON** flag described elsewhere in this page) was encountered.

VERSIONS

Versions of this system call, implementing a wide variety of *advice* values, exist on many other implementations. Other implementations typically implement at least the flags listed above under *Conventional advice flags*, albeit with some variation in semantics.

POSIX.1-2001 describes *posix_madvise*(3) with constants **POSIX_MADV_NORMAL**, **POSIX_MADV_RANDOM**, **POSIX_MADV_SEQUENTIAL**, **POSIX_MADV_WILLNEED**, and **POSIX_MADV_DONTNEED**, and so on, with behavior close to the similarly named flags listed above.

Linux

The Linux implementation requires that the address *addr* be page-aligned, and allows *length* to be zero. If there are some parts of the specified address range that are not mapped, the Linux version of **madvise**() ignores them and applies the call to the rest (but returns **ENOMEM** from the system call, as it should).

madvise(0, 0, advice) will return zero iff *advice* is supported by the kernel and can be relied on to probe for support.

STANDARDS

None.

HISTORY

First appeared in 4.4BSD.

Since Linux 3.18, support for this system call is optional, depending on the setting of the **CONFIG_ADVISE_SYSCALLS** configuration option.

SEE ALSO

getrlimit(2), *memfd_secret*(2), *mincore*(2), *mmap*(2), *mprotect*(2), *msync*(2), *munmap*(2), *prctl*(2), *process_madvise*(2), *posix_madvise*(3), *core*(5)

NAME

mbind − set memory policy for a memory range

LIBRARY

NUMA (Non-Uniform Memory Access) policy library (*libnuma*, *−lnuma*)

SYNOPSIS

#include <numaif.h>

long mbind(void *addr***[.***len***], unsigned long** *len***, int** *mode***,**
const unsigned long *nodemask***[(.***maxnode* **+ ULONG_WIDTH - 1)**
/ ULONG_WIDTH],
unsigned long *maxnode***, unsigned int** *flags***);**

DESCRIPTION

mbind() sets the NUMA memory policy, which consists of a policy mode and zero or more nodes, for the memory range starting with *addr* and continuing for *len* bytes. The memory policy defines from which node memory is allocated.

If the memory range specified by the *addr* and *len* arguments includes an "anonymous" region of memory—that is a region of memory created using the *mmap*(2) system call with the **MAP_ANONYMOUS**—or a memory-mapped file, mapped using the *mmap*(2) system call with the **MAP_PRIVATE** flag, pages will be allocated only according to the specified policy when the application writes (stores) to the page. For anonymous regions, an initial read access will use a shared page in the kernel containing all zeros. For a file mapped with **MAP_PRIVATE**, an initial read access will allocate pages according to the memory policy of the thread that causes the page to be allocated. This may not be the thread that called **mbind**().

The specified policy will be ignored for any **MAP_SHARED** mappings in the specified memory range. Rather the pages will be allocated according to the memory policy of the thread that caused the page to be allocated. Again, this may not be the thread that called **mbind**().

If the specified memory range includes a shared memory region created using the *shmget*(2) system call and attached using the *shmat*(2) system call, pages allocated for the anonymous or shared memory region will be allocated according to the policy specified, regardless of which process attached to the shared memory segment causes the allocation. If, however, the shared memory region was created with the **SHM_HUGETLB** flag, the huge pages will be allocated according to the policy specified only if the page allocation is caused by the process that calls **mbind**() for that region.

By default, **mbind**() has an effect only for new allocations; if the pages inside the range have been already touched before setting the policy, then the policy has no effect. This default behavior may be overridden by the **MPOL_MF_MOVE** and **MPOL_MF_MOVE_ALL** flags described below.

The *mode* argument must specify one of **MPOL_DEFAULT**, **MPOL_BIND**, **MPOL_INTERLEAVE**, **MPOL_PREFERRED**, or **MPOL_LOCAL** (which are described in detail below). All policy modes except **MPOL_DEFAULT** require the caller to specify the node or nodes to which the mode applies, via the *nodemask* argument.

The *mode* argument may also include an optional *mode flag*. The supported *mode flags* are:

MPOL_F_NUMA_BALANCING (since Linux 5.15)
When *mode* is **MPOL_BIND**, enable the kernel NUMA balancing for the task if it is supported by the kernel. If the flag isn't supported by the kernel, or is used with *mode* other than **MPOL_BIND**, −1 is returned and *errno* is set to **EINVAL**.

MPOL_F_STATIC_NODES (since Linux-2.6.26)
A nonempty *nodemask* specifies physical node IDs. Linux does not remap the *nodemask* when the thread moves to a different cpuset context, nor when the set of nodes allowed by the thread's current cpuset context changes.

MPOL_F_RELATIVE_NODES (since Linux-2.6.26)
A nonempty *nodemask* specifies node IDs that are relative to the set of node IDs allowed by the thread's current cpuset.

nodemask points to a bit mask of nodes containing up to *maxnode* bits. The bit mask size is rounded to the next multiple of *sizeof(unsigned long)*, but the kernel will use bits only up to *maxnode*. A NULL value of *nodemask* or a *maxnode* value of zero specifies the empty set of nodes. If the value of

maxnode is zero, the *nodemask* argument is ignored. Where a *nodemask* is required, it must contain at least one node that is on-line, allowed by the thread's current cpuset context (unless the **MPOL_F_STATIC_NODES** mode flag is specified), and contains memory.

The *mode* argument must include one of the following values:

MPOL_DEFAULT
: This mode requests that any nondefault policy be removed, restoring default behavior. When applied to a range of memory via **mbind**(), this means to use the thread memory policy, which may have been set with *set_mempolicy*(2). If the mode of the thread memory policy is also **MPOL_DEFAULT**, the system-wide default policy will be used. The system-wide default policy allocates pages on the node of the CPU that triggers the allocation. For **MPOL_DEFAULT**, the *nodemask* and *maxnode* arguments must be specify the empty set of nodes.

MPOL_BIND
: This mode specifies a strict policy that restricts memory allocation to the nodes specified in *nodemask*. If *nodemask* specifies more than one node, page allocations will come from the node with sufficient free memory that is closest to the node where the allocation takes place. Pages will not be allocated from any node not specified in the IR nodemask . (Before Linux 2.6.26, page allocations came from the node with the lowest numeric node ID first, until that node contained no free memory. Allocations then came from the node with the next highest node ID specified in *nodemask* and so forth, until none of the specified nodes contained free memory.)

MPOL_INTERLEAVE
: This mode specifies that page allocations be interleaved across the set of nodes specified in *nodemask*. This optimizes for bandwidth instead of latency by spreading out pages and memory accesses to those pages across multiple nodes. To be effective the memory area should be fairly large, at least 1 MB or bigger with a fairly uniform access pattern. Accesses to a single page of the area will still be limited to the memory bandwidth of a single node.

MPOL_PREFERRED
: This mode sets the preferred node for allocation. The kernel will try to allocate pages from this node first and fall back to other nodes if the preferred nodes is low on free memory. If *nodemask* specifies more than one node ID, the first node in the mask will be selected as the preferred node. If the *nodemask* and *maxnode* arguments specify the empty set, then the memory is allocated on the node of the CPU that triggered the allocation.

MPOL_LOCAL (since Linux 3.8)
: This mode specifies "local allocation"; the memory is allocated on the node of the CPU that triggered the allocation (the "local node"). The *nodemask* and *maxnode* arguments must specify the empty set. If the "local node" is low on free memory, the kernel will try to allocate memory from other nodes. The kernel will allocate memory from the "local node" whenever memory for this node is available. If the "local node" is not allowed by the thread's current cpuset context, the kernel will try to allocate memory from other nodes. The kernel will allocate memory from the "local node" whenever it becomes allowed by the thread's current cpuset context. By contrast, **MPOL_DEFAULT** reverts to the memory policy of the thread (which may be set via *set_mempolicy*(2)); that policy may be something other than "local allocation".

If **MPOL_MF_STRICT** is passed in *flags* and *mode* is not **MPOL_DEFAULT**, then the call fails with the error **EIO** if the existing pages in the memory range don't follow the policy.

If **MPOL_MF_MOVE** is specified in *flags*, then the kernel will attempt to move all the existing pages in the memory range so that they follow the policy. Pages that are shared with other processes will not be moved. If **MPOL_MF_STRICT** is also specified, then the call fails with the error **EIO** if some pages could not be moved. If the **MPOL_INTERLEAVE** policy was specified, pages already residing on the specified nodes will not be moved such that they are interleaved.

If **MPOL_MF_MOVE_ALL** is passed in *flags*, then the kernel will attempt to move all existing pages in the memory range regardless of whether other processes use the pages. The calling thread must be privileged (**CAP_SYS_NICE**) to use this flag. If **MPOL_MF_STRICT** is also specified, then the call fails with the error **EIO** if some pages could not be moved. If the **MPOL_INTERLEAVE** policy was specified, pages already residing on the specified nodes will not be moved such that they are

interleaved.

RETURN VALUE

On success, **mbind**() returns 0; on error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
: Part or all of the memory range specified by *nodemask* and *maxnode* points outside your accessible address space. Or, there was an unmapped hole in the specified memory range specified by *addr* and *len*.

EINVAL
: An invalid value was specified for *flags* or *mode*; or *addr* + *len* was less than *addr*; or *addr* is not a multiple of the system page size. Or, *mode* is **MPOL_DEFAULT** and *nodemask* specified a nonempty set; or *mode* is **MPOL_BIND** or **MPOL_INTERLEAVE** and *nodemask* is empty. Or, *maxnode* exceeds a kernel-imposed limit. Or, *nodemask* specifies one or more node IDs that are greater than the maximum supported node ID. Or, none of the node IDs specified by *nodemask* are on-line and allowed by the thread's current cpuset context, or none of the specified nodes contain memory. Or, the *mode* argument specified both **MPOL_F_STATIC_NODES** and **MPOL_F_RELATIVE_NODES**.

EIO
: **MPOL_MF_STRICT** was specified and an existing page was already on a node that does not follow the policy; or **MPOL_MF_MOVE** or **MPOL_MF_MOVE_ALL** was specified and the kernel was unable to move all existing pages in the range.

ENOMEM
: Insufficient kernel memory was available.

EPERM
: The *flags* argument included the **MPOL_MF_MOVE_ALL** flag and the caller does not have the **CAP_SYS_NICE** privilege.

STANDARDS

Linux.

HISTORY

Linux 2.6.7.

Support for huge page policy was added with Linux 2.6.16. For interleave policy to be effective on huge page mappings the policied memory needs to be tens of megabytes or larger.

Before Linux 5.7. **MPOL_MF_STRICT** was ignored on huge page mappings.

MPOL_MF_MOVE and **MPOL_MF_MOVE_ALL** are available only on Linux 2.6.16 and later.

NOTES

For information on library support, see *numa*(7).

NUMA policy is not supported on a memory-mapped file range that was mapped with the **MAP_SHARED** flag.

The **MPOL_DEFAULT** mode can have different effects for **mbind**() and *set_mempolicy*(2). When **MPOL_DEFAULT** is specified for *set_mempolicy*(2), the thread's memory policy reverts to the system default policy or local allocation. When **MPOL_DEFAULT** is specified for a range of memory using **mbind**(), any pages subsequently allocated for that range will use the thread's memory policy, as set by *set_mempolicy*(2). This effectively removes the explicit policy from the specified range, "falling back" to a possibly nondefault policy. To select explicit "local allocation" for a memory range, specify a *mode* of **MPOL_LOCAL** or **MPOL_PREFERRED** with an empty set of nodes. This method will work for *set_mempolicy*(2), as well.

SEE ALSO

get_mempolicy(2), *getcpu*(2), *mmap*(2), *set_mempolicy*(2), *shmat*(2), *shmget*(2), *numa*(3), *cpuset*(7), *numa*(7), *numactl*(8)

NAME

membarrier − issue memory barriers on a set of threads

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/membarrier.h> /* Definition of **MEMBARRIER_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_membarrier, int *cmd***, unsigned int** *flags***, int** *cpu_id***);**

Note: glibc provides no wrapper for **membarrier**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **membarrier**() system call helps reducing the overhead of the memory barrier instructions required to order memory accesses on multi-core systems. However, this system call is heavier than a memory barrier, so using it effectively is *not* as simple as replacing memory barriers with this system call, but requires understanding of the details below.

Use of memory barriers needs to be done taking into account that a memory barrier always needs to be either matched with its memory barrier counterparts, or that the architecture's memory model doesn't require the matching barriers.

There are cases where one side of the matching barriers (which we will refer to as "fast side") is executed much more often than the other (which we will refer to as "slow side"). This is a prime target for the use of **membarrier**(). The key idea is to replace, for these matching barriers, the fast-side memory barriers by simple compiler barriers, for example:

```
asm volatile ("" : : : "memory")
```

and replace the slow-side memory barriers by calls to **membarrier**().

This will add overhead to the slow side, and remove overhead from the fast side, thus resulting in an overall performance increase as long as the slow side is infrequent enough that the overhead of the **membarrier**() calls does not outweigh the performance gain on the fast side.

The *cmd* argument is one of the following:

MEMBARRIER_CMD_QUERY (since Linux 4.3)
: Query the set of supported commands. The return value of the call is a bit mask of supported commands. **MEMBARRIER_CMD_QUERY**, which has the value 0, is not itself included in this bit mask. This command is always supported (on kernels where **membarrier**() is provided).

MEMBARRIER_CMD_GLOBAL (since Linux 4.16)
: Ensure that all threads from all processes on the system pass through a state where all memory accesses to user-space addresses match program order between entry to and return from the **membarrier**() system call. All threads on the system are targeted by this command.

MEMBARRIER_CMD_GLOBAL_EXPEDITED (since Linux 4.16)
: Execute a memory barrier on all running threads of all processes that previously registered with **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

  Upon return from the system call, the calling thread has a guarantee that all running threads have passed through a state where all memory accesses to user-space addresses match program order between entry to and return from the system call (non-running threads are de facto in such a state). This guarantee is provided only for the threads of processes that previously registered with **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

  Given that registration is about the intent to receive the barriers, it is valid to invoke **MEMBARRIER_CMD_GLOBAL_EXPEDITED** from a process that has not employed **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

  The "expedited" commands complete faster than the non-expedited ones; they never block, but have the downside of causing extra overhead.

MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED (since Linux 4.16)
Register the process's intent to receive **MEMBARRIER_CMD_GLOBAL_EXPEDITED** memory barriers.

MEMBARRIER_CMD_PRIVATE_EXPEDITED (since Linux 4.14)
Execute a memory barrier on each running thread belonging to the same process as the calling thread.

Upon return from the system call, the calling thread has a guarantee that all its running thread siblings have passed through a state where all memory accesses to user-space addresses match program order between entry to and return from the system call (non-running threads are de facto in such a state). This guarantee is provided only for threads in the same process as the calling thread.

The "expedited" commands complete faster than the non-expedited ones; they never block, but have the downside of causing extra overhead.

A process must register its intent to use the private expedited command prior to using it.

MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED (since Linux 4.14)
Register the process's intent to use **MEMBARRIER_CMD_PRIVATE_EXPEDITED**.

MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE (since Linux 4.16)
In addition to providing the memory ordering guarantees described in **MEMBARRIER_CMD_PRIVATE_EXPEDITED**, upon return from system call the calling thread has a guarantee that all its running thread siblings have executed a core serializing instruction. This guarantee is provided only for threads in the same process as the calling thread.

The "expedited" commands complete faster than the non-expedited ones, they never block, but have the downside of causing extra overhead.

A process must register its intent to use the private expedited sync core command prior to using it.

MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE (since Linux 4.16)
Register the process's intent to use **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE**.

MEMBARRIER_CMD_PRIVATE_EXPEDITED_RSEQ (since Linux 5.10)
Ensure the caller thread, upon return from system call, that all its running thread siblings have any currently running rseq critical sections restarted if *flags* parameter is 0; if *flags* parameter is **MEMBARRIER_CMD_FLAG_CPU**, then this operation is performed only on CPU indicated by *cpu_id*. This guarantee is provided only for threads in the same process as the calling thread.

RSEQ membarrier is only available in the "private expedited" form.

A process must register its intent to use the private expedited rseq command prior to using it.

MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_RSEQ (since Linux 5.10)
Register the process's intent to use **MEMBARRIER_CMD_PRIVATE_EXPEDITED_RSEQ**.

MEMBARRIER_CMD_SHARED (since Linux 4.3)
This is an alias for **MEMBARRIER_CMD_GLOBAL** that exists for header backward compatibility.

The *flags* argument must be specified as 0 unless the command is **MEMBARRIER_CMD_PRIVATE_EXPEDITED_RSEQ**, in which case *flags* can be either 0 or **MEMBARRIER_CMD_FLAG_CPU**.

The *cpu_id* argument is ignored unless *flags* is **MEMBARRIER_CMD_FLAG_CPU**, in which case it must specify the CPU targeted by this membarrier command.

All memory accesses performed in program order from each targeted thread are guaranteed to be ordered with respect to **membarrier**().

If we use the semantic *barrier()* to represent a compiler barrier forcing memory accesses to be performed in program order across the barrier, and *smp_mb()* to represent explicit memory barriers forcing

full memory ordering across the barrier, we have the following ordering table for each pairing of *barrier()*, **membarrier**(), and *smp_mb()*. The pair ordering is detailed as (O: ordered, X: not ordered):

	barrier()	smp_mb()	membarrier()
barrier()	X	X	O
smp_mb()	X	O	O
membarrier()	O	O	O

RETURN VALUE

On success, the **MEMBARRIER_CMD_QUERY** operation returns a bit mask of supported commands, and the **MEMBARRIER_CMD_GLOBAL**, **MEMBARRIER_CMD_GLOBAL_EXPEDITED**, **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**, **MEMBARRIER_CMD_PRIVATE_EXPEDITED**, **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED**, **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE**, and **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE** operations return zero. On error, −1 is returned, and *errno* is set to indicate the error.

For a given command, with *flags* set to 0, this system call is guaranteed to always return the same value until reboot. Further calls with the same arguments will lead to the same result. Therefore, with *flags* set to 0, error handling is required only for the first call to **membarrier**().

ERRORS

EINVAL

cmd is invalid, or *flags* is nonzero, or the **MEMBARRIER_CMD_GLOBAL** command is disabled because the *nohz_full* CPU parameter has been set, or the **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE** and **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE** commands are not implemented by the architecture.

ENOSYS

The **membarrier**() system call is not implemented by this kernel.

EPERM

The current process was not registered prior to using private expedited commands.

STANDARDS

Linux.

HISTORY

Linux 4.3.

Before Linux 5.10, the prototype was:

```
int membarrier(int cmd, int flags);
```

NOTES

A memory barrier instruction is part of the instruction set of architectures with weakly ordered memory models. It orders memory accesses prior to the barrier and after the barrier with respect to matching barriers on other cores. For instance, a load fence can order loads prior to and following that fence with respect to stores ordered by store fences.

Program order is the order in which instructions are ordered in the program assembly code.

Examples where **membarrier**() can be useful include implementations of Read-Copy-Update libraries and garbage collectors.

EXAMPLES

Assuming a multithreaded application where "fast_path()" is executed very frequently, and where "slow_path()" is executed infrequently, the following code (x86) can be transformed using **membarrier**():

```
#include <stdlib.h>

static volatile int a, b;

static void
fast_path(int *read_b)
{
```

```
    a = 1;
    asm volatile ("mfence" : : : "memory");
    *read_b = b;
}

static void
slow_path(int *read_a)
{
    b = 1;
    asm volatile ("mfence" : : : "memory");
    *read_a = a;
}

int
main(void)
{
    int read_a, read_b;

    /*
     * Real applications would call fast_path() and slow_path()
     * from different threads. Call those from main() to keep
     * this example short.
     */

    slow_path(&read_a);
    fast_path(&read_b);

    /*
     * read_b == 0 implies read_a == 1 and
     * read_a == 0 implies read_b == 1.
     */

    if (read_b == 0 && read_a == 0)
        abort();

    exit(EXIT_SUCCESS);
}
```

The code above transformed to use **membarrier**() becomes:

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/membarrier.h>

static volatile int a, b;

static int
membarrier(int cmd, unsigned int flags, int cpu_id)
{
    return syscall(__NR_membarrier, cmd, flags, cpu_id);
}

static int
init_membarrier(void)
{
    int ret;
```

```
    /* Check that membarrier() is supported. */

    ret = membarrier(MEMBARRIER_CMD_QUERY, 0, 0);
    if (ret < 0) {
        perror("membarrier");
        return -1;
    }

    if (!(ret & MEMBARRIER_CMD_GLOBAL)) {
        fprintf(stderr,
            "membarrier does not support MEMBARRIER_CMD_GLOBAL\n");
        return -1;
    }

    return 0;
}

static void
fast_path(int *read_b)
{
    a = 1;
    asm volatile ("" : : : "memory");
    *read_b = b;
}

static void
slow_path(int *read_a)
{
    b = 1;
    membarrier(MEMBARRIER_CMD_GLOBAL, 0, 0);
    *read_a = a;
}

int
main(int argc, char *argv[])
{
    int read_a, read_b;

    if (init_membarrier())
        exit(EXIT_FAILURE);

    /*
     * Real applications would call fast_path() and slow_path()
     * from different threads. Call those from main() to keep
     * this example short.
     */

    slow_path(&read_a);
    fast_path(&read_b);

    /*
     * read_b == 0 implies read_a == 1 and
     * read_a == 0 implies read_b == 1.
     */

    if (read_b == 0 && read_a == 0)
        abort();

    exit(EXIT_SUCCESS);
```

```
}
```

NAME

memfd_create − create an anonymous file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/mman.h>

int memfd_create(const char *name, unsigned int flags);
```

DESCRIPTION

memfd_create() creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, and so can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage. Once all references to the file are dropped, it is automatically released. Anonymous memory is used for all backing pages of the file. Therefore, files created by **memfd_create**() have the same semantics as other anonymous memory allocations such as those allocated using *mmap*(2) with the **MAP_ANONYMOUS** flag.

The initial size of the file is set to 0. Following the call, the file size should be set using *ftruncate*(2). (Alternatively, the file may be populated by calls to *write*(2) or similar.)

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory */proc/self/fd/*. The displayed name is always prefixed with *memfd:* and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

The following values may be bitwise ORed in *flags* to change the behavior of **memfd_create**():

MFD_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

MFD_ALLOW_SEALING
: Allow sealing operations on this file. See the discussion of the **F_ADD_SEALS** and **F_GET_SEALS** operations in *fcntl*(2), and also NOTES, below. The initial set of seals is empty. If this flag is not set, the initial set of seals will be **F_SEAL_SEAL**, meaning that no other seals can be set on the file.

MFD_HUGETLB (since Linux 4.14)
: The anonymous file will be created in the hugetlbfs filesystem using huge pages. See the Linux kernel source file *Documentation/admin−guide/mm/hugetlbpage.rst* for more information about hugetlbfs. Specifying both **MFD_HUGETLB** and **MFD_ALLOW_SEALING** in *flags* is supported since Linux 4.16.

MFD_HUGE_2MB
MFD_HUGE_1GB
...
: Used in conjunction with **MFD_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB, 1 GB, ...) on systems that support multiple hugetlb page sizes. Definitions for known huge page sizes are included in the header file <*linux/memfd.h*>.

  For details on encoding huge page sizes not included in the header file, see the discussion of the similarly named constants in *mmap*(2).

Unused bits in *flags* must be 0.

As its return value, **memfd_create**() returns a new file descriptor that can be used to refer to the file. This file descriptor is opened for both reading and writing (**O_RDWR**) and **O_LARGEFILE** is set for the file descriptor.

With respect to *fork*(2) and *execve*(2), the usual semantics apply for the file descriptor created by **memfd_create**(). A copy of the file descriptor is inherited by the child produced by *fork*(2) and refers to the same file. The file descriptor is preserved across *execve*(2), unless the close-on-exec flag has been set.

RETURN VALUE

On success, **memfd_create**() returns a new file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
The address in *name* points to invalid memory.

EINVAL
flags included unknown bits.

EINVAL
name was too long. (The limit is 249 bytes, excluding the terminating null byte.)

EINVAL
Both **MFD_HUGETLB** and **MFD_ALLOW_SEALING** were specified in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
There was insufficient memory to create a new anonymous file.

EPERM
The **MFD_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability) and is not a member of the *sysctl_hugetlb_shm_group* group; see the description of */proc/sys/vm/sysctl_hugetlb_shm_group* in *proc*(5).

STANDARDS

Linux.

HISTORY

Linux 3.17, glibc 2.27.

NOTES

The **memfd_create**() system call provides a simple alternative to manually mounting a *tmpfs*(5) filesystem and creating and opening a file in that filesystem. The primary purpose of **memfd_create**() is to create files and associated file descriptors that are used with the file-sealing APIs provided by *fcntl*(2).

The **memfd_create**() system call also has uses without file sealing (which is why file-sealing is disabled, unless explicitly requested with the **MFD_ALLOW_SEALING** flag). In particular, it can be used as an alternative to creating files in *tmp* or as an alternative to using the *open*(2) **O_TMPFILE** in cases where there is no intention to actually link the resulting file into the filesystem.

File sealing

In the absence of file sealing, processes that communicate via shared memory must either trust each other, or take measures to deal with the possibility that an untrusted peer may manipulate the shared memory region in problematic ways. For example, an untrusted peer might modify the contents of the shared memory at any time, or shrink the shared memory region. The former possibility leaves the local process vulnerable to time-of-check-to-time-of-use race conditions (typically dealt with by copying data from the shared memory region before checking and using it). The latter possibility leaves the local process vulnerable to **SIGBUS** signals when an attempt is made to access a now-nonexistent location in the shared memory region. (Dealing with this possibility necessitates the use of a handler for the **SIGBUS** signal.)

Dealing with untrusted peers imposes extra complexity on code that employs shared memory. Memory sealing enables that extra complexity to be eliminated, by allowing a process to operate secure in the knowledge that its peer can't modify the shared memory in an undesired fashion.

An example of the usage of the sealing mechanism is as follows:

(1) The first process creates a *tmpfs*(5) file using **memfd_create**(). The call yields a file descriptor used in subsequent steps.

(2) The first process sizes the file created in the previous step using *ftruncate*(2), maps it using *mmap*(2), and populates the shared memory with the desired data.

(3) The first process uses the *fcntl*(2) **F_ADD_SEALS** operation to place one or more seals on the file, in order to restrict further modifications on the file. (If placing the seal **F_SEAL_WRITE**, then it will be necessary to first unmap the shared writable mapping created in the previous step. Otherwise, behavior similar to **F_SEAL_WRITE** can be achieved by using **F_SEAL_FUTURE_WRITE**, which will prevent future writes via *mmap*(2) and *write*(2) from succeeding while keeping existing shared writable mappings).

(4) A second process obtains a file descriptor for the *tmpfs*(5) file and maps it. Among the possible ways in which this could happen are the following:

- The process that called **memfd_create**() could transfer the resulting file descriptor to the second process via a UNIX domain socket (see *unix*(7) and *cmsg*(3)). The second process then maps the file using *mmap*(2).
- The second process is created via *fork*(2) and thus automatically inherits the file descriptor and mapping. (Note that in this case and the next, there is a natural trust relationship between the two processes, since they are running under the same user ID. Therefore, file sealing would not normally be necessary.)
- The second process opens the file */proc/*pid*/fd/*fd, where *<pid>* is the PID of the first process (the one that called *memfd_create*()), and *<fd>* is the number of the file descriptor returned by the call to **memfd_create**() in that process. The second process then maps the file using *mmap*(2).

(5) The second process uses the *fcntl*(2) **F_GET_SEALS** operation to retrieve the bit mask of seals that has been applied to the file. This bit mask can be inspected in order to determine what kinds of restrictions have been placed on file modifications. If desired, the second process can apply further seals to impose additional restrictions (so long as the **F_SEAL_SEAL** seal has not yet been applied).

EXAMPLES

Below are shown two example programs that demonstrate the use of **memfd_create**() and the file sealing API.

The first program, *t_memfd_create.c*, creates a *tmpfs*(5) file using **memfd_create**(), sets a size for the file, maps it into memory, and optionally places some seals on the file. The program accepts up to three command-line arguments, of which the first two are required. The first argument is the name to associate with the file, the second argument is the size to be set for the file, and the optional third argument is a string of characters that specify seals to be set on the file.

The second program, *t_get_seals.c*, can be used to open an existing file that was created via **memfd_create**() and inspect the set of seals that have been applied to that file.

The following shell session demonstrates the use of these programs. First we create a *tmpfs*(5) file and set some seals on it:

```
$ ./t_memfd_create my_memfd_file 4096 sw &
[1] 11775
PID: 11775; fd: 3; /proc/11775/fd/3
```

At this point, the *t_memfd_create* program continues to run in the background. From another program, we can obtain a file descriptor for the file created by **memfd_create**() by opening the */proc/*pid*/fd* file that corresponds to the file descriptor opened by **memfd_create**(). Using that pathname, we inspect the content of the */proc/*pid*/fd* symbolic link, and use our *t_get_seals* program to view the seals that have been placed on the file:

```
$ readlink /proc/11775/fd/3
/memfd:my_memfd_file (deleted)
$ ./t_get_seals /proc/11775/fd/3
Existing seals: WRITE SHRINK
```

Program source: t_memfd_create.c

```
#define _GNU_SOURCE
```

```
#include <err.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int           fd;
    char          *name, *seals_arg;
    ssize_t       len;
    unsigned int  seals;

    if (argc < 3) {
        fprintf(stderr, "%s name size [seals]\n", argv[0]);
        fprintf(stderr, "\t'seals' can contain any of the "
                "following characters:\n");
        fprintf(stderr, "\t\tg - F_SEAL_GROW\n");
        fprintf(stderr, "\t\ts - F_SEAL_SHRINK\n");
        fprintf(stderr, "\t\tw - F_SEAL_WRITE\n");
        fprintf(stderr, "\t\tW - F_SEAL_FUTURE_WRITE\n");
        fprintf(stderr, "\t\tS - F_SEAL_SEAL\n");
        exit(EXIT_FAILURE);
    }

    name = argv[1];
    len = atoi(argv[2]);
    seals_arg = argv[3];

    /* Create an anonymous file in tmpfs; allow seals to be
       placed on the file. */

    fd = memfd_create(name, MFD_ALLOW_SEALING);
    if (fd == -1)
        err(EXIT_FAILURE, "memfd_create");

    /* Size the file as specified on the command line. */

    if (ftruncate(fd, len) == -1)
        err(EXIT_FAILURE, "truncate");

    printf("PID: %jd; fd: %d; /proc/%jd/fd/%d\n",
           (intmax_t) getpid(), fd, (intmax_t) getpid(), fd);

    /* Code to map the file and populate the mapping with data
       omitted. */

    /* If a 'seals' command-line argument was supplied, set some
       seals on the file. */

    if (seals_arg != NULL) {
        seals = 0;

        if (strchr(seals_arg, 'g') != NULL)
            seals |= F_SEAL_GROW;
```

```
            if (strchr(seals_arg, 's') != NULL)
                seals |= F_SEAL_SHRINK;
            if (strchr(seals_arg, 'w') != NULL)
                seals |= F_SEAL_WRITE;
            if (strchr(seals_arg, 'W') != NULL)
                seals |= F_SEAL_FUTURE_WRITE;
            if (strchr(seals_arg, 'S') != NULL)
                seals |= F_SEAL_SEAL;

            if (fcntl(fd, F_ADD_SEALS, seals) == -1)
                err(EXIT_FAILURE, "fcntl");
        }

        /* Keep running, so that the file created by memfd_create()
           continues to exist. */

        pause();

        exit(EXIT_SUCCESS);
    }
```

Program source: t_get_seals.c

```
    #define _GNU_SOURCE
    #include <err.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
    main(int argc, char *argv[])
    {
        int           fd;
        unsigned int  seals;

        if (argc != 2) {
            fprintf(stderr, "%s /proc/PID/fd/FD\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        fd = open(argv[1], O_RDWR);
        if (fd == -1)
            err(EXIT_FAILURE, "open");

        seals = fcntl(fd, F_GET_SEALS);
        if (seals == -1)
            err(EXIT_FAILURE, "fcntl");

        printf("Existing seals:");
        if (seals & F_SEAL_SEAL)
            printf(" SEAL");
        if (seals & F_SEAL_GROW)
            printf(" GROW");
        if (seals & F_SEAL_WRITE)
            printf(" WRITE");
        if (seals & F_SEAL_FUTURE_WRITE)
            printf(" FUTURE_WRITE");
        if (seals & F_SEAL_SHRINK)
            printf(" SHRINK");
```

```
        printf("\n");

        /* Code to map the file and access the contents of the
           resulting mapping omitted. */

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fcntl(2), *ftruncate*(2), *memfd_secret*(2), *mmap*(2), *shmget*(2), *shm_open*(3)

NAME

memfd_secret − create an anonymous RAM-based file to access secret memory regions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_memfd_secret, unsigned int  flags);
```

Note: glibc provides no wrapper for **memfd_secret**(), necessitating the use of *syscall*(2).

DESCRIPTION

memfd_secret() creates an anonymous RAM-based file and returns a file descriptor that refers to it. The file provides a way to create and access memory regions with stronger protection than usual RAM-based files and anonymous memory mappings. Once all open references to the file are closed, it is automatically released. The initial size of the file is set to 0. Following the call, the file size should be set using *ftruncate*(2).

The memory areas backing the file created with *memfd_secret*(2) are visible only to the processes that have access to the file descriptor. The memory region is removed from the kernel page tables and only the page tables of the processes holding the file descriptor map the corresponding physical memory. (Thus, the pages in the region can't be accessed by the kernel itself, so that, for example, pointers to the region can't be passed to system calls.)

The following values may be bitwise ORed in *flags* to control the behavior of **memfd_secret**():

FD_CLOEXEC
: Set the close-on-exec flag on the new file descriptor, which causes the region to be removed from the process on *execve*(2). See the description of the **O_CLOEXEC** flag in *open*(2)

As its return value, **memfd_secret**() returns a new file descriptor that refers to an anonymous file. This file descriptor is opened for both reading and writing (**O_RDWR**) and **O_LARGEFILE** is set for the file descriptor.

With respect to *fork*(2) and *execve*(2), the usual semantics apply for the file descriptor created by **memfd_secret**(). A copy of the file descriptor is inherited by the child produced by *fork*(2) and refers to the same file. The file descriptor is preserved across *execve*(2), unless the close-on-exec flag has been set.

The memory region is locked into memory in the same way as with *mlock*(2), so that it will never be written into swap, and hibernation is inhibited for as long as any **memfd_secret**() descriptions exist. However the implementation of **memfd_secret**() will not try to populate the whole range during the *mmap*(2) call that attaches the region into the process's address space; instead, the pages are only actually allocated as they are faulted in. The amount of memory allowed for memory mappings of the file descriptor obeys the same rules as *mlock*(2) and cannot exceed **RLIMIT_MEMLOCK**.

RETURN VALUE

On success, **memfd_secret**() returns a new file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
: *flags* included unknown bits.

EMFILE
: The per-process limit on the number of open file descriptors has been reached.

EMFILE
: The system-wide limit on the total number of open files has been reached.

ENOMEM
: There was insufficient memory to create a new anonymous file.

ENOSYS
: **memfd_secret**() is not implemented on this architecture, or has not been enabled on the kernel command-line with **secretmem_enable**=1.

STANDARDS

Linux.

HISTORY

Linux 5.14.

NOTES

The **memfd_secret**() system call is designed to allow a user-space process to create a range of memory that is inaccessible to anybody else - kernel included. There is no 100% guarantee that kernel won't be able to access memory ranges backed by **memfd_secret**() in any circumstances, but nevertheless, it is much harder to exfiltrate data from these regions.

memfd_secret() provides the following protections:

- Enhanced protection (in conjunction with all the other in-kernel attack prevention systems) against ROP attacks. Absence of any in-kernel primitive for accessing memory backed by **memfd_secret**() means that one-gadget ROP attack can't work to perform data exfiltration. The attacker would need to find enough ROP gadgets to reconstruct the missing page table entries, which significantly increases difficulty of the attack, especially when other protections like the kernel stack size limit and address space layout randomization are in place.
- Prevent cross-process user-space memory exposures. Once a region for a **memfd_secret**() memory mapping is allocated, the user can't accidentally pass it into the kernel to be transmitted somewhere. The memory pages in this region cannot be accessed via the direct map and they are disallowed in get_user_pages.
- Harden against exploited kernel flaws. In order to access memory areas backed by **memfd_secret**(), a kernel-side attack would need to either walk the page tables and create new ones, or spawn a new privileged user-space process to perform secrets exfiltration using *ptrace*(2).

The way **memfd_secret**() allocates and locks the memory may impact overall system performance, therefore the system call is disabled by default and only available if the system administrator turned it on using "secretmem.enable=y" kernel parameter.

To prevent potential data leaks of memory regions backed by **memfd_secret**() from a hybernation image, hybernation is prevented when there are active **memfd_secret**() users.

SEE ALSO

fcntl(2), *ftruncate*(2), *mlock*(2), *memfd_create*(2), *mmap*(2), *setrlimit*(2)

NAME

migrate_pages − move all pages in a process to another set of nodes

LIBRARY

NUMA (Non-Uniform Memory Access) policy library (*libnuma*, −*lnuma*)

SYNOPSIS

#include <numaif.h>

long migrate_pages(int *pid***, unsigned long** *maxnode*,
**const unsigned long ***_old_nodes_,
const unsigned long ***_new_nodes_);**

DESCRIPTION

migrate_pages() attempts to move all pages of the process *pid* that are in memory nodes *old_nodes* to the memory nodes in *new_nodes*. Pages not located in any node in *old_nodes* will not be migrated. As far as possible, the kernel maintains the relative topology relationship inside *old_nodes* during the migration to *new_nodes*.

The *old_nodes* and *new_nodes* arguments are pointers to bit masks of node numbers, with up to *maxnode* bits in each mask. These masks are maintained as arrays of unsigned *long* integers (in the last *long* integer, the bits beyond those specified by *maxnode* are ignored). The *maxnode* argument is the maximum node number in the bit mask plus one (this is the same as in *mbind*(2), but different from *select*(2)).

The *pid* argument is the ID of the process whose pages are to be moved. To move pages in another process, the caller must be privileged (**CAP_SYS_NICE**) or the real or effective user ID of the calling process must match the real or saved-set user ID of the target process. If *pid* is 0, then **migrate_pages**() moves pages of the calling process.

Pages shared with another process will be moved only if the initiating process has the **CAP_SYS_NICE** privilege.

RETURN VALUE

On success **migrate_pages**() returns the number of pages that could not be moved (i.e., a return of zero means that all pages were successfully moved). On error, it returns −1, and sets *errno* to indicate the error.

ERRORS

EFAULT
Part or all of the memory range specified by *old_nodes*/*new_nodes* and *maxnode* points outside your accessible address space.

EINVAL
The value specified by *maxnode* exceeds a kernel-imposed limit. Or, *old_nodes* or *new_nodes* specifies one or more node IDs that are greater than the maximum supported node ID. Or, none of the node IDs specified by *new_nodes* are on-line and allowed by the process's current cpuset context, or none of the specified nodes contain memory.

EPERM
Insufficient privilege (**CAP_SYS_NICE**) to move pages of the process specified by *pid*, or insufficient privilege (**CAP_SYS_NICE**) to access the specified target nodes.

ESRCH
No process matching *pid* could be found.

STANDARDS

Linux.

HISTORY

Linux 2.6.16.

NOTES

For information on library support, see *numa*(7).

Use *get_mempolicy*(2) with the **MPOL_F_MEMS_ALLOWED** flag to obtain the set of nodes that are allowed by the calling process's cpuset. Note that this information is subject to change at any time by manual or automatic reconfiguration of the cpuset.

Use of **migrate_pages**() may result in pages whose location (node) violates the memory policy established for the specified addresses (see *mbind*(2)) and/or the specified process (see *set_mempolicy*(2)). That is, memory policy does not constrain the destination nodes used by **migrate_pages**().

The <*numaif.h*> header is not included with glibc, but requires installing *libnuma−devel* or a similar package.

SEE ALSO

get_mempolicy(2), *mbind*(2), *set_mempolicy*(2), *numa*(3), *numa_maps*(5), *cpuset*(7), *numa*(7), *migratepages*(8), *numastat*(8)

Documentation/vm/page_migration.rst in the Linux kernel source tree

NAME

mincore − determine whether pages are resident in memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

int mincore(void *addr***[.***length***], size_t** *length***, unsigned char *** *vec***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mincore():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

mincore() returns a vector that indicates whether pages of the calling process's virtual memory are resident in core (RAM), and so will not cause a disk access (page fault) if referenced. The kernel returns residency information about the pages starting at the address *addr*, and continuing for *length* bytes.

The *addr* argument must be a multiple of the system page size. The *length* argument need not be a multiple of the page size, but since residency information is returned for whole pages, *length* is effectively rounded up to the next multiple of the page size. One may obtain the page size (**PAGE_SIZE**) using *sysconf(_SC_PAGESIZE)*.

The *vec* argument must point to an array containing at least *(length+PAGE_SIZE−1) / PAGE_SIZE* bytes. On return, the least significant bit of each byte will be set if the corresponding page is currently resident in memory, and be clear otherwise. (The settings of the other bits in each byte are undefined; these bits are reserved for possible later use.) Of course the information returned in *vec* is only a snapshot: pages that are not locked in memory can come and go at any moment, and the contents of *vec* may already be stale by the time this call returns.

RETURN VALUE

On success, **mincore**() returns zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN kernel is temporarily out of resources.

EFAULT
vec points to an invalid address.

EINVAL
addr is not a multiple of the page size.

ENOMEM
length is greater than (*TASK_SIZE − addr*). (This could occur if a negative value is specified for *length*, since that value will be interpreted as a large unsigned integer.) In Linux 2.6.11 and earlier, the error **EINVAL** was returned for this condition.

ENOMEM
addr to *addr* + *length* contained unmapped memory.

STANDARDS

None.

HISTORY

Linux 2.3.99pre1, glibc 2.2.

First appeared in 4.4BSD.

NetBSD, FreeBSD, OpenBSD, Solaris 8, AIX 5.1, SunOS 4.1.

BUGS

Before Linux 2.6.21, **mincore**() did not return correct information for **MAP_PRIVATE** mappings, or for nonlinear mappings (established using *remap_file_pages*(2)).

SEE ALSO

fincore(1), *madvise*(2), *mlock*(2), *mmap*(2), *posix_fadvise*(2), *posix_madvise*(3)

NAME

mkdir, mkdirat − create a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mkdirat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

mkdir() attempts to create a directory named *pathname*.

The argument *mode* specifies the mode for the new directory (see *inode*(7)). It is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created directory is (*mode* & ~*umask* & 0777). Whether other *mode* bits are honored for the created directory depends on the operating system. For Linux, see NOTES below.

The newly created directory will be owned by the effective user ID of the process. If the directory containing the file has the set-group-ID bit set, or if the filesystem is mounted with BSD group semantics (*mount −o bsdgroups* or, synonymously *mount −o grpid*), the new directory will inherit the group ownership from its parent; otherwise it will be owned by the effective group ID of the process.

If the parent directory has the set-group-ID bit set, then so will the newly created directory.

mkdirat()

The **mkdirat**() system call operates in exactly the same way as **mkdir**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **mkdir**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *mkdir*())

If *pathname* is absolute, then *dirfd* is ignored.

See *openat*(2) for an explanation of the need for **mkdirat**().

RETURN VALUE

mkdir() and **mkdirat**() return zero on success. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
The parent directory does not allow write permission to the process, or one of the directories in *pathname* did not allow search permission. (See also *path_resolution*(7).)

EBADF
(**mkdirat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EDQUOT
The user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST
pathname already exists (not necessarily as a directory). This includes the case where *pathname* is a symbolic link, dangling or not.

EFAULT
pathname points outside your accessible address space.

EINVAL
The final component ("basename") of the new directory's *pathname* is invalid (e.g., it contains characters not permitted by the underlying filesystem).

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

EMLINK
The number of links to the parent directory would exceed **LINK_MAX**.

ENAMETOOLONG
pathname was too long.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing *pathname* has no room for the new directory.

ENOSPC
The new directory cannot be created because the user's disk quota is exhausted.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR
(**mkdirat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM
The filesystem containing *pathname* does not support the creation of directories.

EROFS
pathname refers to a file on a read-only filesystem.

VERSIONS

Under Linux, apart from the permission bits, the **S_ISVTX** *mode* bit is also honored.

glibc notes

On older kernels where **mkdirat**() is unavailable, the glibc wrapper function falls back to the use of **mkdir**(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

STANDARDS

POSIX.1-2008.

HISTORY

mkdir()
SVr4, BSD, POSIX.1-2001.

mkdirat()
Linux 2.6.16, glibc 2.4.

NOTES

There are many infelicities in the protocol underlying NFS. Some of these affect **mkdir**().

SEE ALSO

mkdir(1), *chmod*(2), *chown*(2), *mknod*(2), *mount*(2), *rmdir*(2), *stat*(2), *umask*(2), *unlink*(2), *acl*(5), *path_resolution*(7)

NAME

mknod, mknodat − create a special or ordinary file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/stat.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int mknodat(int dirfd, const char *pathname, mode_t mode, dev_t dev);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mknod():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The system call **mknod**() creates a filesystem node (file, device special file, or named pipe) named *pathname*, with attributes specified by *mode* and *dev*.

The *mode* argument specifies both the file mode to use and the type of node to be created. It should be a combination (using bitwise OR) of one of the file types listed below and zero or more of the file mode bits listed in *inode*(7).

The file mode is modified by the process's *umask* in the usual way: in the absence of a default ACL, the permissions of the created node are (*mode* & *~umask*).

The file type must be one of **S_IFREG**, **S_IFCHR**, **S_IFBLK**, **S_IFIFO**, or **S_IFSOCK** to specify a regular file (which will be created empty), character special file, block special file, FIFO (named pipe), or UNIX domain socket, respectively. (Zero file type is equivalent to type **S_IFREG**.)

If the file type is **S_IFCHR** or **S_IFBLK**, then *dev* specifies the major and minor numbers of the newly created device special file (**makedev**(3) may be useful to build the value for *dev*); otherwise it is ignored.

If *pathname* already exists, or is a symbolic link, this call fails with an **EEXIST** error.

The newly created node will be owned by the effective user ID of the process. If the directory containing the node has the set-group-ID bit set, or if the filesystem is mounted with BSD group semantics, the new node will inherit the group ownership from its parent directory; otherwise it will be owned by the effective group ID of the process.

mknodat()

The **mknodat**() system call operates in exactly the same way as **mknod**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **mknod**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *mknod*())

If *pathname* is absolute, then *dirfd* is ignored.

See *openat*(2) for an explanation of the need for **mknodat**().

RETURN VALUE

mknod() and **mknodat**() return zero on success. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
The parent directory does not allow write permission to the process, or one of the directories in the path prefix of *pathname* did not allow search permission. (See also *path_resolution*(7).)

EBADF
(**mknodat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EDQUOT
The user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST
pathname already exists. This includes the case where *pathname* is a symbolic link, dangling or not.

EFAULT
pathname points outside your accessible address space.

EINVAL
mode requested creation of something other than a regular file, device special file, FIFO or socket.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
pathname was too long.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing *pathname* has no room for the new node.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR
(**mknodat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EPERM
mode requested creation of something other than a regular file, FIFO (named pipe), or UNIX domain socket, and the caller is not privileged (Linux: does not have the **CAP_MKNOD** capability); also returned if the filesystem containing *pathname* does not support the type of node requested.

EROFS
pathname refers to a file on a read-only filesystem.

VERSIONS

POSIX.1-2001 says: "The only portable use of **mknod**() is to create a FIFO-special file. If *mode* is not **S_IFIFO** or *dev* is not 0, the behavior of **mknod**() is unspecified." However, nowadays one should never use **mknod**() for this purpose; one should use *mkfifo*(3), a function especially defined for this purpose.

Under Linux, **mknod**() cannot be used to create directories. One should make directories with *mkdir*(2).

STANDARDS

POSIX.1-2008.

HISTORY

mknod()
SVr4, 4.4BSD, POSIX.1-2001 (but see VERSIONS).

mknodat()
Linux 2.6.16, glibc 2.4. POSIX.1-2008.

NOTES

There are many infelicities in the protocol underlying NFS. Some of these affect **mknod**() and **mknodat**().

SEE ALSO

mknod(1), *chmod*(2), *chown*(2), *fcntl*(2), *mkdir*(2), *mount*(2), *socket*(2), *stat*(2), *umask*(2), *unlink*(2), *makedev*(3), *mkfifo*(3), *acl*(5), *path_resolution*(7)

NAME

mlock, mlock2, munlock, mlockall, munlockall − lock and unlock memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/mman.h>

int mlock(const void addr[.len], size_t len);
int mlock2(const void addr[.len], size_t len, unsigned int flags);
int munlock(const void addr[.len], size_t len);

int mlockall(int flags);
int munlockall(void);
```

DESCRIPTION

mlock(), **mlock2**(), and **mlockall**() lock part or all of the calling process's virtual address space into RAM, preventing that memory from being paged to the swap area.

munlock() and **munlockall**() perform the converse operation, unlocking part or all of the calling process's virtual address space, so that pages in the specified virtual address range can be swapped out again if required by the kernel memory manager.

Memory locking and unlocking are performed in units of whole pages.

mlock(), mlock2(), and munlock()

mlock() locks pages in the address range starting at *addr* and continuing for *len* bytes. All pages that contain a part of the specified address range are guaranteed to be resident in RAM when the call returns successfully; the pages are guaranteed to stay in RAM until later unlocked.

mlock2() also locks pages in the specified range starting at *addr* and continuing for *len* bytes. However, the state of the pages contained in that range after the call returns successfully will depend on the value in the *flags* argument.

The *flags* argument can be either 0 or the following constant:

MLOCK_ONFAULT
: Lock pages that are currently resident and mark the entire range so that the remaining nonresident pages are locked when they are populated by a page fault.

If *flags* is 0, **mlock2**() behaves exactly the same as **mlock**().

munlock() unlocks pages in the address range starting at *addr* and continuing for *len* bytes. After this call, all pages that contain a part of the specified memory range can be moved to external swap space again by the kernel.

mlockall() and munlockall()

mlockall() locks all pages mapped into the address space of the calling process. This includes the pages of the code, data, and stack segment, as well as shared libraries, user space kernel data, shared memory, and memory-mapped files. All mapped pages are guaranteed to be resident in RAM when the call returns successfully; the pages are guaranteed to stay in RAM until later unlocked.

The *flags* argument is constructed as the bitwise OR of one or more of the following constants:

MCL_CURRENT
: Lock all pages which are currently mapped into the address space of the process.

MCL_FUTURE
: Lock all pages which will become mapped into the address space of the process in the future. These could be, for instance, new pages required by a growing heap and stack as well as new memory-mapped files or shared memory regions.

MCL_ONFAULT (since Linux 4.4)
: Used together with **MCL_CURRENT**, **MCL_FUTURE**, or both. Mark all current (with **MCL_CURRENT**) or future (with **MCL_FUTURE**) mappings to lock pages when they are faulted in. When used with **MCL_CURRENT**, all present pages are locked, but **mlockall**() will not fault in non-present pages. When used with **MCL_FUTURE**, all future mappings will be marked to lock pages when they are faulted in, but they will not be populated by the

lock when the mapping is created. **MCL_ONFAULT** must be used with either **MCL_CURRENT** or **MCL_FUTURE** or both.

If **MCL_FUTURE** has been specified, then a later system call (e.g., *mmap*(2), *sbrk*(2), *malloc*(3)), may fail if it would cause the number of locked bytes to exceed the permitted maximum (see below). In the same circumstances, stack growth may likewise fail: the kernel will deny stack expansion and deliver a **SIGSEGV** signal to the process.

munlockall() unlocks all pages mapped into the address space of the calling process.

RETURN VALUE

On success, these system calls return 0. On error, −1 is returned, *errno* is set to indicate the error, and no changes are made to any locks in the address space of the process.

ERRORS

EAGAIN
(**mlock**(), **mlock2**(), and *munlock*()) Some or all of the specified address range could not be locked.

EINVAL
(**mlock**(), **mlock2**(), and *munlock*()) The result of the addition *addr+len* was less than *addr* (e.g., the addition may have resulted in an overflow).

EINVAL
(**mlock2**()) Unknown *flags* were specified.

EINVAL
(**mlockall**()) Unknown *flags* were specified or **MCL_ONFAULT** was specified without either **MCL_FUTURE** or **MCL_CURRENT**.

EINVAL
(Not on Linux) *addr* was not a multiple of the page size.

ENOMEM
(**mlock**(), **mlock2**(), and *munlock*()) Some of the specified address range does not correspond to mapped pages in the address space of the process.

ENOMEM
(**mlock**(), **mlock2**(), and *munlock*()) Locking or unlocking a region would result in the total number of mappings with distinct attributes (e.g., locked versus unlocked) exceeding the allowed maximum. (For example, unlocking a range in the middle of a currently locked mapping would result in three mappings: two locked mappings at each end and an unlocked mapping in the middle.)

ENOMEM
(Linux 2.6.9 and later) the caller had a nonzero **RLIMIT_MEMLOCK** soft resource limit, but tried to lock more memory than the limit permitted. This limit is not enforced if the process is privileged (**CAP_IPC_LOCK**).

ENOMEM
(Linux 2.4 and earlier) the calling process tried to lock more than half of RAM.

EPERM
The caller is not privileged, but needs privilege (**CAP_IPC_LOCK**) to perform the requested operation.

EPERM
(**munlockall**()) (Linux 2.6.8 and earlier) The caller was not privileged (**CAP_IPC_LOCK**).

VERSIONS

Linux

Under Linux, **mlock**(), **mlock2**(), and **munlock**() automatically round *addr* down to the nearest page boundary. However, the POSIX.1 specification of **mlock**() and **munlock**() allows an implementation to require that *addr* is page aligned, so portable applications should ensure this.

The *VmLck* field of the Linux-specific */proc/*pid*/status* file shows how many kilobytes of memory the process with ID *PID* has locked using **mlock**(), **mlock2**(), **mlockall**(), and *mmap*(2) **MAP_LOCKED**.

STANDARDS

mlock()
munlock()
mlockall()
munlockall()
POSIX.1-2008.

mlock2()
Linux.

On POSIX systems on which **mlock**() and **munlock**() are available, **_POSIX_MEMLOCK_RANGE** is defined in <*unistd.h*> and the number of bytes in a page can be determined from the constant **PAGESIZE** (if defined) in <*limits.h*> or by calling *sysconf(_SC_PAGESIZE)*.

On POSIX systems on which **mlockall**() and **munlockall**() are available, **_POSIX_MEMLOCK** is defined in <*unistd.h*> to a value greater than 0. (See also *sysconf*(3).)

HISTORY

mlock()
munlock()
mlockall()
munlockall()
POSIX.1-2001, POSIX.1-2008, SVr4.

mlock2()
Linux 4.4, glibc 2.27.

NOTES

Memory locking has two main applications: real-time algorithms and high-security data processing. Real-time applications require deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with *sched_setscheduler*(2). Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets could be transferred onto a persistent swap store medium, where they might be accessible to the enemy long after the security software has erased the secrets in RAM and terminated. (But be aware that the suspend mode on laptops and some desktop computers will save a copy of the system's RAM to disk, regardless of memory locks.)

Real-time processes that are using **mlockall**() to prevent delays on page faults should reserve enough locked stack pages before entering the time-critical section, so that no page fault can be caused by function calls. This can be achieved by calling a function that allocates a sufficiently large automatic variable (an array) and writes to the memory occupied by this array in order to touch these stack pages. This way, enough pages will be mapped for the stack and can be locked into RAM. The dummy writes ensure that not even copy-on-write page faults can occur in the critical section.

Memory locks are not inherited by a child created via *fork*(2) and are automatically removed (unlocked) during an *execve*(2) or when the process terminates. The **mlockall**() **MCL_FUTURE** and **MCL_FUTURE | MCL_ONFAULT** settings are not inherited by a child created via *fork*(2) and are cleared during an *execve*(2).

Note that *fork*(2) will prepare the address space for a copy-on-write operation. The consequence is that any write access that follows will cause a page fault that in turn may cause high latencies for a real-time process. Therefore, it is crucial not to invoke *fork*(2) after an **mlockall**() or **mlock**() operation—not even from a thread which runs at a low priority within a process which also has a thread running at elevated priority.

The memory lock on an address range is automatically removed if the address range is unmapped via *munmap*(2).

Memory locks do not stack, that is, pages which have been locked several times by calls to **mlock**(), **mlock2**(), or **mlockall**() will be unlocked by a single call to **munlock**() for the corresponding range or by **munlockall**(). Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

If a call to **mlockall**() which uses the **MCL_FUTURE** flag is followed by another call that does not specify this flag, the changes made by the **MCL_FUTURE** call will be lost.

The **mlock2**() **MLOCK_ONFAULT** flag and the **mlockall**() **MCL_ONFAULT** flag allow efficient memory locking for applications that deal with large mappings where only a (small) portion of pages in the mapping are touched. In such cases, locking all of the pages in a mapping would incur a significant penalty for memory locking.

Limits and permissions

In Linux 2.6.8 and earlier, a process must be privileged (**CAP_IPC_LOCK**) in order to lock memory and the **RLIMIT_MEMLOCK** soft resource limit defines a limit on how much memory the process may lock.

Since Linux 2.6.9, no limits are placed on the amount of memory that a privileged process can lock and the **RLIMIT_MEMLOCK** soft resource limit instead defines a limit on how much memory an unprivileged process may lock.

BUGS

In Linux 4.8 and earlier, a bug in the kernel's accounting of locked memory for unprivileged processes (i.e., without **CAP_IPC_LOCK**) meant that if the region specified by *addr* and *len* overlapped an existing lock, then the already locked bytes in the overlapping region were counted twice when checking against the limit. Such double accounting could incorrectly calculate a "total locked memory" value for the process that exceeded the **RLIMIT_MEMLOCK** limit, with the result that **mlock**() and **mlock2**() would fail on requests that should have succeeded. This bug was fixed in Linux 4.9.

In Linux 2.4 series of kernels up to and including Linux 2.4.17, a bug caused the **mlockall**() **MCL_FUTURE** flag to be inherited across a *fork*(2). This was rectified in Linux 2.4.18.

Since Linux 2.6.9, if a privileged process calls *mlockall(MCL_FUTURE)* and later drops privileges (loses the **CAP_IPC_LOCK** capability by, for example, setting its effective UID to a nonzero value), then subsequent memory allocations (e.g., *mmap*(2), *brk*(2)) will fail if the **RLIMIT_MEMLOCK** resource limit is encountered.

SEE ALSO

mincore(2), *mmap*(2), *setrlimit*(2), *shmctl*(2), *sysconf*(3), *proc*(5), *capabilities*(7)

NAME

mmap, munmap − map or unmap files or devices into memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/mman.h>

void *mmap(void addr[.length], size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void addr[.length], size_t length);
```

See NOTES for information on feature test macro requirements.

DESCRIPTION

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the kernel will pick a nearby page boundary (but always above or equal to the value specified by */proc/sys/vm/mmap_min_addr*) and attempt to create the mapping there. If another mapping already exists there, the kernel picks a new address that may or may not depend on the hint. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

After the **mmap**() call has returned, the file descriptor, *fd*, can be closed immediately without invalidating the mapping.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

The flags argument

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED
: Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of *msync*(2).)

MAP_SHARED_VALIDATE (since Linux 4.15)
: This flag provides the same behavior as **MAP_SHARED** except that **MAP_SHARED** mappings ignore unknown flags in *flags*. By contrast, when creating a mapping using **MAP_SHARED_VALIDATE**, the kernel verifies all passed flags are known and fails the mapping with the error **EOPNOTSUPP** for unknown flags. This mapping type is also required to be able to use some mapping flags (e.g., **MAP_SYNC**).

MAP_PRIVATE
: Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped

region.

Both **MAP_SHARED** and **MAP_PRIVATE** are described in POSIX.1-2001 and POSIX.1-2008. **MAP_SHARED_VALIDATE** is a Linux extension.

In addition, zero or more of the following values can be ORed in *flags*:

MAP_32BIT (since Linux 2.4.20, 2.6)
Put the mapping into the first 2 Gigabytes of the process address space. This flag is supported only on x86-64, for 64-bit programs. It was added to allow thread stacks to be allocated somewhere in the first 2 GB of memory, so as to improve context-switch performance on some early 64-bit processors. Modern x86-64 processors no longer have this performance problem, so use of this flag is not required on those systems. The **MAP_32BIT** flag is ignored when **MAP_FIXED** is set.

MAP_ANON
Synonym for **MAP_ANONYMOUS**; provided for compatibility with other implementations.

MAP_ANONYMOUS
The mapping is not backed by any file; its contents are initialized to zero. The *fd* argument is ignored; however, some implementations require *fd* to be −1 if **MAP_ANONYMOUS** (or **MAP_ANON**) is specified, and portable applications should ensure this. The *offset* argument should be zero. Support for **MAP_ANONYMOUS** in conjunction with **MAP_SHARED** was added in Linux 2.4.

MAP_DENYWRITE
This flag is ignored. (Long ago—Linux 2.0 and earlier—it signaled that attempts to write to the underlying file should fail with **ETXTBSY**. But this was a source of denial-of-service attacks.)

MAP_EXECUTABLE
This flag is ignored.

MAP_FILE
Compatibility flag. Ignored.

MAP_FIXED
Don't interpret *addr* as a hint: place the mapping at exactly that address. *addr* must be suitably aligned: for most architectures a multiple of the page size is sufficient; however, some architectures may impose additional restrictions. If the memory region specified by *addr* and *length* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, **mmap**() will fail.

Software that aspires to be portable should use the **MAP_FIXED** flag with care, keeping in mind that the exact layout of a process's memory mappings is allowed to change significantly between Linux versions, C library versions, and operating system releases. *Carefully read the discussion of this flag in NOTES!*

MAP_FIXED_NOREPLACE (since Linux 4.17)
This flag provides behavior that is similar to **MAP_FIXED** with respect to the *addr* enforcement, but differs in that **MAP_FIXED_NOREPLACE** never clobbers a preexisting mapped range. If the requested range would collide with an existing mapping, then this call fails with the error **EEXIST.** This flag can therefore be used as a way to atomically (with respect to other threads) attempt to map an address range: one thread will succeed; all others will report failure.

Note that older kernels which do not recognize the **MAP_FIXED_NOREPLACE** flag will typically (upon detecting a collision with a preexisting mapping) fall back to a "non-**MAP_FIXED**" type of behavior: they will return an address that is different from the requested address. Therefore, backward-compatible software should check the returned address against the requested address.

MAP_GROWSDOWN
This flag is used for stacks. It indicates to the kernel virtual memory system that the mapping should extend downward in memory. The return address is one page lower than the memory area that is actually created in the process's virtual address space. Touching an address in the

"guard" page below the mapping will cause the mapping to grow by a page. This growth can be repeated until the mapping grows to within a page of the high end of the next lower mapping, at which point touching the "guard" page will result in a **SIGSEGV** signal.

MAP_HUGETLB (since Linux 2.6.32)
Allocate the mapping using "huge" pages. See the Linux kernel source file *Documentation/admin−guide/mm/hugetlbpage.rst* for further information, as well as NOTES, below.

MAP_HUGE_2MB
MAP_HUGE_1GB (since Linux 3.8)
Used in conjunction with **MAP_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that support multiple hugetlb page sizes.

More generally, the desired huge page size can be configured by encoding the base-2 logarithm of the desired page size in the six bits at the offset **MAP_HUGE_SHIFT**. (A value of zero in this bit field provides the default huge page size; the default huge page size can be discovered via the *Hugepagesize* field exposed by */proc/meminfo*.) Thus, the above two constants are defined as:

```
#define MAP_HUGE_2MB    (21 << MAP_HUGE_SHIFT)
#define MAP_HUGE_1GB    (30 << MAP_HUGE_SHIFT)
```

The range of huge page sizes that are supported by the system can be discovered by listing the subdirectories in */sys/kernel/mm/hugepages*.

MAP_LOCKED (since Linux 2.5.37)
Mark the mapped region to be locked in the same way as *mlock*(2). This implementation will try to populate (prefault) the whole range but the **mmap**() call doesn't fail with **ENOMEM** if this fails. Therefore major faults might happen later on. So the semantic is not as strong as *mlock*(2). One should use **mmap**() plus *mlock*(2) when major faults are not acceptable after the initialization of the mapping. The **MAP_LOCKED** flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)
This flag is meaningful only in conjunction with **MAP_POPULATE**. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM. Since Linux 2.6.23, this flag causes **MAP_POPULATE** to do nothing. One day, the combination of **MAP_POPULATE** and **MAP_NONBLOCK** may be reimplemented.

MAP_NORESERVE
Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in *proc*(5). Before Linux 2.6, this flag had effect only for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)
Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. This will help to reduce blocking on page faults later. The **mmap**() call doesn't fail if the mapping cannot be populated (for example, due to limitations on the number of mapped huge pages when using **MAP_HUGETLB**). Support for **MAP_POPULATE** in conjunction with private mappings was added in Linux 2.6.23.

MAP_STACK (since Linux 2.6.27)
Allocate the mapping at an address suitable for a process or thread stack.

This flag is currently a no-op on Linux. However, by employing this flag, applications can ensure that they transparently obtain support if the flag is implemented in the future. Thus, it is used in the glibc threading implementation to allow for the fact that some architectures may (later) require special treatment for stack allocations. A further reason to employ this flag is portability: **MAP_STACK** exists (and has an effect) on some other systems (e.g., some of the BSDs).

MAP_SYNC (since Linux 4.15)
This flag is available only with the **MAP_SHARED_VALIDATE** mapping type; mappings of type **MAP_SHARED** will silently ignore this flag. This flag is supported only for files supporting DAX (direct mapping of persistent memory). For other files, creating a mapping with

this flag results in an **EOPNOTSUPP** error.

Shared file mappings with this flag provide the guarantee that while some memory is mapped writable in the address space of the process, it will be visible in the same file at the same offset even after the system crashes or is rebooted. In conjunction with the use of appropriate CPU instructions, this provides users of such mappings with a more efficient way of making data modifications persistent.

MAP_UNINITIALIZED (since Linux 2.6.33)
Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if the kernel was configured with the **CONFIG_MMAP_ALLOW_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001 and POSIX.1-2008. However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

munmap()
The **munmap**() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address *addr* must be a multiple of the page size (but *length* need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

RETURN VALUE

On success, **mmap**() returns a pointer to the mapped area. On error, the value **MAP_FAILED** (that is, *(void *) −1*) is returned, and *errno* is set to indicate the error.

On success, **munmap**() returns 0. On failure, it returns −1, and *errno* is set to indicate the error (probably to **EINVAL**).

ERRORS

EACCES
A file descriptor refers to a non-regular file. Or a file mapping was requested, but *fd* is not open for reading. Or **MAP_SHARED** was requested and **PROT_WRITE** is set, but *fd* is not open in read/write (**O_RDWR**) mode. Or **PROT_WRITE** is set, but the file is append-only.

EAGAIN
The file has been locked, or too much memory has been locked (see *setrlimit*(2)).

EBADF
fd is not a valid file descriptor (and **MAP_ANONYMOUS** was not set).

EEXIST
MAP_FIXED_NOREPLACE was specified in *flags*, and the range covered by *addr* and *length* clashes with an existing mapping.

EINVAL
We don't like *addr*, *length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

EINVAL
(since Linux 2.6.12) *length* was 0.

EINVAL
flags contained none of **MAP_PRIVATE**, **MAP_SHARED**, or **MAP_SHARED_VALIDATE**.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
The underlying filesystem of the specified file does not support memory mapping.

ENOMEM
No memory is available.

ENOMEM
The process's maximum number of mappings would have been exceeded. This error can also occur for **munmap**(), when unmapping a region in the middle of an existing mapping, since this results in two smaller mappings on either side of the region being unmapped.

ENOMEM
(since Linux 4.7) The process's **RLIMIT_DATA** limit, described in *getrlimit*(2), would have been exceeded.

ENOMEM
We don't like *addr*, because it exceeds the virtual address space of the CPU.

EOVERFLOW
On 32-bit architecture together with the large file extension (i.e., using 64-bit *off_t*): the number of pages used for *length* plus number of pages used for *offset* would overflow *unsigned long* (32 bits).

EPERM
The *prot* argument asks for **PROT_EXEC** but the mapped area belongs to a file on a filesystem that was mounted no-exec.

EPERM
The operation was prevented by a file seal; see *fcntl*(2).

EPERM
The **MAP_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability) and is not a member of the *sysctl_hugetlb_shm_group* group; see the description of */proc/sys/vm/sysctl_hugetlb_shm_group* in *proc_sys*(5).

ETXTBSY
MAP_DENYWRITE was set but the object specified by *fd* is open for writing.

Use of a mapped region can result in these signals:

SIGSEGV
Attempted write into a region mapped as read-only.

SIGBUS
Attempted access to a page of the buffer that lies beyond the end of the mapped file. For an explanation of the treatment of the bytes in the page that corresponds to the end of a mapped file that is not a multiple of the page size, see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mmap(), **munmap**()	Thread safety	MT-Safe

VERSIONS

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**. It is architecture dependent whether **PROT_READ** implies **PROT_EXEC** or not. Portable programs should always set **PROT_EXEC** if they intend to execute code in the new mapping.

The portable way to create a mapping is to specify *addr* as 0 (NULL), and omit **MAP_FIXED** from *flags*. In this case, the system chooses the address for the mapping; the address is chosen so as not to conflict with any existing mapping, and will not be 0. If the **MAP_FIXED** flag is specified, and *addr* is 0 (NULL), then the mapped address will be 0 (NULL).

Certain *flags* constants are defined only if suitable feature test macros are defined (possibly by default): **_DEFAULT_SOURCE** with glibc 2.19 or later; or **_BSD_SOURCE** or **_SVID_SOURCE** in glibc 2.19 and earlier. (Employing **_GNU_SOURCE** also suffices, and requiring that macro specifically would have been more logical, since these flags are all Linux-specific.) The relevant flags are: **MAP_32BIT**, **MAP_ANONYMOUS** (and the synonym **MAP_ANON**), **MAP_DENYWRITE**, **MAP_EXECUTABLE**, **MAP_FILE**, **MAP_GROWSDOWN**, **MAP_HUGETLB**, **MAP_LOCKED**, **MAP_NONBLOCK**, **MAP_NORESERVE**, **MAP_POPULATE**, and **MAP_STACK**.

C library/kernel differences

This page describes the interface provided by the glibc **mmap**() wrapper function. Originally, this function invoked a system call of the same name. Since Linux 2.4, that system call has been superseded by *mmap2*(2), and nowadays the glibc **mmap**() wrapper function invokes *mmap2*(2) with a suitably adjusted value for *offset*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD.

On POSIX systems on which **mmap**(), *msync*(2), and **munmap**() are available, **_POSIX_MAPPED_FILES** is defined in *<unistd.h>* to a value greater than 0. (See also *sysconf*(3).)

NOTES

Memory mapped by **mmap**() is preserved across *fork*(2), with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining bytes in the partial page at the end of the mapping are zeroed when mapped, and modifications to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

An application can determine which pages of a mapping are currently resident in the buffer/page cache using *mincore*(2).

Using MAP_FIXED safely

The only safe use for **MAP_FIXED** is where the address range specified by *addr* and *length* was previously reserved using another mapping; otherwise, the use of **MAP_FIXED** is hazardous because it forcibly removes preexisting mappings, making it easy for a multithreaded process to corrupt its own address space.

For example, suppose that thread A looks through */proc/*pid*/maps* in order to locate an unused address range that it can map using **MAP_FIXED**, while thread B simultaneously acquires part or all of that same address range. When thread A subsequently employs **mmap(MAP_FIXED)**, it will effectively clobber the mapping that thread B created. In this scenario, thread B need not create a mapping directly; simply making a library call that, internally, uses *dlopen*(3) to load some other shared library, will suffice. The *dlopen*(3) call will map the library into the process's address space. Furthermore, almost any library call may be implemented in a way that adds memory mappings to the address space, either with this technique, or by simply allocating memory. Examples include *brk*(2), *malloc*(3), *pthread_create*(3), and the PAM libraries .

Since Linux 4.17, a multithreaded program can use the **MAP_FIXED_NOREPLACE** flag to avoid the hazard described above when attempting to create a mapping at a fixed address that has not been reserved by a preexisting mapping.

Timestamps changes for file-backed mappings

For file-backed mappings, the *st_atime* field for the mapped file may be updated at any time between the **mmap**() and the corresponding unmapping; the first reference to a mapped page will update the field if it has not been already.

The *st_ctime* and *st_mtime* field for a file mapped with **PROT_WRITE** and **MAP_SHARED** will be updated after a write to the mapped region, and before a subsequent *msync*(2) with the **MS_SYNC** or **MS_ASYNC** flag, if one occurs.

Huge page (Huge TLB) mappings

For mappings that employ huge pages, the requirements for the arguments of **mmap**() and **munmap**() differ somewhat from the requirements for mappings that use the native system page size.

For **mmap**(), *offset* must be a multiple of the underlying huge page size. The system automatically aligns *length* to be a multiple of the underlying huge page size.

For **munmap**(), *addr*, and *length* must both be a multiple of the underlying huge page size.

BUGS

On Linux, there are no guarantees like those suggested above under **MAP_NORESERVE**. By default, any process can be killed at any moment when the system runs out of memory.

Before Linux 2.6.7, the **MAP_POPULATE** flag has effect only if *prot* is specified as **PROT_NONE**.

SUSv3 specifies that **mmap**() should fail if *length* is 0. However, before Linux 2.6.12, **mmap**() succeeded in this case: no mapping was created and the call returned *addr*. Since Linux 2.6.12, **mmap**() fails with the error **EINVAL** for this case.

POSIX specifies that the system shall always zero fill any partial page at the end of the object and that system will never write any modification of the object beyond its end. On Linux, when you write data to such partial page after the end of the object, the data stays in the page cache even after the file is closed and unmapped and even though the data is never written to the file itself, subsequent mappings may see the modified content. In some cases, this could be fixed by calling *msync*(2) before the unmap takes place; however, this doesn't work on *tmpfs*(5) (for example, when using the POSIX shared memory interface documented in *shm_overview*(7)).

EXAMPLES

The following program prints part of the file specified in its first command-line argument to standard output. The range of bytes to be printed is specified via offset and length values in the second and third command-line arguments. The program creates a memory mapping of the required pages of the file and then uses *write*(2) to output the desired bytes.

Program source

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    int          fd;
    char         *addr;
    off_t        offset, pa_offset;
    size_t       length;
    ssize_t      s;
    struct stat  sb;

    if (argc < 3 || argc > 4) {
        fprintf(stderr, "%s file offset [length]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        handle_error("open");

    if (fstat(fd, &sb) == -1)           /* To obtain file size */
        handle_error("fstat");

    offset = atoi(argv[2]);
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
        /* offset for mmap() must be page aligned */

    if (offset >= sb.st_size) {
        fprintf(stderr, "offset is past end of file\n");
        exit(EXIT_FAILURE);
    }
```

```
        if (argc == 4) {
            length = atoi(argv[3]);
            if (offset + length > sb.st_size)
                length = sb.st_size - offset;
                    /* Can't display bytes past end of file */

        } else {    /* No length arg ==> display to end of file */
            length = sb.st_size - offset;
        }

        addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
                    MAP_PRIVATE, fd, pa_offset);
        if (addr == MAP_FAILED)
            handle_error("mmap");

        s = write(STDOUT_FILENO, addr + offset - pa_offset, length);
        if (s != length) {
            if (s == -1)
                handle_error("write");

            fprintf(stderr, "partial write");
            exit(EXIT_FAILURE);
        }

        munmap(addr, length + offset - pa_offset);
        close(fd);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ftruncate(2), *getpagesize*(2), *memfd_create*(2), *mincore*(2), *mlock*(2), *mmap2*(2), *mprotect*(2), *mremap*(2), *msync*(2), *remap_file_pages*(2), *setrlimit*(2), *shmat*(2), *userfaultfd*(2), *shm_open*(3), *shm_overview*(7)

The descriptions of the following files in *proc*(5): */proc/*pid*/maps*, */proc/*pid*/map_files*, and */proc/*pid*/smaps*.

B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128–129 and 389–391.

NAME

mmap2 − map files or devices into memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/mman.h>    /* Definition of MAP_* and PROT_* constants */
#include <sys/syscall.h> /* Definition of SYS_* constants */
#include <unistd.h>

void *syscall(SYS_mmap2, unsigned long addr, unsigned long length,
              unsigned long prot, unsigned long flags,
              unsigned long fd, unsigned long pgoffset);
```

DESCRIPTION

This is probably not the system call that you are interested in; instead, see *mmap*(2), which describes the glibc wrapper function that invokes this system call.

The **mmap2**() system call provides the same interface as *mmap*(2), except that the final argument specifies the offset into the file in 4096-byte units (instead of bytes, as is done by *mmap*(2)). This enables applications that use a 32-bit *off_t* to map large files (up to 2^44 bytes).

RETURN VALUE

On success, **mmap2**() returns a pointer to the mapped area. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
: Problem with getting the data from user space.

EINVAL
: (Various platforms where the page size is not 4096 bytes.) *offset * 4096* is not a multiple of the system page size.

mmap2() can also return any of the errors described in *mmap*(2).

VERSIONS

On architectures where this system call is present, the glibc **mmap**() wrapper function invokes this system call rather than the *mmap*(2) system call.

This system call does not exist on x86-64.

On ia64, the unit for *offset* is actually the system page size, rather than 4096 bytes.

STANDARDS

Linux.

HISTORY

Linux 2.3.31.

SEE ALSO

getpagesize(2), *mmap*(2), *mremap*(2), *msync*(2), *shm_open*(3)

NAME

modify_ldt − get or set a per-process LDT entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <asm/ldt.h>      /* Definition of struct user_desc */
#include <sys/syscall.h>  /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_modify_ldt, int  func, void ptr[.bytecount],
        unsigned long bytecount);
```

Note: glibc provides no wrapper for **modify_ldt**(), necessitating the use of *syscall*(2).

DESCRIPTION

modify_ldt() reads or writes the local descriptor table (LDT) for a process. The LDT is an array of segment descriptors that can be referenced by user code. Linux allows processes to configure a per-process (actually per-mm) LDT. For more information about the LDT, see the Intel Software Developer's Manual or the AMD Architecture Programming Manual.

When *func* is 0, **modify_ldt**() reads the LDT into the memory pointed to by *ptr*. The number of bytes read is the smaller of *bytecount* and the actual size of the LDT, although the kernel may act as though the LDT is padded with additional trailing zero bytes. On success, **modify_ldt**() will return the number of bytes read.

When *func* is 1 or 0x11, **modify_ldt**() modifies the LDT entry indicated by *ptr−>entry_number*. *ptr* points to a *user_desc* structure and *bytecount* must equal the size of this structure.

The *user_desc* structure is defined in <*asm/ldt.h*> as:

```
struct user_desc {
    unsigned int  entry_number;
    unsigned int  base_addr;
    unsigned int  limit;
    unsigned int  seg_32bit:1;
    unsigned int  contents:2;
    unsigned int  read_exec_only:1;
    unsigned int  limit_in_pages:1;
    unsigned int  seg_not_present:1;
    unsigned int  useable:1;
};
```

In Linux 2.4 and earlier, this structure was named *modify_ldt_ldt_s*.

The *contents* field is the segment type (data, expand-down data, non-conforming code, or conforming code). The other fields match their descriptions in the CPU manual, although **modify_ldt**() cannot set the hardware-defined "accessed" bit described in the CPU manual.

A *user_desc* is considered "empty" if *read_exec_only* and *seg_not_present* are set to 1 and all of the other fields are 0. An LDT entry can be cleared by setting it to an "empty" *user_desc* or, if *func* is 1, by setting both *base* and *limit* to 0.

A conforming code segment (i.e., one with *contents==3*) will be rejected if *func* is 1 or if *seg_not_present* is 0.

When *func* is 2, **modify_ldt**() will read zeros. This appears to be a leftover from Linux 2.4.

RETURN VALUE

On success, **modify_ldt**() returns either the actual number of bytes read (for reading) or 0 (for writing). On failure, **modify_ldt**() returns −1 and sets *errno* to indicate the error.

ERRORS

EFAULT

ptr points outside the address space.

EINVAL
ptr is 0, or *func* is 1 and *bytecount* is not equal to the size of the structure *user_desc*, or *func* is 1 or 0x11 and the new LDT entry has invalid values.

ENOSYS
func is neither 0, 1, 2, nor 0x11.

STANDARDS
Linux.

NOTES
modify_ldt() should not be used for thread-local storage, as it slows down context switches and only supports a limited number of threads. Threading libraries should use *set_thread_area*(2) or *arch_prctl*(2) instead, except on extremely old kernels that do not support those system calls.

The normal use for **modify_ldt**() is to run legacy 16-bit or segmented 32-bit code. Not all kernels allow 16-bit segments to be installed, however.

Even on 64-bit kernels, **modify_ldt**() cannot be used to create a long mode (i.e., 64-bit) code segment. The undocumented field "lm" in *user_desc* is not useful, and, despite its name, does not result in a long mode segment.

BUGS
On 64-bit kernels before Linux 3.19, setting the "lm" bit in *user_desc* prevents the descriptor from being considered empty. Keep in mind that the "lm" bit does not exist in the 32-bit headers, but these buggy kernels will still notice the bit even when set in a 32-bit process.

SEE ALSO
arch_prctl(2), *set_thread_area*(2), *vm86*(2)

NAME

mount − mount filesystem

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/mount.h>

int mount(const char *source, const char *target,
          const char * filesystemtype, unsigned long mountflags,
          const void *_Nullable data);
```

DESCRIPTION

mount() attaches the filesystem specified by *source* (which is often a pathname referring to a device, but can also be the pathname of a directory or file, or a dummy string) to the location (a directory or file) specified by the pathname in *target*.

Appropriate privilege (Linux: the **CAP_SYS_ADMIN** capability) is required to mount filesystems.

Values for the *filesystemtype* argument supported by the kernel are listed in */proc/filesystems* (e.g., "btrfs", "ext4", "jfs", "xfs", "vfat", "fuse", "tmpfs", "cgroup", "proc", "mqueue", "nfs", "cifs", "iso9660"). Further types may become available when the appropriate modules are loaded.

The *data* argument is interpreted by the different filesystems. Typically it is a string of comma-separated options understood by this filesystem. See *mount*(8) for details of the options available for each filesystem type. This argument may be specified as NULL, if there are no options.

A call to **mount**() performs one of a number of general types of operation, depending on the bits specified in *mountflags*. The choice of which operation to perform is determined by testing the bits set in *mountflags*, with the tests being conducted in the order listed here:

- Remount an existing mount: *mountflags* includes **MS_REMOUNT**.
- Create a bind mount: *mountflags* includes **MS_BIND**.
- Change the propagation type of an existing mount: *mountflags* includes one of **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE**.
- Move an existing mount to a new location: *mountflags* includes **MS_MOVE**.
- Create a new mount: *mountflags* includes none of the above flags.

Each of these operations is detailed later in this page. Further flags may be specified in *mountflags* to modify the behavior of **mount**(), as described below.

Additional mount flags

The list below describes the additional flags that can be specified in *mountflags*. Note that some operation types ignore some or all of these flags, as described later in this page.

MS_DIRSYNC (since Linux 2.5.19)
: Make directory changes on this filesystem synchronous. (This property can be obtained for individual directories or subtrees using *chattr*(1)

MS_LAZYTIME (since Linux 4.0)
: Reduce on-disk updates of inode timestamps (atime, mtime, ctime) by maintaining these changes only in memory. The on-disk timestamps are updated only when:

  - the inode needs to be updated for some change unrelated to file timestamps;
  - the application employs *fsync*(2), *syncfs*(2), or *sync*(2);
  - an undeleted inode is evicted from memory; or
  - more than 24 hours have passed since the inode was written to disk.

  This mount option significantly reduces writes needed to update the inode's timestamps, especially mtime and atime. However, in the event of a system crash, the atime and mtime fields on disk might be out of date by up to 24 hours.

  Examples of workloads where this option could be of significant benefit include frequent random writes to preallocated files, as well as cases where the **MS_STRICTATIME** mount

option is also enabled. (The advantage of combining **MS_STRICTATIME** and **MS_LAZYTIME** is that *stat*(2) will return the correctly updated atime, but the atime updates will be flushed to disk only in the cases listed above.)

MS_MANDLOCK
Permit mandatory locking on files in this filesystem. (Mandatory locking must still be enabled on a per-file basis, as described in *fcntl*(2).) Since Linux 4.5, this mount option requires the **CAP_SYS_ADMIN** capability and a kernel configured with the **CONFIG_MANDATORY_FILE_LOCKING** option. Mandatory locking has been fully deprecated in Linux 5.15, so this flag should be considered deprecated.

MS_NOATIME
Do not update access times for (all types of) files on this filesystem.

MS_NODEV
Do not allow access to devices (special files) on this filesystem.

MS_NODIRATIME
Do not update access times for directories on this filesystem. This flag provides a subset of the functionality provided by **MS_NOATIME**; that is, **MS_NOATIME** implies **MS_NODIRATIME**.

MS_NOEXEC
Do not allow programs to be executed from this filesystem.

MS_NOSUID
Do not honor set-user-ID and set-group-ID bits or file capabilities when executing programs from this filesystem. In addition, SELinux domain transitions require the permission *nosuid_transition*, which in turn needs also the policy capability *nnp_nosuid_transition*.

MS_RDONLY
Mount filesystem read-only.

MS_REC (since Linux 2.4.11)
Used in conjunction with **MS_BIND** to create a recursive bind mount, and in conjunction with the propagation type flags to recursively change the propagation type of all of the mounts in a subtree. See below for further details.

MS_RELATIME (since Linux 2.6.20)
When a file on this filesystem is accessed, update the file's last access time (atime) only if the current value of atime is less than or equal to the file's last modification time (mtime) or last status change time (ctime). This option is useful for programs, such as *mutt*(1), that need to know when a file has been read since it was last modified. Since Linux 2.6.30, the kernel defaults to the behavior provided by this flag (unless **MS_NOATIME** was specified), and the **MS_STRICTATIME** flag is required to obtain traditional semantics. In addition, since Linux 2.6.30, the file's last access time is always updated if it is more than 1 day old.

MS_SILENT (since Linux 2.6.17)
Suppress the display of certain (*printk*()) warning messages in the kernel log. This flag supersedes the misnamed and obsolete **MS_VERBOSE** flag (available since Linux 2.4.12), which has the same meaning.

MS_STRICTATIME (since Linux 2.6.30)
Always update the last access time (atime) when files on this filesystem are accessed. (This was the default behavior before Linux 2.6.30.) Specifying this flag overrides the effect of setting the **MS_NOATIME** and **MS_RELATIME** flags.

MS_SYNCHRONOUS
Make writes on this filesystem synchronous (as though the **O_SYNC** flag to *open*(2) was specified for all file opens to this filesystem).

MS_NOSYMFOLLOW (since Linux 5.10)
Do not follow symbolic links when resolving paths. Symbolic links can still be created, and *readlink*(1), *readlink*(2), *realpath*(1), and *realpath*(3) all still work properly.

From Linux 2.4 onward, some of the above flags are settable on a per-mount basis, while others apply to the superblock of the mounted filesystem, meaning that all mounts of the same filesystem share those

flags. (Previously, all of the flags were per-superblock.)

The per-mount-point flags are as follows:

- Since Linux 2.4: **MS_NODEV**, **MS_NOEXEC**, and **MS_NOSUID** flags are settable on a per-mount-point basis.
- Additionally, since Linux 2.6.16: **MS_NOATIME** and **MS_NODIRATIME**.
- Additionally, since Linux 2.6.20: **MS_RELATIME**.

The following flags are per-superblock: **MS_DIRSYNC**, **MS_LAZYTIME**, **MS_MANDLOCK**, **MS_SILENT**, and **MS_SYNCHRONOUS**. The initial settings of these flags are determined on the first mount of the filesystem, and will be shared by all subsequent mounts of the same filesystem. Subsequently, the settings of the flags can be changed via a remount operation (see below). Such changes will be visible via all mounts associated with the filesystem.

Since Linux 2.6.16, **MS_RDONLY** can be set or cleared on a per-mount-point basis as well as on the underlying filesystem superblock. The mounted filesystem will be writable only if neither the filesystem nor the mountpoint are flagged as read-only.

Remounting an existing mount

An existing mount may be remounted by specifying **MS_REMOUNT** in *mountflags*. This allows you to change the *mountflags* and *data* of an existing mount without having to unmount and remount the filesystem. *target* should be the same value specified in the initial **mount**() call.

The *source* and *filesystemtype* arguments are ignored.

The *mountflags* and *data* arguments should match the values used in the original **mount**() call, except for those parameters that are being deliberately changed.

The following *mountflags* can be changed: **MS_LAZYTIME**, **MS_MANDLOCK**, **MS_NOATIME**, **MS_NODEV**, **MS_NODIRATIME**, **MS_NOEXEC**, **MS_NOSUID**, **MS_RELATIME**, **MS_RDONLY**, **MS_STRICTATIME** (whose effect is to clear the **MS_NOATIME** and **MS_RELATIME** flags), and **MS_SYNCHRONOUS**. Attempts to change the setting of the **MS_DIRSYNC** and **MS_SILENT** flags during a remount are silently ignored. Note that changes to per-superblock flags are visible via all mounts of the associated filesystem (because the per-superblock flags are shared by all mounts).

Since Linux 3.17, if none of **MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME**, or **MS_STRICTATIME** is specified in *mountflags*, then the remount operation preserves the existing values of these flags (rather than defaulting to **MS_RELATIME**).

Since Linux 2.6.26, the **MS_REMOUNT** flag can be used with **MS_BIND** to modify only the per-mount-point flags. This is particularly useful for setting or clearing the "read-only" flag on a mount without changing the underlying filesystem. Specifying *mountflags* as:

```
MS_REMOUNT | MS_BIND | MS_RDONLY
```

will make access through this mountpoint read-only, without affecting other mounts.

Creating a bind mount

If *mountflags* includes **MS_BIND** (available since Linux 2.4), then perform a bind mount. A bind mount makes a file or a directory subtree visible at another point within the single directory hierarchy. Bind mounts may cross filesystem boundaries and span *chroot*(2) jails.

The *filesystemtype* and *data* arguments are ignored.

The remaining bits (other than **MS_REC**, described below) in the *mountflags* argument are also ignored. (The bind mount has the same mount options as the underlying mount.) However, see the discussion of remounting above, for a method of making an existing bind mount read-only.

By default, when a directory is bind mounted, only that directory is mounted; if there are any submounts under the directory tree, they are not bind mounted. If the **MS_REC** flag is also specified, then a recursive bind mount operation is performed: all submounts under the *source* subtree (other than unbindable mounts) are also bind mounted at the corresponding location in the *target* subtree.

Changing the propagation type of an existing mount

If *mountflags* includes one of **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE** (all available since Linux 2.6.15), then the propagation type of an existing mount is changed. If more

than one of these flags is specified, an error results.

The only other flags that can be specified while changing the propagation type are **MS_REC** (described below) and **MS_SILENT** (which is ignored).

The *source*, *filesystemtype*, and *data* arguments are ignored.

The meanings of the propagation type flags are as follows:

MS_SHARED
: Make this mount shared. Mount and unmount events immediately under this mount will propagate to the other mounts that are members of this mount's peer group. Propagation here means that the same mount or unmount will automatically occur under all of the other mounts in the peer group. Conversely, mount and unmount events that take place under peer mounts will propagate to this mount.

MS_PRIVATE
: Make this mount private. Mount and unmount events do not propagate into or out of this mount.

MS_SLAVE
: If this is a shared mount that is a member of a peer group that contains other members, convert it to a slave mount. If this is a shared mount that is a member of a peer group that contains no other members, convert it to a private mount. Otherwise, the propagation type of the mount is left unchanged.

  When a mount is a slave, mount and unmount events propagate into this mount from the (master) shared peer group of which it was formerly a member. Mount and unmount events under this mount do not propagate to any peer.

  A mount can be the slave of another peer group while at the same time sharing mount and unmount events with a peer group of which it is a member.

MS_UNBINDABLE
: Make this mount unbindable. This is like a private mount, and in addition this mount can't be bind mounted. When a recursive bind mount (**mount**() with the **MS_BIND** and **MS_REC** flags) is performed on a directory subtree, any unbindable mounts within the subtree are automatically pruned (i.e., not replicated) when replicating that subtree to produce the target subtree.

By default, changing the propagation type affects only the *target* mount. If the **MS_REC** flag is also specified in *mountflags*, then the propagation type of all mounts under *target* is also changed.

For further details regarding mount propagation types (including the default propagation type assigned to new mounts), see *mount_namespaces*(7).

Moving a mount
: If *mountflags* contains the flag **MS_MOVE** (available since Linux 2.4.18), then move a subtree: *source* specifies an existing mount and *target* specifies the new location to which that mount is to be relocated. The move is atomic: at no point is the subtree unmounted.

  The remaining bits in the *mountflags* argument are ignored, as are the *filesystemtype* and *data* arguments.

Creating a new mount
: If none of **MS_REMOUNT**, **MS_BIND**, **MS_MOVE**, **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE** is specified in *mountflags*, then **mount**() performs its default action: creating a new mount. *source* specifies the source for the new mount, and *target* specifies the directory at which to create the mount point.

  The *filesystemtype* and *data* arguments are employed, and further bits may be specified in *mountflags* to modify the behavior of the call.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The error values given below result from filesystem type independent errors. Each filesystem type may have its own special errors and its own special behavior. See the Linux kernel source code for details.

EACCES
A component of a path was not searchable. (See also *path_resolution*(7).)

EACCES
Mounting a read-only filesystem was attempted without giving the **MS_RDONLY** flag.

The filesystem may be read-only for various reasons, including: it resides on a read-only optical disk; it is resides on a device with a physical switch that has been set to mark the device read-only; the filesystem implementation was compiled with read-only support; or errors were detected when initially mounting the filesystem, so that it was marked read-only and can't be remounted as read-write (until the errors are fixed).

Some filesystems instead return the error **EROFS** on an attempt to mount a read-only filesystem.

EACCES
The block device *source* is located on a filesystem mounted with the **MS_NODEV** option.

EBUSY
An attempt was made to stack a new mount directly on top of an existing mount point that was created in this mount namespace with the same *source* and *target*.

EBUSY
source cannot be remounted read-only, because it still holds files open for writing.

EFAULT
One of the pointer arguments points outside the user address space.

EINVAL
source had an invalid superblock.

EINVAL
A remount operation (**MS_REMOUNT**) was attempted, but *source* was not already mounted on *target*.

EINVAL
A move operation (**MS_MOVE**) was attempted, but the mount tree under *source* includes unbindable mounts and *target* is a mount that has propagation type **MS_SHARED**.

EINVAL
A move operation (**MS_MOVE**) was attempted, but the parent mount of *source* mount has propagation type **MS_SHARED**.

EINVAL
A move operation (**MS_MOVE**) was attempted, but *source* was not a mount, or was '/'.

EINVAL
A bind operation (**MS_BIND**) was requested where *source* referred a mount namespace magic link (i.e., a */proc/*pid*/ns/mnt* magic link or a bind mount to such a link) and the propagation type of the parent mount of *target* was **MS_SHARED**, but propagation of the requested bind mount could lead to a circular dependency that might prevent the mount namespace from ever being freed.

EINVAL
mountflags includes more than one of **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE**.

EINVAL
mountflags includes **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE** and also includes a flag other than **MS_REC** or **MS_SILENT**.

EINVAL
An attempt was made to bind mount an unbindable mount.

EINVAL
In an unprivileged mount namespace (i.e., a mount namespace owned by a user namespace that was created by an unprivileged user), a bind mount operation (**MS_BIND**) was attempted without specifying (**MS_REC**), which would have revealed the filesystem tree underneath one of the submounts of the directory being bound.

ELOOP
Too many links encountered during pathname resolution.

ELOOP
A move operation was attempted, and *target* is a descendant of *source*.

EMFILE
(In case no block device is required:) Table of dummy devices is full.

ENAMETOOLONG
A pathname was longer than **MAXPATHLEN**.

ENODEV
filesystemtype not configured in the kernel.

ENOENT
A pathname was empty or had a nonexistent component.

ENOMEM
The kernel could not allocate a free page to copy filenames or data into.

ENOTBLK
source is not a block device (and a device was required).

ENOTDIR
target, or a prefix of *source*, is not a directory.

ENXIO
The major number of the block device *source* is out of range.

EPERM
The caller does not have the required privileges.

EPERM
An attempt was made to modify (**MS_REMOUNT**) the **MS_RDONLY**, **MS_NOSUID**, or **MS_NOEXEC** flag, or one of the "atime" flags (**MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME**) of an existing mount, but the mount is locked; see *mount_namespaces*(7).

EROFS
Mounting a read-only filesystem was attempted without giving the **MS_RDONLY** flag. See **EACCES**, above.

STANDARDS
Linux.

HISTORY
The definitions of **MS_DIRSYNC**, **MS_MOVE**, **MS_PRIVATE**, **MS_REC**, **MS_RELATIME**, **MS_SHARED**, **MS_SLAVE**, **MS_STRICTATIME**, and **MS_UNBINDABLE** were added to glibc headers in glibc 2.12.

Since Linux 2.4 a single filesystem can be mounted at multiple mount points, and multiple mounts can be stacked on the same mount point.

The *mountflags* argument may have the magic number 0xC0ED (**MS_MGC_VAL**) in the top 16 bits. (All of the other flags discussed in DESCRIPTION occupy the low order 16 bits of *mountflags*.) Specifying **MS_MGC_VAL** was required before Linux 2.4, but since Linux 2.4 is no longer required and is ignored if specified.

The original **MS_SYNC** flag was renamed **MS_SYNCHRONOUS** in 1.1.69 when a different **MS_SYNC** was added to <*mman.h*>.

Before Linux 2.4 an attempt to execute a set-user-ID or set-group-ID program on a filesystem mounted with **MS_NOSUID** would fail with **EPERM**. Since Linux 2.4 the set-user-ID and set-group-ID bits are just silently ignored in this case.

NOTES
Mount namespaces
Starting with Linux 2.4.19, Linux provides mount namespaces. A mount namespace is the set of filesystem mounts that are visible to a process. Mount namespaces can be (and usually are) shared between multiple processes, and changes to the namespace (i.e., mounts and unmounts) by one process

are visible to all other processes sharing the same namespace. (The pre-2.4.19 Linux situation can be considered as one in which a single namespace was shared by every process on the system.)

A child process created by *fork*(2) shares its parent's mount namespace; the mount namespace is preserved across an *execve*(2).

A process can obtain a private mount namespace if: it was created using the *clone*(2) **CLONE_NEWNS** flag, in which case its new namespace is initialized to be a *copy* of the namespace of the process that called *clone*(2); or it calls *unshare*(2) with the **CLONE_NEWNS** flag, which causes the caller's mount namespace to obtain a private copy of the namespace that it was previously sharing with other processes, so that future mounts and unmounts by the caller are invisible to other processes (except child processes that the caller subsequently creates) and vice versa.

For further details on mount namespaces, see *mount_namespaces*(7).

Parental relationship between mounts

Each mount has a parent mount. The overall parental relationship of all mounts defines the single directory hierarchy seen by the processes within a mount namespace.

The parent of a new mount is defined when the mount is created. In the usual case, the parent of a new mount is the mount of the filesystem containing the directory or file at which the new mount is attached. In the case where a new mount is stacked on top of an existing mount, the parent of the new mount is the previous mount that was stacked at that location.

The parental relationship between mounts can be discovered via the */proc/*pid*/mountinfo* file (see below).

/proc/*pid*/mounts* and */proc/*pid*/mountinfo

The Linux-specific */proc/*pid*/mounts* file exposes the list of mounts in the mount namespace of the process with the specified ID. The */proc/*pid*/mountinfo* file exposes even more information about mounts, including the propagation type and mount ID information that makes it possible to discover the parental relationship between mounts. See *proc*(5) and *mount_namespaces*(7) for details of this file.

SEE ALSO

mountpoint(1), *chroot*(2), *ioctl_iflags*(2), *mount_setattr*(2), *pivot_root*(2), *umount*(2), *mount_namespaces*(7), *path_resolution*(7), *findmnt*(8), *lsblk*(8), *mount*(8), *umount*(8)

NAME

mount_setattr − change properties of a mount or mount tree

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/fcntl.h> /* Definition of **AT_*** constants */
#include <linux/mount.h> /* Definition of **MOUNT_ATTR_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_mount_setattr, int *dirfd***, const char *** *pathname***,**
unsigned int *flags***, struct mount_attr *** *attr***, size_t** *size***);**

Note: glibc provides no wrapper for **mount_setattr**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **mount_setattr**() system call changes the mount properties of a mount or an entire mount tree. If *pathname* is a relative pathname, then it is interpreted relative to the directory referred to by the file descriptor *dirfd*. If *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process. If *pathname* is the empty string and **AT_EMPTY_PATH** is specified in *flags*, then the mount properties of the mount identified by *dirfd* are changed. (See *openat*(2) for an explanation of why the *dirfd* argument is useful.)

The **mount_setattr**() system call uses an extensible structure (*struct mount_attr*) to allow for future extensions. Any non-flag extensions to **mount_setattr**() will be implemented as new fields appended to the this structure, with a zero value in a new field resulting in the kernel behaving as though that extension field was not present. Therefore, the caller *must* zero-fill this structure on initialization. See the "Extensibility" subsection under **NOTES** for more details.

The *size* argument should usually be specified as *sizeof(struct mount_attr)*. However, if the caller is using a kernel that supports an extended *struct mount_attr*, but the caller does not intend to make use of these features, it is possible to pass the size of an earlier version of the structure together with the extended structure. This allows the kernel to not copy later parts of the structure that aren't used anyway. With each extension that changes the size of *struct mount_attr*, the kernel will expose a definition of the form **MOUNT_ATTR_SIZE_VER***number*. For example, the macro for the size of the initial version of *struct mount_attr* is **MOUNT_ATTR_SIZE_VER0**.

The *flags* argument can be used to alter the pathname resolution behavior. The supported values are:

AT_EMPTY_PATH
If *pathname* is the empty string, change the mount properties on *dirfd* itself.

AT_RECURSIVE
Change the mount properties of the entire mount tree.

AT_SYMLINK_NOFOLLOW
Don't follow trailing symbolic links.

AT_NO_AUTOMOUNT
Don't trigger automounts.

The *attr* argument of **mount_setattr**() is a structure of the following form:

```
struct mount_attr {
    __u64 attr_set;     /* Mount properties to set */
    __u64 attr_clr;     /* Mount properties to clear */
    __u64 propagation;  /* Mount propagation type */
    __u64 userns_fd;    /* User namespace file descriptor */
};
```

The *attr_set* and *attr_clr* members are used to specify the mount properties that are supposed to be set or cleared for a mount or mount tree. Flags set in *attr_set* enable a property on a mount or mount tree, and flags set in *attr_clr* remove a property from a mount or mount tree.

When changing mount properties, the kernel will first clear the flags specified in the *attr_clr* field, and then set the flags specified in the *attr_set* field. For example, these settings:

```
struct mount_attr attr = {
    .attr_clr = MOUNT_ATTR_NOEXEC | MOUNT_ATTR_NODEV,
    .attr_set = MOUNT_ATTR_RDONLY | MOUNT_ATTR_NOSUID,
};
```

are equivalent to the following steps:

```
unsigned int current_mnt_flags = mnt->mnt_flags;

/*
 * Clear all flags set in .attr_clr,
 * clearing MOUNT_ATTR_NOEXEC and MOUNT_ATTR_NODEV.
 */
current_mnt_flags &= ~attr->attr_clr;

/*
 * Now set all flags set in .attr_set,
 * applying MOUNT_ATTR_RDONLY and MOUNT_ATTR_NOSUID.
 */
current_mnt_flags |= attr->attr_set;

mnt->mnt_flags = current_mnt_flags;
```

As a result of this change, the mount or mount tree (a) is read-only; (b) blocks the execution of set-user-ID and set-group-ID programs; (c) allows execution of programs; and (d) allows access to devices.

Multiple changes with the same set of flags requested in *attr_clr* and *attr_set* are guaranteed to be idempotent after the changes have been applied.

The following mount attributes can be specified in the *attr_set* or *attr_clr* fields:

MOUNT_ATTR_RDONLY
: If set in *attr_set*, makes the mount read-only. If set in *attr_clr*, removes the read-only setting if set on the mount.

MOUNT_ATTR_NOSUID
: If set in *attr_set*, causes the mount not to honor the set-user-ID and set-group-ID mode bits and file capabilities when executing programs. If set in *attr_clr*, clears the set-user-ID, set-group-ID, and file capability restriction if set on this mount.

MOUNT_ATTR_NODEV
: If set in *attr_set*, prevents access to devices on this mount. If set in *attr_clr*, removes the restriction that prevented accessing devices on this mount.

MOUNT_ATTR_NOEXEC
: If set in *attr_set*, prevents executing programs on this mount. If set in *attr_clr*, removes the restriction that prevented executing programs on this mount.

MOUNT_ATTR_NOSYMFOLLOW
: If set in *attr_set*, prevents following symbolic links on this mount. If set in *attr_clr*, removes the restriction that prevented following symbolic links on this mount.

MOUNT_ATTR_NODIRATIME
: If set in *attr_set*, prevents updating access time for directories on this mount. If set in *attr_clr*, removes the restriction that prevented updating access time for directories. Note that **MOUNT_ATTR_NODIRATIME** can be combined with other access-time settings and is implied by the noatime setting. All other access-time settings are mutually exclusive.

MOUNT_ATTR__ATIME - changing access-time settings
: The access-time values listed below are an enumeration that includes the value zero, expressed in the bits defined by the mask **MOUNT_ATTR__ATIME**. Even though these bits are an enumeration (in contrast to the other mount flags such as **MOUNT_ATTR_NOEXEC**), they are nonetheless passed in *attr_set* and *attr_clr* for consistency with *fsmount*(2), which introduced this behavior.

Note that, since the access-time values are an enumeration rather than bit values, a caller wanting to transition to a different access-time setting cannot simply specify the access-time setting in *attr_set*, but must also include **MOUNT_ATTR__ATIME** in the *attr_clr* field. The kernel will verify that **MOUNT_ATTR__ATIME** isn't partially set in *attr_clr* (i.e., either all bits in the **MOUNT_ATTR__ATIME** bit field are either set or clear), and that *attr_set* doesn't have any access-time bits set if **MOUNT_ATTR__ATIME** isn't set in *attr_clr*.

MOUNT_ATTR_RELATIME
When a file is accessed via this mount, update the file's last access time (atime) only if the current value of atime is less than or equal to the file's last modification time (mtime) or last status change time (ctime).

To enable this access-time setting on a mount or mount tree, **MOUNT_ATTR_RELATIME** must be set in *attr_set* and **MOUNT_ATTR__ATIME** must be set in the *attr_clr* field.

MOUNT_ATTR_NOATIME
Do not update access times for (all types of) files on this mount.

To enable this access-time setting on a mount or mount tree, **MOUNT_ATTR_NOATIME** must be set in *attr_set* and **MOUNT_ATTR__ATIME** must be set in the *attr_clr* field.

MOUNT_ATTR_STRICTATIME
Always update the last access time (atime) when files are accessed on this mount.

To enable this access-time setting on a mount or mount tree, **MOUNT_ATTR_STRICTATIME** must be set in *attr_set* and **MOUNT_ATTR__ATIME** must be set in the *attr_clr* field.

MOUNT_ATTR_IDMAP
If set in *attr_set*, creates an ID-mapped mount. The ID mapping is taken from the user namespace specified in *userns_fd* and attached to the mount.

Since it is not supported to change the ID mapping of a mount after it has been ID mapped, it is invalid to specify **MOUNT_ATTR_IDMAP** in *attr_clr*.

For further details, see the subsection "ID-mapped mounts" under NOTES.

The *propagation* field is used to specify the propagation type of the mount or mount tree. This field either has the value zero, meaning leave the propagation type unchanged, or it has one of the following values:

MS_PRIVATE
Turn all mounts into private mounts.

MS_SHARED
Turn all mounts into shared mounts.

MS_SLAVE
Turn all mounts into dependent mounts.

MS_UNBINDABLE
Turn all mounts into unbindable mounts.

For further details on the above propagation types, see *mount_namespaces*(7).

RETURN VALUE

On success, **mount_setattr**() returns zero. On error, −1 is returned and *errno* is set to indicate the cause of the error.

ERRORS

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EBADF
userns_fd is not a valid file descriptor.

EBUSY
The caller tried to change the mount to **MOUNT_ATTR_RDONLY**, but the mount still holds files open for writing.

EBUSY
The caller tried to create an ID-mapped mount raising **MOUNT_ATTR_IDMAP** and specifying *userns_fd* but the mount still holds files open for writing.

EINVAL
The pathname specified via the *dirfd* and *pathname* arguments to **mount_setattr**() isn't a mount point.

EINVAL
An unsupported value was set in *flags*.

EINVAL
An unsupported value was specified in the *attr_set* field of *mount_attr*.

EINVAL
An unsupported value was specified in the *attr_clr* field of *mount_attr*.

EINVAL
An unsupported value was specified in the *propagation* field of *mount_attr*.

EINVAL
More than one of **MS_SHARED**, **MS_SLAVE**, **MS_PRIVATE**, or **MS_UNBINDABLE** was set in the *propagation* field of *mount_attr*.

EINVAL
An access-time setting was specified in the *attr_set* field without **MOUNT_ATTR__ATIME** being set in the *attr_clr* field.

EINVAL
MOUNT_ATTR_IDMAP was specified in *attr_clr*.

EINVAL
A file descriptor value was specified in *userns_fd* which exceeds **INT_MAX**.

EINVAL
A valid file descriptor value was specified in *userns_fd*, but the file descriptor did not refer to a user namespace.

EINVAL
The underlying filesystem does not support ID-mapped mounts.

EINVAL
The mount that is to be ID mapped is not a detached mount; that is, the mount has not previously been visible in a mount namespace.

EINVAL
A partial access-time setting was specified in *attr_clr* instead of **MOUNT_ATTR__ATIME** being set.

EINVAL
The mount is located outside the caller's mount namespace.

EINVAL
The underlying filesystem has been mounted in a mount namespace that is owned by a noninitial user namespace

ENOENT
A pathname was empty or had a nonexistent component.

ENOMEM
When changing mount propagation to **MS_SHARED**, a new peer group ID needs to be allocated for all mounts without a peer group ID set. This allocation failed because there was not enough memory to allocate the relevant internal structures.

ENOSPC
When changing mount propagation to **MS_SHARED**, a new peer group ID needs to be allocated for all mounts without a peer group ID set. This allocation failed because the kernel has run out of IDs.

EPERM
One of the mounts had at least one of **MOUNT_ATTR_NOATIME**, **MOUNT_ATTR_NODEV**, **MOUNT_ATTR_NODIRATIME**, **MOUNT_ATTR_NOEXEC**, **MOUNT_ATTR_NOSUID**, or **MOUNT_ATTR_RDONLY** set and the flag is locked. Mount attributes become locked on a mount if:

- A new mount or mount tree is created causing mount propagation across user namespaces (i.e., propagation to a mount namespace owned by a different user namespace). The kernel will lock the aforementioned flags to prevent these sensitive properties from being altered.
- A new mount and user namespace pair is created. This happens for example when specifying **CLONE_NEWUSER | CLONE_NEWNS** in *unshare*(2), *clone*(2), or *clone3*(2). The aforementioned flags become locked in the new mount namespace to prevent sensitive mount properties from being altered. Since the newly created mount namespace will be owned by the newly created user namespace, a calling process that is privileged in the new user namespace would—in the absence of such locking—be able to alter sensitive mount properties (e.g., to remount a mount that was marked read-only as read-write in the new mount namespace).

EPERM
A valid file descriptor value was specified in *userns_fd*, but the file descriptor refers to the initial user namespace.

EPERM
An attempt was made to add an ID mapping to a mount that is already ID mapped.

EPERM
The caller does not have **CAP_SYS_ADMIN** in the initial user namespace.

STANDARDS
Linux.

HISTORY
Linux 5.12.

NOTES
ID-mapped mounts
Creating an ID-mapped mount makes it possible to change the ownership of all files located under a mount. Thus, ID-mapped mounts make it possible to change ownership in a temporary and localized way. It is a localized change because the ownership changes are visible only via a specific mount. All other users and locations where the filesystem is exposed are unaffected. It is a temporary change because the ownership changes are tied to the lifetime of the mount.

Whenever callers interact with the filesystem through an ID-mapped mount, the ID mapping of the mount will be applied to user and group IDs associated with filesystem objects. This encompasses the user and group IDs associated with inodes and also the following *xattr*(7) keys:

- *security.capability*, whenever filesystem capabilities are stored or returned in the **VFS_CAP_REVISION_3** format, which stores a root user ID alongside the capabilities (see *capabilities*(7)).
- *system.posix_acl_access* and *system.posix_acl_default*, whenever user IDs or group IDs are stored in **ACL_USER** or **ACL_GROUP** entries.

The following conditions must be met in order to create an ID-mapped mount:

- The caller must have the **CAP_SYS_ADMIN** capability in the user namespace the filesystem was mounted in.
- The underlying filesystem must support ID-mapped mounts. Currently, the following filesystems support ID-mapped mounts:

  - *xfs*(5) (since Linux 5.12)
  - *ext4*(5) (since Linux 5.12)
  - **FAT** (since Linux 5.12)
  - *btrfs*(5) (since Linux 5.15)
  - **ntfs3** (since Linux 5.15)
  - **f2fs** (since Linux 5.18)
  - **erofs** (since Linux 5.19)
  - **overlayfs** (ID-mapped lower and upper layers supported since Linux 5.19)
- The mount must not already be ID-mapped. This also implies that the ID mapping of a mount cannot be altered.
- The mount must not have any writers.
- The mount must be a detached mount; that is, it must have been created by calling *open_tree*(2) with the **OPEN_TREE_CLONE** flag and it must not already have been visible in a mount namespace. (To put things another way: the mount must not have been attached to the filesystem hierarchy with a system call such as *move_mount*(2)

ID mappings can be created for user IDs, group IDs, and project IDs. An ID mapping is essentially a mapping of a range of user or group IDs into another or the same range of user or group IDs. ID mappings are written to map files as three numbers separated by white space. The first two numbers specify the starting user or group ID in each of the two user namespaces. The third number specifies the range of the ID mapping. For example, a mapping for user IDs such as "1000 1001 1" would indicate that user ID 1000 in the caller's user namespace is mapped to user ID 1001 in its ancestor user namespace. Since the map range is 1, only user ID 1000 is mapped.

It is possible to specify up to 340 ID mappings for each ID mapping type. If any user IDs or group IDs are not mapped, all files owned by that unmapped user or group ID will appear as being owned by the overflow user ID or overflow group ID respectively.

Further details on setting up ID mappings can be found in *user_namespaces*(7).

In the common case, the user namespace passed in *userns_fd* (together with **MOUNT_ATTR_IDMAP** in *attr_set*) to create an ID-mapped mount will be the user namespace of a container. In other scenarios it will be a dedicated user namespace associated with a user's login session as is the case for portable home directories in *systemd-homed.service*(8)). It is also perfectly fine to create a dedicated user namespace for the sake of ID mapping a mount.

ID-mapped mounts can be useful in the following and a variety of other scenarios:

- Sharing files or filesystems between multiple users or multiple machines, especially in complex scenarios. For example, ID-mapped mounts are used to implement portable home directories in *systemd-homed.service*(8), where they allow users to move their home directory to an external storage device and use it on multiple computers where they are assigned different user IDs and group IDs. This effectively makes it possible to assign random user IDs and group IDs at login time.
- Sharing files or filesystems from the host with unprivileged containers. This allows a user to avoid having to change ownership permanently through *chown*(2).
- ID mapping a container's root filesystem. Users don't need to change ownership permanently through *chown*(2). Especially for large root filesystems, using *chown*(2) can be prohibitively expensive.
- Sharing files or filesystems between containers with non-overlapping ID mappings.
- Implementing discretionary access (DAC) permission checking for filesystems lacking a concept of ownership.
- Efficiently changing ownership on a per-mount basis. In contrast to *chown*(2), changing ownership of large sets of files is instantaneous with ID-mapped mounts. This is especially useful when ownership of an entire root filesystem of a virtual machine or container is to be changed as mentioned above. With ID-mapped mounts, a single **mount_setattr**() system call will be sufficient to change the ownership of all files.
- Taking the current ownership into account. ID mappings specify precisely what a user or group ID is supposed to be mapped to. This contrasts with the *chown*(2) system call which cannot by itself take the current ownership of the files it changes into account. It simply changes the ownership to

the specified user ID and group ID.

- Locally and temporarily restricted ownership changes. ID-mapped mounts make it possible to change ownership locally, restricting the ownership changes to specific mounts, and temporarily as the ownership changes only apply as long as the mount exists. By contrast, changing ownership via the *chown*(2) system call changes the ownership globally and permanently.

Extensibility

In order to allow for future extensibility, **mount_setattr**() requires the user-space application to specify the size of the *mount_attr* structure that it is passing. By providing this information, it is possible for **mount_setattr**() to provide both forwards- and backwards-compatibility, with *size* acting as an implicit version number. (Because new extension fields will always be appended, the structure size will always increase.) This extensibility design is very similar to other system calls such as *perf_setattr*(2), *perf_event_open*(2), *clone3*(2) and *openat2*(2).

Let *usize* be the size of the structure as specified by the user-space application, and let *ksize* be the size of the structure which the kernel supports, then there are three cases to consider:

- If *ksize* equals *usize*, then there is no version mismatch and *attr* can be used verbatim.
- If *ksize* is larger than *usize*, then there are some extension fields that the kernel supports which the user-space application is unaware of. Because a zero value in any added extension field signifies a no-op, the kernel treats all of the extension fields not provided by the user-space application as having zero values. This provides backwards-compatibility.
- If *ksize* is smaller than *usize*, then there are some extension fields which the user-space application is aware of but which the kernel does not support. Because any extension field must have its zero values signify a no-op, the kernel can safely ignore the unsupported extension fields if they are all zero. If any unsupported extension fields are non-zero, then −1 is returned and *errno* is set to **E2BIG**. This provides forwards-compatibility.

Because the definition of *struct mount_attr* may change in the future (with new fields being added when system headers are updated), user-space applications should zero-fill *struct mount_attr* to ensure that recompiling the program with new headers will not result in spurious errors at run time. The simplest way is to use a designated initializer:

```
struct mount_attr attr = {
    .attr_set = MOUNT_ATTR_RDONLY,
    .attr_clr = MOUNT_ATTR_NODEV
};
```

Alternatively, the structure can be zero-filled using *memset*(3) or similar functions:

```
struct mount_attr attr;
memset(&attr, 0, sizeof(attr));
attr.attr_set = MOUNT_ATTR_RDONLY;
attr.attr_clr = MOUNT_ATTR_NODEV;
```

A user-space application that wishes to determine which extensions the running kernel supports can do so by conducting a binary search on *size* with a structure which has every byte nonzero (to find the largest value which doesn't produce an error of **E2BIG**).

EXAMPLES

```
/*
 * This program allows the caller to create a new detached mount
 * and set various properties on it.
 */
#define _GNU_SOURCE
#include <err.h>
#include <fcntl.h>
#include <getopt.h>
#include <linux/mount.h>
#include <linux/types.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

static inline int
mount_setattr(int dirfd, const char *pathname, unsigned int flags,
              struct mount_attr *attr, size_t size)
{
    return syscall(SYS_mount_setattr, dirfd, pathname, flags,
                   attr, size);
}

static inline int
open_tree(int dirfd, const char *filename, unsigned int flags)
{
    return syscall(SYS_open_tree, dirfd, filename, flags);
}

static inline int
move_mount(int from_dirfd, const char *from_pathname,
           int to_dirfd, const char *to_pathname, unsigned int flags)
{
    return syscall(SYS_move_mount, from_dirfd, from_pathname,
                   to_dirfd, to_pathname, flags);
}

static const struct option longopts[] = {
    {"map-mount",       required_argument,  NULL,  'a'},
    {"recursive",       no_argument,        NULL,  'b'},
    {"read-only",       no_argument,        NULL,  'c'},
    {"block-setid",     no_argument,        NULL,  'd'},
    {"block-devices",   no_argument,        NULL,  'e'},
    {"block-exec",      no_argument,        NULL,  'f'},
    {"no-access-time",  no_argument,        NULL,  'g'},
    { NULL,             0,                  NULL,   0 },
};

int
main(int argc, char *argv[])
{
    int                fd_userns = -1;
    int                fd_tree;
    int                index = 0;
    int                ret;
    bool               recursive = false;
    const char         *source;
    const char         *target;
    struct mount_attr  *attr = &(struct mount_attr){};

    while ((ret = getopt_long_only(argc, argv, "",
                                   longopts, &index)) != -1) {
        switch (ret) {
        case 'a':
            fd_userns = open(optarg, O_RDONLY | O_CLOEXEC);
            if (fd_userns == -1)
                err(EXIT_FAILURE, "open(%s)", optarg);
            break;
        case 'b':
            recursive = true;
```

```
            break;
        case 'c':
            attr->attr_set |= MOUNT_ATTR_RDONLY;
            break;
        case 'd':
            attr->attr_set |= MOUNT_ATTR_NOSUID;
            break;
        case 'e':
            attr->attr_set |= MOUNT_ATTR_NODEV;
            break;
        case 'f':
            attr->attr_set |= MOUNT_ATTR_NOEXEC;
            break;
        case 'g':
            attr->attr_set |= MOUNT_ATTR_NOATIME;
            attr->attr_clr |= MOUNT_ATTR__ATIME;
            break;
        default:
            errx(EXIT_FAILURE, "Invalid argument specified");
        }
    }

    if ((argc - optind) < 2)
        errx(EXIT_FAILURE, "Missing source or target mount point");

    source = argv[optind];
    target = argv[optind + 1];

    /* In the following, -1 as the 'dirfd' argument ensures that
       open_tree() fails if 'source' is not an absolute pathname. */

    fd_tree = open_tree(-1, source,
                        OPEN_TREE_CLONE | OPEN_TREE_CLOEXEC |
                        AT_EMPTY_PATH | (recursive ? AT_RECURSIVE : 0));
    if (fd_tree == -1)
        err(EXIT_FAILURE, "open(%s)", source);

    if (fd_userns >= 0) {
        attr->attr_set  |= MOUNT_ATTR_IDMAP;
        attr->userns_fd = fd_userns;
    }

    ret = mount_setattr(fd_tree, "",
                        AT_EMPTY_PATH | (recursive ? AT_RECURSIVE : 0),
                        attr, sizeof(struct mount_attr));
    if (ret == -1)
        err(EXIT_FAILURE, "mount_setattr");

    close(fd_userns);

    /* In the following, -1 as the 'to_dirfd' argument ensures that
       open_tree() fails if 'target' is not an absolute pathname. */

    ret = move_mount(fd_tree, "", -1, target,
                     MOVE_MOUNT_F_EMPTY_PATH);
    if (ret == -1)
        err(EXIT_FAILURE, "move_mount() to %s", target);

    close(fd_tree);
```

```
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

newgidmap(1), *newuidmap*(1), *clone*(2), *mount*(2), *unshare*(2), *proc*(5), *capabilities*(7), *mount_namespaces*(7), *user_namespaces*(7), *xattr*(7)

NAME

move_pages − move individual pages of a process to another node

LIBRARY

NUMA (Non-Uniform Memory Access) policy library (*libnuma*, −*lnuma*)

SYNOPSIS

#include <numaif.h>

long move_pages(int *pid***, unsigned long** *count***, void *** *pages***[.***count***],**
const int *nodes***[.***count***], int** *status***[.***count***], int** *flags***);**

DESCRIPTION

move_pages() moves the specified *pages* of the process *pid* to the memory nodes specified by *nodes*. The result of the move is reflected in *status*. The *flags* indicate constraints on the pages to be moved.

pid is the ID of the process in which pages are to be moved. If *pid* is 0, then **move_pages**() moves pages of the calling process.

To move pages in another process requires the following privileges:

- Up to and including Linux 4.12: the caller must be privileged (**CAP_SYS_NICE**) or the real or effective user ID of the calling process must match the real or saved-set user ID of the target process.
- The older rules allowed the caller to discover various virtual address choices made by the kernel that could lead to the defeat of address-space-layout randomization for a process owned by the same UID as the caller, the rules were changed starting with Linux 4.13. Since Linux 4.13, permission is governed by a ptrace access mode **PTRACE_MODE_READ_REALCREDS** check with respect to the target process; see *ptrace*(2).

count is the number of pages to move. It defines the size of the three arrays *pages*, *nodes*, and *status*.

pages is an array of pointers to the pages that should be moved. These are pointers that should be aligned to page boundaries. Addresses are specified as seen by the process specified by *pid*.

nodes is an array of integers that specify the desired location for each page. Each element in the array is a node number. *nodes* can also be NULL, in which case **move_pages**() does not move any pages but instead will return the node where each page currently resides, in the *status* array. Obtaining the status of each page may be necessary to determine pages that need to be moved.

status is an array of integers that return the status of each page. The array contains valid values only if **move_pages**() did not return an error. Preinitialization of the array to a value which cannot represent a real numa node or valid error of status array could help to identify pages that have been migrated.

flags specify what types of pages to move. **MPOL_MF_MOVE** means that only pages that are in exclusive use by the process are to be moved. **MPOL_MF_MOVE_ALL** means that pages shared between multiple processes can also be moved. The process must be privileged (**CAP_SYS_NICE**) to use **MPOL_MF_MOVE_ALL**.

Page states in the status array

The following values can be returned in each element of the *status* array.

0..MAX_NUMNODES
: Identifies the node on which the page resides.

−EACCES
: The page is mapped by multiple processes and can be moved only if **MPOL_MF_MOVE_ALL** is specified.

−EBUSY
: The page is currently busy and cannot be moved. Try again later. This occurs if a page is undergoing I/O or another kernel subsystem is holding a reference to the page.

−EFAULT
: This is a zero page or the memory area is not mapped by the process.

−EIO Unable to write back a page. The page has to be written back in order to move it since the page is dirty and the filesystem does not provide a migration function that would allow the move of dirty pages.

−EINVAL
A dirty page cannot be moved. The filesystem does not provide a migration function and has no ability to write back pages.

−ENOENT
The page is not present.

−ENOMEM
Unable to allocate memory on target node.

RETURN VALUE

On success **move_pages**() returns zero. On error, it returns −1, and sets *errno* to indicate the error. If positive value is returned, it is the number of nonmigrated pages.

ERRORS

Positive value
The number of nonmigrated pages if they were the result of nonfatal reasons (since Linux 4.17).

E2BIG Too many pages to move. Since Linux 2.6.29, the kernel no longer generates this error.

EACCES
One of the target nodes is not allowed by the current cpuset.

EFAULT
Parameter array could not be accessed.

EINVAL
Flags other than **MPOL_MF_MOVE** and **MPOL_MF_MOVE_ALL** was specified or an attempt was made to migrate pages of a kernel thread.

ENODEV
One of the target nodes is not online.

EPERM
The caller specified **MPOL_MF_MOVE_ALL** without sufficient privileges (**CAP_SYS_NICE**). Or, the caller attempted to move pages of a process belonging to another user but did not have privilege to do so (**CAP_SYS_NICE**).

ESRCH
Process does not exist.

STANDARDS

Linux.

HISTORY

Linux 2.6.18.

NOTES

For information on library support, see *numa*(7).

Use *get_mempolicy*(2) with the **MPOL_F_MEMS_ALLOWED** flag to obtain the set of nodes that are allowed by the current cpuset. Note that this information is subject to change at any time by manual or automatic reconfiguration of the cpuset.

Use of this function may result in pages whose location (node) violates the memory policy established for the specified addresses (See *mbind*(2)) and/or the specified process (See *set_mempolicy*(2)). That is, memory policy does not constrain the destination nodes used by **move_pages**().

The <*numaif.h*> header is not included with glibc, but requires installing *libnuma−devel* or a similar package.

SEE ALSO

get_mempolicy(2), *mbind*(2), *set_mempolicy*(2), *numa*(3), *numa_maps*(5), *cpuset*(7), *numa*(7), *migratepages*(8), *numastat*(8)

NAME

mprotect, pkey_mprotect − set protection on a region of memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

int mprotect(void *addr***[.***len***], size_t** *len***, int** *prot***);**

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_mprotect(void *addr***[.***len***], size_t** *len***, int** *prot***, int** *pkey***);**

DESCRIPTION

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr*+*len*−1]. *addr* must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a **SIGSEGV** signal for the process.

prot is a combination of the following access flags: **PROT_NONE** or a bitwise OR of the other values in the following list:

PROT_NONE
The memory cannot be accessed at all.

PROT_READ
The memory can be read.

PROT_WRITE
The memory can be modified.

PROT_EXEC
The memory can be executed.

PROT_SEM (since Linux 2.5.7)
The memory can be used for atomic operations. This flag was introduced as part of the *futex*(2) implementation (in order to guarantee the ability to perform atomic operations required by commands such as **FUTEX_WAIT**), but is not currently used in on any architecture.

PROT_SAO (since Linux 2.6.26)
The memory should have strong access ordering. This feature is specific to the PowerPC architecture (version 2.06 of the architecture specification adds the SAO CPU feature, and it is available on POWER 7 or PowerPC A2, for example).

Additionally (since Linux 2.6.0), *prot* can have one of the following flags set:

PROT_GROWSUP
Apply the protection mode up to the end of a mapping that grows upwards. (Such mappings are created for the stack area on architectures—for example, HP-PARISC—that have an upwardly growing stack.)

PROT_GROWSDOWN
Apply the protection mode down to the beginning of a mapping that grows downward (which should be a stack segment or a segment mapped with the **MAP_GROWSDOWN** flag set).

Like **mprotect**(), **pkey_mprotect**() changes the protection on the pages specified by *addr* and *len*. The *pkey* argument specifies the protection key (see *pkeys*(7)) to assign to the memory. The protection key must be allocated with *pkey_alloc*(2) before it is passed to **pkey_mprotect**(). For an example of the use of this system call, see *pkeys*(7).

RETURN VALUE

On success, **mprotect**() and **pkey_mprotect**() return zero. On error, these system calls return −1, and *errno* is set to indicate the error.

ERRORS

EACCES
The memory cannot be given the specified access. This can happen, for example, if you *mmap*(2) a file to which you have read-only access, then ask **mprotect**() to mark it **PROT_WRITE**.

EINVAL
addr is not a valid pointer, or not a multiple of the system page size.

EINVAL
(**pkey_mprotect**()) *pkey* has not been allocated with *pkey_alloc*(2)

EINVAL
Both **PROT_GROWSUP** and **PROT_GROWSDOWN** were specified in *prot*.

EINVAL
Invalid flags specified in *prot*.

EINVAL
(PowerPC architecture) **PROT_SAO** was specified in *prot*, but SAO hardware feature is not available.

ENOMEM
Internal kernel structures could not be allocated.

ENOMEM
Addresses in the range [*addr*, *addr*+*len*−1] are invalid for the address space of the process, or specify one or more pages that are not mapped. (Before Linux 2.4.19, the error **EFAULT** was incorrectly produced for these cases.)

ENOMEM
Changing the protection of a memory region would result in the total number of mappings with distinct attributes (e.g., read versus read/write protection) exceeding the allowed maximum. (For example, making the protection of a range **PROT_READ** in the middle of a region currently protected as **PROT_READ|PROT_WRITE** would result in three mappings: two read/write mappings at each end and a read-only mapping in the middle.)

VERSIONS

POSIX says that the behavior of **mprotect**() is unspecified if it is applied to a region of memory that was not obtained via *mmap*(2).

On Linux, it is always permissible to call **mprotect**() on any address in a process's address space (except for the kernel vsyscall area). In particular, it can be used to change existing code mappings to be writable.

Whether **PROT_EXEC** has any effect different from **PROT_READ** depends on processor architecture, kernel version, and process state. If **READ_IMPLIES_EXEC** is set in the process's personality flags (see *personality*(2)), specifying **PROT_READ** will implicitly add **PROT_EXEC**.

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**.

POSIX.1 says that an implementation may permit access other than that specified in *prot*, but at a minimum can allow write access only if **PROT_WRITE** has been set, and must not allow any access if **PROT_NONE** has been set.

Applications should be careful when mixing use of **mprotect**() and **pkey_mprotect**(). On x86, when **mprotect**() is used with *prot* set to **PROT_EXEC** a pkey may be allocated and set on the memory implicitly by the kernel, but only when the pkey was 0 previously.

On systems that do not support protection keys in hardware, **pkey_mprotect**() may still be used, but *pkey* must be set to −1. When called this way, the operation of **pkey_mprotect**() is equivalent to **mprotect**().

STANDARDS

mprotect()
POSIX.1-2008.

pkey_mprotect()
Linux.

HISTORY

mprotect()
POSIX.1-2001, SVr4.

pkey_mprotect()
Linux 4.9, glibc 2.27.

NOTES

EXAMPLES

The program below demonstrates the use of **mprotect**(). The program allocates four pages of memory, makes the third of these pages read-only, and then executes a loop that walks upward through the allocated region modifying bytes.

An example of what we might see when running the program is the following:

```
$ ./a.out
Start of region:        0x804c000
Got SIGSEGV at address: 0x804e000
```

Program source

```
#include <malloc.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static char *buffer;

static void
handler(int sig, siginfo_t *si, void *unused)
{
    /* Note: calling printf() from a signal handler is not safe
       (and should not be done in production programs), since
       printf() is not async-signal-safe; see signal-safety(7).
       Nevertheless, we use printf() here as a simple way of
       showing that the handler was called. */

    printf("Got SIGSEGV at address: %p\n", si->si_addr);
    exit(EXIT_FAILURE);
}

int
main(void)
{
    int               pagesize;
    struct sigaction  sa;

    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handler;
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        handle_error("sigaction");

    pagesize = sysconf(_SC_PAGE_SIZE);
    if (pagesize == -1)
        handle_error("sysconf");
```

```
        /* Allocate a buffer aligned on a page boundary;
           initial protection is PROT_READ | PROT_WRITE. */

        buffer = memalign(pagesize, 4 * pagesize);
        if (buffer == NULL)
            handle_error("memalign");

        printf("Start of region:        %p\n", buffer);

        if (mprotect(buffer + pagesize * 2, pagesize,
                     PROT_READ) == -1)
            handle_error("mprotect");

        for (char *p = buffer ; ; )
            *(p++) = 'a';

        printf("Loop completed\n");     /* Should never happen */
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

mmap(2), *sysconf*(3), *pkeys*(7)

NAME

mq_getsetattr − get/set message queue attributes

SYNOPSIS

#include <mqueue.h> /* Definition of **struct mq_attr** */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_mq_getsetattr, mqd_t *mqdes***,**
const struct mq_attr *****newattr*, struct mq_attr *****oldattr***);**

DESCRIPTION

Do not use this system call.

This is the low-level system call used to implement *mq_getattr*(3) and *mq_setattr*(3). For an explanation of how this system call operates, see the description of *mq_setattr*(3).

STANDARDS

None.

NOTES

Never call it unless you are writing a C library!

SEE ALSO

mq_getattr(3), *mq_overview*(7)

NAME

mremap − remap a virtual memory address

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/mman.h>

void *mremap(void old_address[.old_size], size_t old_size,
             size_t new_size, int flags, ... /* void *new_address */);
```

DESCRIPTION

mremap() expands (or shrinks) an existing memory mapping, potentially moving it at the same time (controlled by the *flags* argument and the available virtual address space).

old_address is the old address of the virtual memory block that you want to expand (or shrink). Note that *old_address* has to be page aligned. *old_size* is the old size of the virtual memory block. *new_size* is the requested size of the virtual memory block after the resize. An optional fifth argument, *new_address*, may be provided; see the description of **MREMAP_FIXED** below.

If the value of *old_size* is zero, and *old_address* refers to a shareable mapping (see the description of **MAP_SHARED** in *mmap*(2)), then **mremap**() will create a new mapping of the same pages. *new_size* will be the size of the new mapping and the location of the new mapping may be specified with *new_address*; see the description of **MREMAP_FIXED** below. If a new mapping is requested via this method, then the **MREMAP_MAYMOVE** flag must also be specified.

The *flags* bit-mask argument may be 0, or include the following flags:

MREMAP_MAYMOVE
: By default, if there is not sufficient space to expand a mapping at its current location, then **mremap**() fails. If this flag is specified, then the kernel is permitted to relocate the mapping to a new virtual address, if necessary. If the mapping is relocated, then absolute pointers into the old mapping location become invalid (offsets relative to the starting address of the mapping should be employed).

MREMAP_FIXED (since Linux 2.3.31)
: This flag serves a similar purpose to the **MAP_FIXED** flag of *mmap*(2). If this flag is specified, then **mremap**() accepts a fifth argument, *void *new_address*, which specifies a page-aligned address to which the mapping must be moved. Any previous mapping at the address range specified by *new_address* and *new_size* is unmapped.

  If **MREMAP_FIXED** is specified, then **MREMAP_MAYMOVE** must also be specified.

MREMAP_DONTUNMAP (since Linux 5.7)
: This flag, which must be used in conjunction with **MREMAP_MAYMOVE**, remaps a mapping to a new address but does not unmap the mapping at *old_address*.

  The **MREMAP_DONTUNMAP** flag can be used only with private anonymous mappings (see the description of **MAP_PRIVATE** and **MAP_ANONYMOUS** in *mmap*(2)).

  After completion, any access to the range specified by *old_address* and *old_size* will result in a page fault. The page fault will be handled by a *userfaultfd*(2) handler if the address is in a range previously registered with *userfaultfd*(2). Otherwise, the kernel allocates a zero-filled page to handle the fault.

  The **MREMAP_DONTUNMAP** flag may be used to atomically move a mapping while leaving the source mapped. See NOTES for some possible applications of **MREMAP_DONTUNMAP**.

If the memory segment specified by *old_address* and *old_size* is locked (using *mlock*(2) or similar), then this lock is maintained when the segment is resized and/or relocated. As a consequence, the amount of memory locked by the process may change.

RETURN VALUE

On success **mremap**() returns a pointer to the new virtual memory area. On error, the value **MAP_FAILED** (that is, *(void *) −1*) is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
The caller tried to expand a memory segment that is locked, but this was not possible without exceeding the **RLIMIT_MEMLOCK** resource limit.

EFAULT
Some address in the range *old_address* to *old_address*+*old_size* is an invalid virtual memory address for this process. You can also get **EFAULT** even if there exist mappings that cover the whole address space requested, but those mappings are of different types.

EINVAL
An invalid argument was given. Possible causes are:

- *old_address* was not page aligned;
- a value other than **MREMAP_MAYMOVE** or **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified in *flags*;
- *new_size* was zero;
- *new_size* or *new_address* was invalid;
- the new address range specified by *new_address* and *new_size* overlapped the old address range specified by *old_address* and *old_size*;
- **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified without also specifying **MREMAP_MAYMOVE**;
- **MREMAP_DONTUNMAP** was specified, but one or more pages in the range specified by *old_address* and *old_size* were not private anonymous;
- **MREMAP_DONTUNMAP** was specified and *old_size* was not equal to *new_size*;
- *old_size* was zero and *old_address* does not refer to a shareable mapping (but see BUGS);
- *old_size* was zero and the **MREMAP_MAYMOVE** flag was not specified.

ENOMEM
Not enough memory was available to complete the operation. Possible causes are:

- The memory area cannot be expanded at the current virtual address, and the **MREMAP_MAYMOVE** flag is not set in *flags*. Or, there is not enough (virtual) memory available.
- **MREMAP_DONTUNMAP** was used causing a new mapping to be created that would exceed the (virtual) memory available. Or, it would exceed the maximum number of allowed mappings.

STANDARDS
Linux.

HISTORY
Prior to glibc 2.4, glibc did not expose the definition of **MREMAP_FIXED**, and the prototype for **mremap**() did not allow for the *new_address* argument.

NOTES
mremap() changes the mapping between virtual addresses and memory pages. This can be used to implement a very efficient *realloc*(3).

In Linux, memory is divided into pages. A process has (one or) several linear virtual memory segments. Each virtual memory segment has one or more mappings to real memory pages (in the page table). Each virtual memory segment has its own protection (access rights), which may cause a segmentation violation (**SIGSEGV**) if the memory is accessed incorrectly (e.g., writing to a read-only segment). Accessing virtual memory outside of the segments will also cause a segmentation violation.

If **mremap**() is used to move or expand an area locked with *mlock*(2) or equivalent, the **mremap**() call will make a best effort to populate the new area but will not fail with **ENOMEM** if the area cannot be populated.

MREMAP_DONTUNMAP use cases

Possible applications for **MREMAP_DONTUNMAP** include:

- Non-cooperative *userfaultfd*(2): an application can yank out a virtual address range using **MREMAP_DONTUNMAP** and then employ a *userfaultfd*(2) handler to handle the page faults that subsequently occur as other threads in the process touch pages in the yanked range.
- Garbage collection: **MREMAP_DONTUNMAP** can be used in conjunction with *userfaultfd*(2) to implement garbage collection algorithms (e.g., in a Java virtual machine). Such an implementation can be cheaper (and simpler) than conventional garbage collection techniques that involve marking pages with protection **PROT_NONE** in conjunction with the use of a **SIGSEGV** handler to catch accesses to those pages.

BUGS

Before Linux 4.14, if *old_size* was zero and the mapping referred to by *old_address* was a private mapping (see the description of **MAP_PRIVATE** in *mmap*(2)), **mremap**() created a new private mapping unrelated to the original mapping. This behavior was unintended and probably unexpected in user-space applications (since the intention of **mremap**() is to create a new mapping based on the original mapping). Since Linux 4.14, **mremap**() fails with the error **EINVAL** in this scenario.

SEE ALSO

brk(2), *getpagesize*(2), *getrlimit*(2), *mlock*(2), *mmap*(2), *sbrk*(2), *malloc*(3), *realloc*(3)

Your favorite text book on operating systems for more information on paged memory (e.g., *Modern Operating Systems* by Andrew S. Tanenbaum, *Inside Linux* by Randolph Bentson, *The Design of the UNIX Operating System* by Maurice J. Bach)

NAME

msgctl − System V message control operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/msg.h>

int msgctl(int *msqid***, int** *op***, struct msqid_ds *****buf***);**

DESCRIPTION

msgctl() performs the control operation specified by *op* on the System V message queue with identifier *msqid*.

The *msqid_ds* data structure is defined in <*sys/msg.h*> as follows:

```
struct msqid_ds {
    struct ipc_perm msg_perm;   /* Ownership and permissions */
    time_t          msg_stime;  /* Time of last msgsnd(2) */
    time_t          msg_rtime;  /* Time of last msgrcv(2) */
    time_t          msg_ctime;  /* Time of creation or last
                                   modification by msgctl() */
    unsigned long   msg_cbytes; /* # of bytes in queue */
    msgqnum_t       msg_qnum;   /* # number of messages in queue */
    msglen_t        msg_qbytes; /* Maximum # of bytes in queue */
    pid_t           msg_lspid;  /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;  /* PID of last msgrcv(2) */
};
```

The fields of the *msqid_ds* structure are as follows:

msg_perm This is an *ipc_perm* structure (see below) that specifies the access permissions on the message queue.

msg_stime Time of the last *msgsnd*(2) system call.

msg_rtime Time of the last *msgrcv*(2) system call.

msg_ctime Time of creation of queue or time of last **msgctl**() **IPC_SET** operation.

msg_cbytes Number of bytes in all messages currently on the message queue. This is a nonstandard Linux extension that is not specified in POSIX.

msg_qnum Number of messages currently on the message queue.

msg_qbytes Maximum number of bytes of message text allowed on the message queue.

msg_lspid ID of the process that performed the last *msgsnd*(2) system call.

msg_lrpid ID of the process that performed the last *msgrcv*(2) system call.

The *ipc_perm* structure is defined as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
    key_t          __key;       /* Key supplied to msgget(2) */
    uid_t          uid;         /* Effective UID of owner */
    gid_t          gid;         /* Effective GID of owner */
    uid_t          cuid;        /* Effective UID of creator */
    gid_t          cgid;        /* Effective GID of creator */
    unsigned short mode;        /* Permissions */
    unsigned short __seq;       /* Sequence number */
};
```

The least significant 9 bits of the *mode* field of the *ipc_perm* structure define the access permissions for the message queue. The permission bits are as follows:

0400 Read by user
0200 Write by user
0040 Read by group

0020 Write by group
0004 Read by others
0002 Write by others

Bits 0100, 0010, and 0001 (the execute bits) are unused by the system.

Valid values for *op* are:

IPC_STAT
Copy information from the kernel data structure associated with *msqid* into the *msqid_ds* structure pointed to by *buf*. The caller must have read permission on the message queue.

IPC_SET
Write the values of some members of the *msqid_ds* structure pointed to by *buf* to the kernel data structure associated with this message queue, updating also its *msg_ctime* member.

The following members of the structure are updated: *msg_qbytes*, *msg_perm.uid*, *msg_perm.gid*, and (the least significant 9 bits of) *msg_perm.mode*.

The effective UID of the calling process must match the owner (*msg_perm.uid*) or creator (*msg_perm.cuid*) of the message queue, or the caller must be privileged. Appropriate privilege (Linux: the **CAP_SYS_RESOURCE** capability) is required to raise the *msg_qbytes* value beyond the system parameter **MSGMNB**.

IPC_RMID
Immediately remove the message queue, awakening all waiting reader and writer processes (with an error return and *errno* set to **EIDRM**). The calling process must have appropriate privileges or its effective user ID must be either that of the creator or owner of the message queue. The third argument to **msgctl**() is ignored in this case.

IPC_INFO (Linux-specific)
Return information about system-wide message queue limits and parameters in the structure pointed to by *buf*. This structure is of type *msginfo* (thus, a cast is required), defined in <*sys/msg.h*> if the **_GNU_SOURCE** feature test macro is defined:

```
struct msginfo {
    int msgpool; /* Size in kibibytes of buffer pool
                    used to hold message data;
                    unused within kernel */
    int msgmap;  /* Maximum number of entries in message
                    map; unused within kernel */
    int msgmax;  /* Maximum number of bytes that can be
                    written in a single message */
    int msgmnb;  /* Maximum number of bytes that can be
                    written to queue; used to initialize
                    msg_qbytes during queue creation
                    (msgget(2)) */
    int msgmni;  /* Maximum number of message queues */
    int msgssz;  /* Message segment size;
                    unused within kernel */
    int msgtql;  /* Maximum number of messages on all queues
                    in system; unused within kernel */
    unsigned short msgseg;
                 /* Maximum number of segments;
                    unused within kernel */
};
```

The *msgmni*, *msgmax*, and *msgmnb* settings can be changed via */proc* files of the same name; see *proc*(5) for details.

MSG_INFO (Linux-specific)
Return a *msginfo* structure containing the same information as for **IPC_INFO**, except that the following fields are returned with information about system resources consumed by message queues: the *msgpool* field returns the number of message queues that currently exist on the system; the *msgmap* field returns the total number of messages in all queues on the system;

and the *msgtql* field returns the total number of bytes in all messages in all queues on the system.

MSG_STAT (Linux-specific)
Return a *msqid_ds* structure as for **IPC_STAT**. However, the *msqid* argument is not a queue identifier, but instead an index into the kernel's internal array that maintains information about all message queues on the system.

MSG_STAT_ANY (Linux-specific, since Linux 4.17)
Return a *msqid_ds* structure as for **MSG_STAT**. However, *msg_perm.mode* is not checked for read access for *msqid* meaning that any user can employ this operation (just as any user may read */proc/sysvipc/msg* to obtain the same information).

RETURN VALUE

On success, **IPC_STAT**, **IPC_SET**, and **IPC_RMID** return 0. A successful **IPC_INFO** or **MSG_INFO** operation returns the index of the highest used entry in the kernel's internal array recording information about all message queues. (This information can be used with repeated **MSG_STAT** or **MSG_STAT_ANY** operations to obtain information about all queues on the system.) A successful **MSG_STAT** or **MSG_STAT_ANY** operation returns the identifier of the queue whose index was given in *msqid*.

On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
The argument *op* is equal to **IPC_STAT** or **MSG_STAT**, but the calling process does not have read permission on the message queue *msqid*, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EFAULT
The argument *op* has the value **IPC_SET** or **IPC_STAT**, but the address pointed to by *buf* isn't accessible.

EIDRM
The message queue was removed.

EINVAL
Invalid value for *op* or *msqid*. Or: for a **MSG_STAT** operation, the index value specified in *msqid* referred to an array slot that is currently unused.

EPERM
The argument *op* has the value **IPC_SET** or **IPC_RMID**, but the effective user ID of the calling process is not the creator (as found in *msg_perm.cuid*) or the owner (as found in *msg_perm.uid*) of the message queue, and the caller is not privileged (Linux: does not have the **CAP_SYS_ADMIN** capability).

EPERM
An attempt (**IPC_SET**) was made to increase *msg_qbytes* beyond the system parameter **MSGMNB**, but the caller is not privileged (Linux: does not have the **CAP_SYS_RESOURCE** capability).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

Various fields in the *struct msqid_ds* were typed as *short* under Linux 2.2 and have become *long* under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by an **IPC_64** flag in *op*.)

NOTES

The **IPC_INFO**, **MSG_STAT**, and **MSG_INFO** operations are used by the *ipcs*(1) program to provide information on allocated resources. In the future these may modified or moved to a */proc* filesystem interface.

SEE ALSO

msgget(2), *msgrcv*(2), *msgsnd*(2), *capabilities*(7), *mq_overview*(7), *sysvipc*(7)

NAME

msgget − get a System V message queue identifier

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/msg.h>

int msgget(key_t *key***, int** *msgflg***);**

DESCRIPTION

The **msgget**() system call returns the System V message queue identifier associated with the value of the *key* argument. It may be used either to obtain the identifier of a previously created message queue (when *msgflg* is zero and *key* does not have the value **IPC_PRIVATE**), or to create a new set.

A new message queue is created if *key* has the value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no message queue with the given key *key* exists, and **IPC_CREAT** is specified in *msgflg*.

If *msgflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a message queue already exists for *key*, then **msgget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for *open*(2).)

Upon creation, the least significant bits of the argument *msgflg* define the permissions of the message queue. These permission bits have the same format and semantics as the permissions specified for the *mode* argument of *open*(2). (The execute permissions are not used.)

If a new message queue is created, then its associated data structure *msqid_ds* (see *msgctl*(2)) is initialized as follows:

- *msg_perm.cuid* and *msg_perm.uid* are set to the effective user ID of the calling process.
- *msg_perm.cgid* and *msg_perm.gid* are set to the effective group ID of the calling process.
- The least significant 9 bits of *msg_perm.mode* are set to the least significant 9 bits of *msgflg*.
- *msg_qnum*, *msg_lspid*, *msg_lrpid*, *msg_stime*, and *msg_rtime* are set to 0.
- *msg_ctime* is set to the current time.
- *msg_qbytes* is set to the system limit **MSGMNB**.

If the message queue already exists the permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

On success, **msgget**() returns the message queue identifier (a nonnegative integer). On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
A message queue exists for *key*, but the calling process does not have permission to access the queue, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EEXIST
IPC_CREAT and **IPC_EXCL** were specified in *msgflg*, but a message queue already exists for *key*.

ENOENT
No message queue exists for *key* and *msgflg* did not specify **IPC_CREAT**.

ENOMEM
A message queue has to be created but the system does not have enough memory for the new data structure.

ENOSPC
A message queue has to be created but the system limit for the maximum number of message queues (**MSGMNI**) would be exceeded.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

Linux

Until Linux 2.3.20, Linux would return **EIDRM** for a **msgget**() on a message queue scheduled for deletion.

NOTES

IPC_PRIVATE isn't a flag field but a *key_t* type. If this special value is used for *key*, the system call ignores everything but the least significant 9 bits of *msgflg* and creates a new message queue (on success).

The following is a system limit on message queue resources affecting a **msgget**() call:

MSGMNI

System-wide limit on the number of message queues. Before Linux 3.19, the default value for this limit was calculated using a formula based on available system memory. Since Linux 3.19, the default value is 32,000. On Linux, this limit can be read and modified via */proc/sys/kernel/msgmni*.

BUGS

The name choice **IPC_PRIVATE** was perhaps unfortunate, **IPC_NEW** would more clearly show its function.

SEE ALSO

msgctl(2), *msgrcv*(2), *msgsnd*(2), *ftok*(3), *capabilities*(7), *mq_overview*(7), *sysvipc*(7)

NAME

msgrcv, msgsnd − System V message queue operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/msg.h>

int msgsnd(int *msqid***, const void** *msgp***[.***msgsz***], size_t** *msgsz***,**
int *msgflg***);**

ssize_t msgrcv(int *msqid***, void** *msgp***[.***msgsz***], size_t** *msgsz***, long** *msgtyp***,**
int *msgflg***);**

DESCRIPTION

The **msgsnd**() and **msgrcv**() system calls are used to send messages to, and receive messages from, a System V message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

The *msgp* argument is a pointer to a caller-defined structure of the following general form:

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
};
```

The *mtext* field is an array (or other structure) whose size is specified by *msgsz*, a nonnegative integer value. Messages of zero length (i.e., no *mtext* field) are permitted. The *mtype* field must have a strictly positive integer value. This value can be used by the receiving process for message selection (see the description of **msgrcv**() below).

msgsnd()

The **msgsnd**() system call appends a copy of the message pointed to by *msgp* to the message queue whose identifier is specified by *msqid*.

If sufficient space is available in the queue, **msgsnd**() succeeds immediately. The queue capacity is governed by the *msg_qbytes* field in the associated data structure for the message queue. During queue creation this field is initialized to **MSGMNB** bytes, but this limit can be modified using *msgctl*(2). A message queue is considered to be full if either of the following conditions is true:

- Adding a new message to the queue would cause the total number of bytes in the queue to exceed the queue's maximum size (the *msg_qbytes* field).
- Adding another message to the queue would cause the total number of messages in the queue to exceed the queue's maximum size (the *msg_qbytes* field). This check is necessary to prevent an unlimited number of zero-length messages being placed on the queue. Although such messages contain no data, they nevertheless consume (locked) kernel memory.

If insufficient space is available in the queue, then the default behavior of **msgsnd**() is to block until space becomes available. If **IPC_NOWAIT** is specified in *msgflg*, then the call instead fails with the error **EAGAIN**.

A blocked **msgsnd**() call may also fail if:

- the queue is removed, in which case the system call fails with *errno* set to **EIDRM**; or
- a signal is caught, in which case the system call fails with *errno* set to **EINTR**;**see** *signal*(7). (**msgsnd**() is never automatically restarted after being interrupted by a signal handler, regardless of the setting of the **SA_RESTART** flag when establishing a signal handler.)

Upon successful completion the message queue data structure is updated as follows:

- *msg_lspid* is set to the process ID of the calling process.
- *msg_qnum* is incremented by 1.
- *msg_stime* is set to the current time.

msgrcv()

The **msgrcv**() system call removes a message from the queue specified by *msqid* and places it in the buffer pointed to by *msgp*.

The argument *msgsz* specifies the maximum size in bytes for the member *mtext* of the structure pointed to by the *msgp* argument. If the message text has length greater than *msgsz*, then the behavior depends on whether **MSG_NOERROR** is specified in *msgflg*. If **MSG_NOERROR** is specified, then the message text will be truncated (and the truncated part will be lost); if **MSG_NOERROR** is not specified, then the message isn't removed from the queue and the system call fails returning −1 with *errno* set to **E2BIG**.

Unless **MSG_COPY** is specified in *msgflg* (see below), the *msgtyp* argument specifies the type of message requested, as follows:

- If *msgtyp* is 0, then the first message in the queue is read.
- If *msgtyp* is greater than 0, then the first message in the queue of type *msgtyp* is read, unless **MSG_EXCEPT** was specified in *msgflg*, in which case the first message in the queue of type not equal to *msgtyp* will be read.
- If *msgtyp* is less than 0, then the first message in the queue with the lowest type less than or equal to the absolute value of *msgtyp* will be read.

The *msgflg* argument is a bit mask constructed by ORing together zero or more of the following flags:

IPC_NOWAIT
: Return immediately if no message of the requested type is in the queue. The system call fails with *errno* set to **ENOMSG**.

MSG_COPY (since Linux 3.8)
: Nondestructively fetch a copy of the message at the ordinal position in the queue specified by *msgtyp* (messages are considered to be numbered starting at 0).

  This flag must be specified in conjunction with **IPC_NOWAIT**, with the result that, if there is no message available at the given position, the call fails immediately with the error **ENOMSG**. Because they alter the meaning of *msgtyp* in orthogonal ways, **MSG_COPY** and **MSG_EXCEPT** may not both be specified in *msgflg*.

  The **MSG_COPY** flag was added for the implementation of the kernel checkpoint-restore facility and is available only if the kernel was built with the **CONFIG_CHECKPOINT_RESTORE** option.

MSG_EXCEPT
: Used with *msgtyp* greater than 0 to read the first message in the queue with message type that differs from *msgtyp*.

MSG_NOERROR
: To truncate the message text if longer than *msgsz* bytes.

If no message of the requested type is available and **IPC_NOWAIT** isn't specified in *msgflg*, the calling process is blocked until one of the following conditions occurs:

- A message of the desired type is placed in the queue.
- The message queue is removed from the system. In this case, the system call fails with *errno* set to **EIDRM**.
- The calling process catches a signal. In this case, the system call fails with *errno* set to **EINTR**. (**msgrcv**() is never automatically restarted after being interrupted by a signal handler, regardless of the setting of the **SA_RESTART** flag when establishing a signal handler.)

Upon successful completion the message queue data structure is updated as follows:

msg_lrpid is set to the process ID of the calling process.

msg_qnum is decremented by 1.

msg_rtime is set to the current time.

RETURN VALUE

On success, **msgsnd**() returns 0 and **msgrcv**() returns the number of bytes actually copied into the *mtext* array. On failure, both functions return −1, and set *errno* to indicate the error.

ERRORS

msgsnd() can fail with the following errors:

EACCES
The calling process does not have write permission on the message queue, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EAGAIN
The message can't be sent due to the *msg_qbytes* limit for the queue and **IPC_NOWAIT** was specified in *msgflg*.

EFAULT
The address pointed to by *msgp* isn't accessible.

EIDRM
The message queue was removed.

EINTR
Sleeping on a full message queue condition, the process caught a signal.

EINVAL
Invalid *msqid* value, or nonpositive *mtype* value, or invalid *msgsz* value (less than 0 or greater than the system value **MSGMAX**).

ENOMEM
The system does not have enough memory to make a copy of the message pointed to by *msgp*.

msgrcv() can fail with the following errors:

E2BIG The message text length is greater than *msgsz* and **MSG_NOERROR** isn't specified in *msgflg*.

EACCES
The calling process does not have read permission on the message queue, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EFAULT
The address pointed to by *msgp* isn't accessible.

EIDRM
While the process was sleeping to receive a message, the message queue was removed.

EINTR
While the process was sleeping to receive a message, the process caught a signal; see *signal*(7).

EINVAL
msqid was invalid, or *msgsz* was less than 0.

EINVAL (since Linux 3.14)
msgflg specified **MSG_COPY**, but not **IPC_NOWAIT**.

EINVAL (since Linux 3.14)
msgflg specified both **MSG_COPY** and **MSG_EXCEPT**.

ENOMSG
IPC_NOWAIT was specified in *msgflg* and no message of the requested type existed on the message queue.

ENOMSG
IPC_NOWAIT and **MSG_COPY** were specified in *msgflg* and the queue contains less than *msgtyp* messages.

ENOSYS (since Linux 3.8)
Both **MSG_COPY** and **IPC_NOWAIT** were specified in *msgflg*, and this kernel was configured without **CONFIG_CHECKPOINT_RESTORE**.

STANDARDS

POSIX.1-2008.

The **MSG_EXCEPT** and **MSG_COPY** flags are Linux-specific; their definitions can be obtained by defining the **_GNU_SOURCE** feature test macro.

HISTORY

POSIX.1-2001, SVr4.

The *msgp* argument is declared as *struct msgbuf* * in glibc 2.0 and 2.1. It is declared as *void* * in glibc 2.2 and later, as required by SUSv2 and SUSv3.

NOTES

The following limits on message queue resources affect the **msgsnd**() call:

MSGMAX
: Maximum size of a message text, in bytes (default value: 8192 bytes). On Linux, this limit can be read and modified via */proc/sys/kernel/msgmax*.

MSGMNB
: Maximum number of bytes that can be held in a message queue (default value: 16384 bytes). On Linux, this limit can be read and modified via */proc/sys/kernel/msgmnb*. A privileged process (Linux: a process with the **CAP_SYS_RESOURCE** capability) can increase the size of a message queue beyond **MSGMNB** using the *msgctl*(2) **IPC_SET** operation.

The implementation has no intrinsic system-wide limits on the number of message headers (**MSGTQL**) and the number of bytes in the message pool (**MSGPOOL**).

BUGS

In Linux 3.13 and earlier, if **msgrcv**() was called with the **MSG_COPY** flag, but without **IPC_NOWAIT**, and the message queue contained less than *msgtyp* messages, then the call would block until the next message is written to the queue. At that point, the call would return a copy of the message, *regardless* of whether that message was at the ordinal position *msgtyp*. This bug is fixed in Linux 3.14.

Specifying both **MSG_COPY** and **MSC_EXCEPT** in *msgflg* is a logical error (since these flags impose different interpretations on *msgtyp*). In Linux 3.13 and earlier, this error was not diagnosed by **msgrcv**(). This bug is fixed in Linux 3.14.

EXAMPLES

The program below demonstrates the use of **msgsnd**() and **msgrcv**().

The example program is first run with the **−s** option to send a message and then run again with the **−r** option to receive a message.

The following shell session shows a sample run of the program:

```
$ ./a.out -s
sent: a message at Wed Mar  4 16:25:45 2015

$ ./a.out -r
message received: a message at Wed Mar  4 16:25:45 2015
```

Program source

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>
#include <unistd.h>

struct msgbuf {
    long mtype;
    char mtext[80];
};
```

```
static void
usage(char *prog_name, char *msg)
{
    if (msg != NULL)
        fputs(msg, stderr);

    fprintf(stderr, "Usage: %s [options]\n", prog_name);
    fprintf(stderr, "Options are:\n");
    fprintf(stderr, "-s        send message using msgsnd()\n");
    fprintf(stderr, "-r        read message using msgrcv()\n");
    fprintf(stderr, "-t        message type (default is 1)\n");
    fprintf(stderr, "-k        message queue key (default is 1234)\n");
    exit(EXIT_FAILURE);
}

static void
send_msg(int qid, int msgtype)
{
    time_t         t;
    struct msgbuf  msg;

    msg.mtype = msgtype;

    time(&t);
    snprintf(msg.mtext, sizeof(msg.mtext), "a message at %s",
             ctime(&t));

    if (msgsnd(qid, &msg, sizeof(msg.mtext),
               IPC_NOWAIT) == -1)
    {
        perror("msgsnd error");
        exit(EXIT_FAILURE);
    }
    printf("sent: %s\n", msg.mtext);
}

static void
get_msg(int qid, int msgtype)
{
    struct msgbuf msg;

    if (msgrcv(qid, &msg, sizeof(msg.mtext), msgtype,
               MSG_NOERROR | IPC_NOWAIT) == -1) {
        if (errno != ENOMSG) {
            perror("msgrcv");
            exit(EXIT_FAILURE);
        }
        printf("No message available for msgrcv()\n");
    } else {
        printf("message received: %s\n", msg.mtext);
    }
}

int
main(int argc, char *argv[])
{
    int  qid, opt;
    int  mode = 0;              /* 1 = send, 2 = receive */
```

```
        int  msgtype = 1;
        int  msgkey = 1234;

        while ((opt = getopt(argc, argv, "srt:k:")) != -1) {
            switch (opt) {
            case 's':
                mode = 1;
                break;
            case 'r':
                mode = 2;
                break;
            case 't':
                msgtype = atoi(optarg);
                if (msgtype <= 0)
                    usage(argv[0], "-t option must be greater than 0\n");
                break;
            case 'k':
                msgkey = atoi(optarg);
                break;
            default:
                usage(argv[0], "Unrecognized option\n");
            }
        }

        if (mode == 0)
            usage(argv[0], "must use either -s or -r option\n");

        qid = msgget(msgkey, IPC_CREAT | 0666);

        if (qid == -1) {
            perror("msgget");
            exit(EXIT_FAILURE);
        }

        if (mode == 2)
            get_msg(qid, msgtype);
        else
            send_msg(qid, msgtype);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

msgctl(2), *msgget*(2), *capabilities*(7), *mq_overview*(7), *sysvipc*(7)

NAME

msync − synchronize a file with a memory map

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

int msync(void *addr***[.***length***], size_t** *length***, int** *flags***);**

DESCRIPTION

msync() flushes changes made to the in-core copy of a file that was mapped into memory using *mmap*(2) back to the filesystem. Without use of this call, there is no guarantee that changes are written back before *munmap*(2) is called. To be more precise, the part of the file that corresponds to the memory area starting at *addr* and having length *length* is updated.

The *flags* argument should specify exactly one of **MS_ASYNC** and **MS_SYNC**, and may additionally include the **MS_INVALIDATE** bit. These bits have the following meanings:

MS_ASYNC
Specifies that an update be scheduled, but the call returns immediately.

MS_SYNC
Requests an update and waits for it to complete.

MS_INVALIDATE
Asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBUSY
MS_INVALIDATE was specified in *flags*, and a memory lock exists for the specified address range.

EINVAL
addr is not a multiple of PAGESIZE; or any bit other than **MS_ASYNC** | **MS_INVALIDATE** | **MS_SYNC** is set in *flags*; or both **MS_SYNC** and **MS_ASYNC** are set in *flags*.

ENOMEM
The indicated memory (or part of it) was not mapped.

VERSIONS

According to POSIX, either **MS_SYNC** or **MS_ASYNC** must be specified in *flags*, and indeed failure to include one of these flags will cause **msync**() to fail on some systems. However, Linux permits a call to **msync**() that specifies neither of these flags, with semantics that are (currently) equivalent to specifying **MS_ASYNC**. (Since Linux 2.6.19, **MS_ASYNC** is in fact a no-op, since the kernel properly tracks dirty pages and flushes them to storage as necessary.) Notwithstanding the Linux behavior, portable, future-proof applications should ensure that they specify either **MS_SYNC** or **MS_ASYNC** in *flags*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

This call was introduced in Linux 1.3.21, and then used **EFAULT** instead of **ENOMEM**. In Linux 2.4.19, this was changed to the POSIX value **ENOMEM**.

On POSIX systems on which **msync**() is available, both **_POSIX_MAPPED_FILES** and **_POSIX_SYNCHRONIZED_IO** are defined in <*unistd.h*> to a value greater than 0. (See also *sysconf*(3).)

SEE ALSO

mmap(2)

B.O. Gallmeister, POSIX.4, O’Reilly, pp. 128–129 and 389–391.

NAME

nanosleep − high-resolution sleep

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

int nanosleep(const struct timespec *duration,
              struct timespec *_Nullable rem);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nanosleep():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

nanosleep() suspends the execution of the calling thread until either at least the time specified in **duration* has elapsed, or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process.

If the call is interrupted by a signal handler, **nanosleep**() returns −1, sets *errno* to **EINTR**, and writes the remaining time into the structure pointed to by *rem* unless *rem* is NULL. The value of **rem* can then be used to call **nanosleep**() again and complete the specified pause (but see NOTES).

The *timespec*(3) structure is used to specify intervals of time with nanosecond precision.

The value of the nanoseconds field must be in the range [0, 999999999].

Compared to *sleep*(3) and *usleep*(3), **nanosleep**() has the following advantages: it provides a higher resolution for specifying the sleep interval; POSIX.1 explicitly specifies that it does not interact with signals; and it makes the task of resuming a sleep that has been interrupted by a signal handler easier.

RETURN VALUE

On successfully sleeping for the requested duration, **nanosleep**() returns 0. If the call is interrupted by a signal handler or encounters an error, then it returns −1, with *errno* set to indicate the error.

ERRORS

EFAULT
: Problem with copying information from user space.

EINTR
: The pause has been interrupted by a signal that was delivered to the thread (see *signal*(7)). The remaining sleep time has been written into **rem* so that the thread can easily call **nanosleep**() again and continue with the pause.

EINVAL
: The value in the *tv_nsec* field was not in the range [0, 999999999] or *tv_sec* was negative.

VERSIONS

POSIX.1 specifies that **nanosleep**() should measure time against the **CLOCK_REALTIME** clock. However, Linux measures the time using the **CLOCK_MONOTONIC** clock. This probably does not matter, since the POSIX.1 specification for *clock_settime*(2) says that discontinuous changes in **CLOCK_REALTIME** should not affect **nanosleep**():

> Setting the value of the **CLOCK_REALTIME** clock via *clock_settime*(2) shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the **nanosleep**() function; ... Consequently, these time services shall expire when the requested duration elapses, independently of the new or old value of the clock.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

In order to support applications requiring much more precise pauses (e.g., in order to control some time-critical hardware), **nanosleep**() would handle pauses of up to 2 milliseconds by busy waiting with microsecond precision when called from a thread scheduled under a real-time policy like

SCHED_FIFO or **SCHED_RR**. This special extension was removed in Linux 2.5.39, and is thus not available in Linux 2.6.0 and later kernels.

NOTES

If the *duration* is not an exact multiple of the granularity underlying clock (see *time*(7)), then the interval will be rounded up to the next multiple. Furthermore, after the sleep completes, there may still be a delay before the CPU becomes free to once again execute the calling thread.

The fact that **nanosleep**() sleeps for a relative interval can be problematic if the call is repeatedly restarted after being interrupted by signals, since the time between the interruptions and restarts of the call will lead to drift in the time when the sleep finally completes. This problem can be avoided by using *clock_nanosleep*(2) with an absolute time value.

BUGS

If a program that catches signals and uses **nanosleep**() receives signals at a very high rate, then scheduling delays and rounding errors in the kernel's calculation of the sleep interval and the returned *remain* value mean that the *remain* value may steadily *increase* on successive restarts of the **nanosleep**() call. To avoid such problems, use *clock_nanosleep*(2) with the **TIMER_ABSTIME** flag to sleep to an absolute deadline.

In Linux 2.4, if **nanosleep**() is stopped by a signal (e.g., **SIGTSTP**), then the call fails with the error **EINTR** after the thread is resumed by a **SIGCONT** signal. If the system call is subsequently restarted, then the time that the thread spent in the stopped state is *not* counted against the sleep interval. This problem is fixed in Linux 2.6.0 and later kernels.

SEE ALSO

clock_nanosleep(2), *restart_syscall*(2), *sched_setscheduler*(2), *timer_create*(2), *sleep*(3), *timespec*(3), *usleep*(3), *time*(7)

NAME

nfsservctl − syscall interface to kernel nfs daemon

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <linux/nfsd/syscall.h>

long nfsservctl(int *cmd***, struct nfsctl_arg ***_argp_**,**
union nfsctl_res ***_resp_);**

DESCRIPTION

Note: Since Linux 3.1, this system call no longer exists. It has been replaced by a set of files in the *nfsd* filesystem; see *nfsd*(7)

```
/*
 * These are the commands understood by nfsctl().
 */
#define NFSCTL_SVC        0  /* This is a server process. */
#define NFSCTL_ADDCLIENT  1  /* Add an NFS client. */
#define NFSCTL_DELCLIENT  2  /* Remove an NFS client. */
#define NFSCTL_EXPORT     3  /* Export a filesystem. */
#define NFSCTL_UNEXPORT   4  /* Unexport a filesystem. */
#define NFSCTL_UGIDUPDATE 5  /* Update a client's UID/GID map
                                (only in Linux 2.4.x and earlier). */
#define NFSCTL_GETFH      6  /* Get a file handle (used by mountd(8))
                                (only in Linux 2.4.x and earlier). */

struct nfsctl_arg {
    int                     ca_version;     /* safeguard */
    union {
        struct nfsctl_svc     u_svc;
        struct nfsctl_client  u_client;
        struct nfsctl_export  u_export;
        struct nfsctl_uidmap  u_umap;
        struct nfsctl_fhparm  u_getfh;
        unsigned int          u_debug;
    } u;
}

union nfsctl_res {
        struct knfs_fh          cr_getfh;
        unsigned int            cr_debug;
};
```

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

STANDARDS

Linux.

HISTORY

Removed in Linux 3.1. Removed in glibc 2.28.

SEE ALSO

nfsd(7)

NAME

nice − change process priority

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int nice(int inc);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nice():
_XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

nice() adds *inc* to the nice value for the calling thread. (A higher nice value means a lower priority.)

The range of the nice value is +19 (low priority) to −20 (high priority). Attempts to set a nice value outside the range are clamped to the range.

Traditionally, only a privileged process could lower the nice value (i.e., set a higher priority). However, since Linux 2.6.12, an unprivileged process can decrease the nice value of a target process that has a suitable **RLIMIT_NICE** soft limit; see *getrlimit*(2) for details.

RETURN VALUE

On success, the new nice value is returned (but see NOTES below). On error, −1 is returned, and *errno* is set to indicate the error.

A successful call can legitimately return −1. To detect an error, set *errno* to 0 before the call, and check whether it is nonzero after **nice**() returns −1.

ERRORS

EPERM
The calling process attempted to increase its priority by supplying a negative *inc* but has insufficient privileges. Under Linux, the **CAP_SYS_NICE** capability is required. (But see the discussion of the **RLIMIT_NICE** resource limit in *setrlimit*(2).)

VERSIONS

C library/kernel differences

POSIX.1 specifies that **nice**() should return the new nice value. However, the raw Linux system call returns 0 on success. Likewise, the **nice**() wrapper function provided in glibc 2.2.3 and earlier returns 0 on success.

Since glibc 2.2.4, the **nice**() wrapper function provided by glibc provides conformance to POSIX.1 by calling *getpriority*(2) to obtain the new nice value, which is then returned to the caller.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

For further details on the nice value, see *sched*(7).

Note: the addition of the "autogroup" feature in Linux 2.6.38 means that the nice value no longer has its traditional effect in many circumstances. For details, see *sched*(7).

SEE ALSO

nice(1), *renice*(1), *fork*(2), *getpriority*(2), *getrlimit*(2), *setpriority*(2), *capabilities*(7), *sched*(7)

NAME

open, openat, creat − open and possibly create a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>

int open(const char *pathname, int flags, ...
        /* mode_t mode */ );

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags, ...
        /* mode_t mode */ );

/* Documented separately, in openat2(2): */
int openat2(int dirfd, const char *pathname,
        const struct open_how *how, size_t size);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

openat():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _ATFILE_SOURCE

DESCRIPTION

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls (*read*(2), *write*(2), *lseek*(2), *fcntl*(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an *execve*(2) (i.e., the **FD_CLOEXEC** file descriptor flag described in *fcntl*(2) is initially disabled); the **O_CLOEXEC** flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see *lseek*(2)).

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise ORed in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see *fcntl*(2) for details.

The full list of file creation flags and file status flags is as follows:

O_APPEND

The file is opened in append mode. Before each *write*(2), the file offset is positioned at the end of the file, as if with *lseek*(2). The modification of the file offset and the write operation are performed as a single atomic step.

O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_ASYNC
Enable signal-driven I/O: generate a signal (**SIGIO** by default, but this can be changed via *fcntl*(2)) when input or output becomes possible on this file descriptor. This feature is available only for terminals, pseudoterminals, sockets, and (since Linux 2.6) pipes and FIFOs. See *fcntl*(2) for further details. See also BUGS, below.

O_CLOEXEC (since Linux 2.6.23)
Enable the close-on-exec flag for the new file descriptor. Specifying this flag permits a program to avoid additional *fcntl*(2) **F_SETFD** operations to set the **FD_CLOEXEC** flag.

Note that the use of this flag is essential in some multithreaded programs, because using a separate *fcntl*(2) **F_SETFD** operation to set the **FD_CLOEXEC** flag does not suffice to avoid race conditions where one thread opens a file descriptor and attempts to set its close-on-exec flag using *fcntl*(2) at the same time as another thread does a *fork*(2) plus *execve*(2). Depending on the order of execution, the race may lead to the file descriptor returned by **open**() being unintentionally leaked to the program executed by the child process created by *fork*(2). (This kind of race is in principle possible for any system call that creates a file descriptor whose close-on-exec flag should be set, and various other Linux system calls provide an equivalent of the **O_CLOEXEC** flag to deal with this problem.)

O_CREAT
If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The group ownership (group ID) of the new file is set either to the effective group ID of the process (System V semantics) or to the group ID of the parent directory (BSD semantics). On Linux, the behavior depends on whether the set-group-ID mode bit is set on the parent directory: if that bit is set, then BSD semantics apply; otherwise, System V semantics apply. For some filesystems, the behavior also depends on the *bsdgroups* and *sysvgroups* mount options described in *mount*(8)

The *mode* argument specifies the file mode bits to be applied when a new file is created. If neither **O_CREAT** nor **O_TMPFILE** is specified in *flags*, then *mode* is ignored (and can thus be specified as 0, or simply omitted). The *mode* argument **must** be supplied if **O_CREAT** or **O_TMPFILE** is specified in *flags*; if it is not supplied, some arbitrary bytes from the stack will be applied as the file mode.

The effective mode is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created file is *(mode & ~umask)*.

Note that *mode* applies only to future accesses of the newly created file; the **open**() call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU
00700 user (file owner) has read, write, and execute permission

S_IRUSR
00400 user has read permission

S_IWUSR
00200 user has write permission

S_IXUSR
00100 user has execute permission

S_IRWXG
00070 group has read, write, and execute permission

S_IRGRP
00040 group has read permission

S_IWGRP
00020 group has write permission

S_IXGRP
00010 group has execute permission

S_IRWXO
00007 others have read, write, and execute permission

S_IROTH
00004 others have read permission

S_IWOTH
00002 others have write permission

S_IXOTH
00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode* is unspecified. On Linux, the following bits are also honored in *mode*:

S_ISUID 0004000 set-user-ID bit

S_ISGID 0002000 set-group-ID bit (see *inode*(7)).

S_ISVTX
0001000 sticky bit (see *inode*(7)).

O_DIRECT (since Linux 2.4.10)
Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user-space buffers. The **O_DIRECT** flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the **O_SYNC** flag that data and necessary metadata are transferred. To guarantee synchronous I/O, **O_SYNC** must be used in addition to **O_DIRECT**. See NOTES below for further discussion.

A semantically similar (but deprecated) interface for block devices is described in *raw*(8)

O_DIRECTORY
If *pathname* is not a directory, cause the open to fail. This flag was added in Linux 2.1.126, to avoid denial-of-service problems if *opendir*(3) is called on a FIFO or tape device.

O_DSYNC
Write operations on the file will complete according to the requirements of synchronized I/O *data* integrity completion.

By the time *write*(2) (and similar) return, the output data has been transferred to the underlying hardware, along with any file metadata that would be required to retrieve that data (i.e., as though each *write*(2) was followed by a call to *fdatasync*(2)). *See NOTES below*.

O_EXCL
Ensure that this call creates the file: if this flag is specified in conjunction with **O_CREAT**, and *pathname* already exists, then **open**() fails with the error **EEXIST**.

When these two flags are specified, symbolic links are not followed: if *pathname* is a symbolic link, then **open**() fails regardless of where the symbolic link points.

In general, the behavior of **O_EXCL** is undefined if it is used without **O_CREAT**. There is one exception: on Linux 2.6 and later, **O_EXCL** can be used without **O_CREAT** if *pathname* refers to a block device. If the block device is in use by the system (e.g., mounted), **open**() fails with the error **EBUSY**.

On NFS, **O_EXCL** is supported only when using NFSv3 or later on kernel 2.6 or later. In NFS environments where **O_EXCL** support is not provided, programs that rely on it for performing locking tasks will contain a race condition. Portable programs that want to perform atomic file locking using a lockfile, and need to avoid reliance on NFS support for **O_EXCL**, can create a unique file on the same filesystem (e.g., incorporating hostname and PID), and use *link*(2) to make a link to the lockfile. If *link*(2) returns 0, the lock is successful. Otherwise, use *stat*(2) on the unique file to check if its link count has increased to 2, in which case the lock is also successful.

O_LARGEFILE
(LFS) Allow files whose sizes cannot be represented in an *off_t* (but can be represented in an *off64_t*) to be opened. The **_LARGEFILE64_SOURCE** macro must be defined (before including *any* header files) in order to obtain this definition. Setting the **_FILE_OFFSET_BITS** feature test macro to 64 (rather than using **O_LARGEFILE**) is the preferred method of accessing large files on 32-bit systems (see *feature_test_macros*(7)).

O_NOATIME (since Linux 2.6.8)
Do not update the file last access time (*st_atime* in the inode) when the file is *read*(2).

This flag can be employed only if one of the following conditions is true:

- The effective UID of the process matches the owner UID of the file.
- The calling process has the **CAP_FOWNER** capability in its user namespace and the owner UID of the file has a mapping in the namespace.

This flag is intended for use by indexing or backup programs, where its use can significantly reduce the amount of disk activity. This flag may not be effective on all filesystems. One example is NFS, where the server maintains the access time.

O_NOCTTY
If *pathname* refers to a terminal device—see *tty*(4)—it will not become the process's controlling terminal even if the process does not have one.

O_NOFOLLOW
If the trailing component (i.e., basename) of *pathname* is a symbolic link, then the open fails, with the error **ELOOP**. Symbolic links in earlier components of the pathname will still be followed. (Note that the **ELOOP** error that can occur in this case is indistinguishable from the case where an open fails because there are too many symbolic links found while resolving components in the prefix part of the pathname.)

This flag is a FreeBSD extension, which was added in Linux 2.1.126, and has subsequently been standardized in POSIX.1-2008.

See also **O_PATH** below.

O_NONBLOCK or **O_NDELAY**
When possible, the file is opened in nonblocking mode. Neither the **open**() nor any subsequent I/O operations on the file descriptor which is returned will cause the calling process to wait.

Note that the setting of this flag has no effect on the operation of *poll*(2), *select*(2), *epoll*(7), and similar, since those interfaces merely inform the caller about whether a file descriptor is "ready", meaning that an I/O operation performed on the file descriptor with the **O_NONBLOCK** flag *clear* would not block.

Note that this flag has no effect for regular files and block devices; that is, I/O operations will (briefly) block when device activity is required, regardless of whether **O_NONBLOCK** is set. Since **O_NONBLOCK** semantics might eventually be implemented, applications should not depend upon blocking behavior when specifying this flag for regular files and block devices.

For the handling of FIFOs (named pipes), see also *fifo*(7). For a discussion of the effect of **O_NONBLOCK** in conjunction with mandatory file locks and with file leases, see *fcntl*(2).

O_PATH (since Linux 2.6.39)
Obtain a file descriptor that can be used for two purposes: to indicate a location in the filesystem tree and to perform operations that act purely at the file descriptor level. The file itself is not opened, and other file operations (e.g., *read*(2), *write*(2), *fchmod*(2), *fchown*(2), *fgetxattr*(2), *ioctl*(2), *mmap*(2)) fail with the error **EBADF**.

The following operations *can* be performed on the resulting file descriptor:

- *close*(2).
- *fchdir*(2), if the file descriptor refers to a directory (since Linux 3.5).

- *fstat*(2) (since Linux 3.6).
- *fstatfs*(2) (since Linux 3.12).
- Duplicating the file descriptor (**dup**(2), *fcntl*(2) **F_DUPFD**, etc.).
- Getting and setting file descriptor flags (**fcntl**(2) **F_GETFD** and **F_SETFD**).
- Retrieving open file status flags using the *fcntl*(2) **F_GETFL** operation: the returned flags will include the bit **O_PATH**.
- Passing the file descriptor as the *dirfd* argument of **openat**() and the other "*at()" system calls. This includes *linkat*(2) with **AT_EMPTY_PATH** (or via procfs using **AT_SYMLINK_FOLLOW**) even if the file is not a directory.
- Passing the file descriptor to another process via a UNIX domain socket (see **SCM_RIGHTS** in *unix*(7)).

When **O_PATH** is specified in *flags*, flag bits other than **O_CLOEXEC**, **O_DIRECTORY**, and **O_NOFOLLOW** are ignored.

Opening a file or directory with the **O_PATH** flag requires no permissions on the object itself (but does require execute permission on the directories in the path prefix). Depending on the subsequent operation, a check for suitable file permissions may be performed (e.g., *fchdir*(2) requires execute permission on the directory referred to by its file descriptor argument). By contrast, obtaining a reference to a filesystem object by opening it with the **O_RDONLY** flag requires that the caller have read permission on the object, even when the subsequent operation (e.g., *fchdir*(2), *fstat*(2)) does not require read permission on the object.

If *pathname* is a symbolic link and the **O_NOFOLLOW** flag is also specified, then the call returns a file descriptor referring to the symbolic link. This file descriptor can be used as the *dirfd* argument in calls to *fchownat*(2), *fstatat*(2), *linkat*(2), and *readlinkat*(2) with an empty pathname to have the calls operate on the symbolic link.

If *pathname* refers to an automount point that has not yet been triggered, so no other filesystem is mounted on it, then the call returns a file descriptor referring to the automount directory without triggering a mount. *fstatfs*(2) can then be used to determine if it is, in fact, an untriggered automount point (**.f_type == AUTOFS_SUPER_MAGIC**).

One use of **O_PATH** for regular files is to provide the equivalent of POSIX.1's **O_EXEC** functionality. This permits us to open a file for which we have execute permission but not read permission, and then execute that file, with steps something like the following:

```
char buf[PATH_MAX];
fd = open("some_prog", O_PATH);
snprintf(buf, PATH_MAX, "/proc/self/fd/%d", fd);
execl(buf, "some_prog", (char *) NULL);
```

An **O_PATH** file descriptor can also be passed as the argument of *fexecve*(3).

O_SYNC

Write operations on the file will complete according to the requirements of synchronized I/O *file* integrity completion (by contrast with the synchronized I/O *data* integrity completion provided by **O_DSYNC**.)

By the time *write*(2) (or similar) returns, the output data and associated file metadata have been transferred to the underlying hardware (i.e., as though each *write*(2) was followed by a call to *fsync*(2)). *See NOTES below*.

O_TMPFILE (since Linux 3.11)

Create an unnamed temporary regular file. The *pathname* argument specifies a directory; an unnamed inode will be created in that directory's filesystem. Anything written to the resulting file will be lost when the last file descriptor is closed, unless the file is given a name.

O_TMPFILE must be specified with one of **O_RDWR** or **O_WRONLY** and, optionally, **O_EXCL**. If **O_EXCL** is not specified, then *linkat*(2) can be used to link the temporary file into the filesystem, making it permanent, using code like the following:

```
char path[PATH_MAX];
fd = open("/path/to/dir", O_TMPFILE | O_RDWR,
                          S_IRUSR | S_IWUSR);

/* File I/O on 'fd'... */

linkat(fd, "", AT_FDCWD, "/path/for/file", AT_EMPTY_PATH);

/* If the caller doesn't have the CAP_DAC_READ_SEARCH
   capability (needed to use AT_EMPTY_PATH with linkat(2)),
   and there is a proc(5) filesystem mounted, then the
   linkat(2) call above can be replaced with:

snprintf(path, PATH_MAX,  "/proc/self/fd/%d", fd);
linkat(AT_FDCWD, path, AT_FDCWD, "/path/for/file",
                          AT_SYMLINK_FOLLOW);
*/
```

In this case, the **open**() *mode* argument determines the file permission mode, as with **O_CREAT**.

Specifying **O_EXCL** in conjunction with **O_TMPFILE** prevents a temporary file from being linked into the filesystem in the above manner. (Note that the meaning of **O_EXCL** in this case is different from the meaning of **O_EXCL** otherwise.)

There are two main use cases for **O_TMPFILE**:

- Improved *tmpfile*(3) functionality: race-free creation of temporary files that (1) are automatically deleted when closed; (2) can never be reached via any pathname; (3) are not subject to symlink attacks; and (4) do not require the caller to devise unique names.
- Creating a file that is initially invisible, which is then populated with data and adjusted to have appropriate filesystem attributes (**fchown**(2), *fchmod*(2), *fsetxattr*(2), etc.) before being atomically linked into the filesystem in a fully formed state (using *linkat*(2) as described above).

O_TMPFILE requires support by the underlying filesystem; only a subset of Linux filesystems provide that support. In the initial implementation, support was provided in the ext2, ext3, ext4, UDF, Minix, and tmpfs filesystems. Support for other filesystems has subsequently been added as follows: XFS (Linux 3.15); Btrfs (Linux 3.16); F2FS (Linux 3.16); and ubifs (Linux 4.9)

O_TRUNC
If the file already exists and is a regular file and the access mode allows writing (i.e., is **O_RDWR** or **O_WRONLY**) it will be truncated to length 0. If the file is a FIFO or terminal device file, the **O_TRUNC** flag is ignored. Otherwise, the effect of **O_TRUNC** is unspecified.

creat()
A call to **creat**() is equivalent to calling **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

openat()
The **openat**() system call operates in exactly the same way as **open**(), except for the differences described here.

The *dirfd* argument is used in conjunction with the *pathname* argument as follows:

- If the pathname given in *pathname* is absolute, then *dirfd* is ignored.
- If the pathname given in *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *open*())
- If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **open**() for a relative pathname). In this case, *dirfd* must be a directory that

was opened for reading (**O_RDONLY**) or using the **O_PATH** flag.

If the pathname given in *pathname* is relative, and *dirfd* is not a valid file descriptor, an error (**EBADF**) results. (Specifying an invalid file descriptor number in *dirfd* can be used as a means to ensure that *pathname* is absolute.)

openat2(2)

The *openat2*(2) system call is an extension of **openat**(), and provides a superset of the features of **openat**(). It is documented separately, in *openat2*(2).

RETURN VALUE

On success, **open**(), **openat**(), and **creat**() return the new file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

open(), **openat**(), and **creat**() can fail with the following errors:

EACCES

The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *pathname*, or the file did not exist yet and write access to the parent directory is not allowed. (See also *path_resolution*(7).)

EACCES

Where **O_CREAT** is specified, the *protected_fifos* or *protected_regular* sysctl is enabled, the file already exists and is a FIFO or regular file, the owner of the file is neither the current user nor the owner of the containing directory, and the containing directory is both world- or group-writable and sticky. For details, see the descriptions of */proc/sys/fs/protected_fifos* and */proc/sys/fs/protected_regular* in *proc*(5).

EBADF

(**openat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EBUSY

O_EXCL was specified in *flags* and *pathname* refers to a block device that is in use by the system (e.g., it is mounted).

EDQUOT

Where **O_CREAT** is specified, the file does not exist, and the user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST

pathname already exists and **O_CREAT** and **O_EXCL** were used.

EFAULT

pathname points outside your accessible address space.

EFBIG

See **EOVERFLOW**.

EINTR

While blocked waiting to complete an open of a slow device (e.g., a FIFO; see *fifo*(7)), the call was interrupted by a signal handler; see *signal*(7).

EINVAL

The filesystem does not support the **O_DIRECT** flag. See **NOTES** for more information.

EINVAL

Invalid value in *flags*.

EINVAL

O_TMPFILE was specified in *flags*, but neither **O_WRONLY** nor **O_RDWR** was specified.

EINVAL

O_CREAT was specified in *flags* and the final component ("basename") of the new file's *pathname* is invalid (e.g., it contains characters not permitted by the underlying filesystem).

EINVAL

The final component ("basename") of *pathname* is invalid (e.g., it contains characters not permitted by the underlying filesystem).

EISDIR
pathname refers to a directory and the access requested involved writing (that is, **O_WRONLY** or **O_RDWR** is set).

EISDIR
pathname refers to an existing directory, **O_TMPFILE** and one of **O_WRONLY** or **O_RDWR** were specified in *flags*, but this kernel version does not provide the **O_TMPFILE** functionality.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ELOOP
pathname was a symbolic link, and *flags* specified **O_NOFOLLOW** but not **O_PATH**.

EMFILE
The per-process limit on the number of open file descriptors has been reached (see the description of **RLIMIT_NOFILE** in *getrlimit*(2)).

ENAMETOOLONG
pathname was too long.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
pathname refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation **ENXIO** must be returned.)

ENOENT
O_CREAT is not set and the named file does not exist.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOENT
pathname refers to a nonexistent directory, **O_TMPFILE** and one of **O_WRONLY** or **O_RDWR** were specified in *flags*, but this kernel version does not provide the **O_TMPFILE** functionality.

ENOMEM
The named file is a FIFO, but memory for the FIFO buffer can't be allocated because the per-user hard limit on memory allocation for pipes has been reached and the caller is not privileged; see *pipe*(7).

ENOMEM
Insufficient kernel memory was available.

ENOSPC
pathname was to be created but the device containing *pathname* has no room for the new file.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory, or **O_DIRECTORY** was specified and *pathname* was not a directory.

ENOTDIR
(**openat**()) *pathname* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

ENXIO
O_NONBLOCK | **O_WRONLY** is set, the named file is a FIFO, and no process has the FIFO open for reading.

ENXIO
The file is a device special file and no corresponding device exists.

ENXIO
The file is a UNIX domain socket.

EOPNOTSUPP
The filesystem containing *pathname* does not support **O_TMPFILE**.

EOVERFLOW
pathname refers to a regular file that is too large to be opened. The usual scenario here is that an application compiled on a 32-bit platform without *−D_FILE_OFFSET_BITS=64* tried to open a file whose size exceeds *(1<<31)−1* bytes; see also **O_LARGEFILE** above. This is the error specified by POSIX.1; before Linux 2.6.24, Linux gave the error **EFBIG** for this case.

EPERM
The **O_NOATIME** flag was specified, but the effective user ID of the caller did not match the owner of the file and the caller was not privileged.

EPERM
The operation was prevented by a file seal; see *fcntl*(2).

EROFS
pathname refers to a file on a read-only filesystem and write access was requested.

ETXTBSY
pathname refers to an executable image which is currently being executed and write access was requested.

ETXTBSY
pathname refers to a file that is currently in use as a swap file, and the **O_TRUNC** flag was specified.

ETXTBSY
pathname refers to a file that is currently being read by the kernel (e.g., for module/firmware loading), and write access was requested.

EWOULDBLOCK
The **O_NONBLOCK** flag was specified, and an incompatible lease was held on the file (see *fcntl*(2)).

VERSIONS

The (undefined) effect of **O_RDONLY | O_TRUNC** varies among implementations. On many systems the file is actually truncated.

Synchronized I/O

The POSIX.1-2008 "synchronized I/O" option specifies different variants of synchronized I/O, and specifies the **open**() flags **O_SYNC**, **O_DSYNC**, and **O_RSYNC** for controlling the behavior. Regardless of whether an implementation supports this option, it must at least support the use of **O_SYNC** for regular files.

Linux implements **O_SYNC** and **O_DSYNC**, but not **O_RSYNC**. Somewhat incorrectly, glibc defines **O_RSYNC** to have the same value as **O_SYNC**. (**O_RSYNC** is defined in the Linux header file *<asm/fcntl.h>* on HP PA-RISC, but it is not used.)

O_SYNC provides synchronized I/O *file* integrity completion, meaning write operations will flush data and all associated metadata to the underlying hardware. **O_DSYNC** provides synchronized I/O *data* integrity completion, meaning write operations will flush data to the underlying hardware, but will only flush metadata updates that are required to allow a subsequent read operation to complete successfully. Data integrity completion can reduce the number of disk operations that are required for applications that don’t need the guarantees of file integrity completion.

To understand the difference between the two types of completion, consider two pieces of file metadata: the file last modification timestamp (*st_mtime*) and the file length. All write operations will update the last file modification timestamp, but only writes that add data to the end of the file will change the file length. The last modification timestamp is not needed to ensure that a read completes successfully, but the file length is. Thus, **O_DSYNC** would only guarantee to flush updates to the file length metadata (whereas **O_SYNC** would also always flush the last modification timestamp metadata).

Before Linux 2.6.33, Linux implemented only the **O_SYNC** flag for **open**(). However, when that flag was specified, most filesystems actually provided the equivalent of synchronized I/O *data* integrity completion (i.e., **O_SYNC** was actually implemented as the equivalent of **O_DSYNC**).

Since Linux 2.6.33, proper **O_SYNC** support is provided. However, to ensure backward binary compatibility, **O_DSYNC** was defined with the same value as the historical **O_SYNC**, and **O_SYNC** was defined as a new (two-bit) flag value that includes the **O_DSYNC** flag value. This ensures that applications compiled against new headers get at least **O_DSYNC** semantics before Linux 2.6.33.

C library/kernel differences

Since glibc 2.26, the glibc wrapper function for **open**() employs the **openat**() system call, rather than the kernel's **open**() system call. For certain architectures, this is also true before glibc 2.26.

STANDARDS

open()
creat()
openat()
POSIX.1-2008.

openat2(2) Linux.

The **O_DIRECT**, **O_NOATIME**, **O_PATH**, and **O_TMPFILE** flags are Linux-specific. One must define **_GNU_SOURCE** to obtain their definitions.

The **O_CLOEXEC**, **O_DIRECTORY**, and **O_NOFOLLOW** flags are not specified in POSIX.1-2001, but are specified in POSIX.1-2008. Since glibc 2.12, one can obtain their definitions by defining either **_POSIX_C_SOURCE** with a value greater than or equal to 200809L or **_XOPEN_SOURCE** with a value greater than or equal to 700. In glibc 2.11 and earlier, one obtains the definitions by defining **_GNU_SOURCE**.

HISTORY

open()
creat() SVr4, 4.3BSD, POSIX.1-2001.

openat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

NOTES

Under Linux, the **O_NONBLOCK** flag is sometimes used in cases where one wants to open but does not necessarily have the intention to read or write. For example, this may be used to open a device in order to get a file descriptor for use with *ioctl*(2).

Note that **open**() can open device special files, but **creat**() cannot create them; use *mknod*(2) instead.

If the file is newly created, its *st_atime*, *st_ctime*, *st_mtime* fields (respectively, time of last access, time of last status change, and time of last modification; see *stat*(2)) are set to the current time, and so are the *st_ctime* and *st_mtime* fields of the parent directory. Otherwise, if the file is modified because of the **O_TRUNC** flag, its *st_ctime* and *st_mtime* fields are set to the current time.

The files in the */proc/*pid*/fd* directory show the open file descriptors of the process with the PID *pid*. The files in the */proc/*pid*/fdinfo* directory show even more information about these file descriptors. See *proc*(5) for further details of both of these directories.

The Linux header file **<asm/fcntl.h>** doesn't define **O_ASYNC**; the (BSD-derived) **FASYNC** synonym is defined instead.

Open file descriptions

The term open file description is the one used by POSIX to refer to the entries in the system-wide table of open files. In other contexts, this object is variously also called an "open file object", a "file handle", an "open file table entry", or—in kernel-developer parlance—a *struct file*.

When a file descriptor is duplicated (using *dup*(2) or similar), the duplicate refers to the same open file description as the original file descriptor, and the two file descriptors consequently share the file offset and file status flags. Such sharing can also occur between processes: a child process created via *fork*(2) inherits duplicates of its parent's file descriptors, and those duplicates refer to the same open file descriptions.

Each **open**() of a file creates a new open file description; thus, there may be multiple open file descriptions corresponding to a file inode.

On Linux, one can use the *kcmp*(2) **KCMP_FILE** operation to test whether two file descriptors (in the same process or in two different processes) refer to the same open file description.

NFS

There are many infelicities in the protocol underlying NFS, affecting amongst others **O_SYNC** and **O_NDELAY**.

On NFS filesystems with UID mapping enabled, **open**() may return a file descriptor but, for example, *read*(2) requests are denied with **EACCES**. This is because the client performs **open**() by checking the permissions, but UID mapping is performed by the server upon read and write requests.

FIFOs

Opening the read or write end of a FIFO blocks until the other end is also opened (by another process or thread). See *fifo*(7) for further details.

File access mode

Unlike the other values that can be specified in *flags*, the *access mode* values **O_RDONLY**, **O_WRONLY**, and **O_RDWR** do not specify individual bits. Rather, they define the low order two bits of *flags*, and are defined respectively as 0, 1, and 2. In other words, the combination **O_RDONLY | O_WRONLY** is a logical error, and certainly does not have the same meaning as **O_RDWR**.

Linux reserves the special, nonstandard access mode 3 (binary 11) in *flags* to mean: check for read and write permission on the file and return a file descriptor that can't be used for reading or writing. This nonstandard access mode is used by some Linux drivers to return a file descriptor that is to be used only for device-specific *ioctl*(2) operations.

Rationale for openat() and other directory file descriptor APIs

openat() and the other system calls and library functions that take a directory file descriptor argument (i.e., *execveat*(2), *faccessat*(2), *fanotify_mark*(2), *fchmodat*(2), *fchownat*(2), *fspick*(2), *fstatat*(2), *futimesat*(2), *linkat*(2), *mkdirat*(2), *mknodat*(2), *mount_setattr*(2), *move_mount*(2), *name_to_handle_at*(2), *open_tree*(2), *openat2*(2), *readlinkat*(2), *renameat*(2), *renameat2*(2), *statx*(2), *symlinkat*(2), *unlinkat*(2), *utimensat*(2), *mkfifoat*(3), and *scandirat*(3)) address two problems with the older interfaces that preceded them. Here, the explanation is in terms of the **openat**() call, but the rationale is analogous for the other interfaces.

First, **openat**() allows an application to avoid race conditions that could occur when using **open**() to open files in directories other than the current working directory. These race conditions result from the fact that some component of the directory prefix given to **open**() could be changed in parallel with the call to **open**(). Suppose, for example, that we wish to create the file *dir1/dir2/xxx.dep* if the file *dir1/dir2/xxx* exists. The problem is that between the existence check and the file-creation step, *dir1* or *dir2* (which might be symbolic links) could be modified to point to a different location. Such races can be avoided by opening a file descriptor for the target directory, and then specifying that file descriptor as the *dirfd* argument of (say) *fstatat*(2) and **openat**(). The use of the *dirfd* file descriptor also has other benefits:

- the file descriptor is a stable reference to the directory, even if the directory is renamed; and
- the open file descriptor prevents the underlying filesystem from being dismounted, just as when a process has a current working directory on a filesystem.

Second, **openat**() allows the implementation of a per-thread "current working directory", via file descriptor(s) maintained by the application. (This functionality can also be obtained by tricks based on the use of */proc/self/fd/*dirfd, but less efficiently.)

The *dirfd* argument for these APIs can be obtained by using **open**() or **openat**() to open a directory (with either the **O_RDONLY** or the **O_PATH** flag). Alternatively, such a file descriptor can be obtained by applying *dirfd*(3) to a directory stream created using *opendir*(3).

When these APIs are given a *dirfd* argument of **AT_FDCWD** or the specified pathname is absolute, then they handle their pathname argument in the same way as the corresponding conventional APIs. However, in this case, several of the APIs have a *flags* argument that provides access to functionality that is not available with the corresponding conventional APIs.

O_DIRECT

The **O_DIRECT** flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. In Linux alignment restrictions vary by filesystem and kernel version and might be absent entirely. The handling of misaligned **O_DIRECT** I/Os also varies; they can either fail with **EINVAL** or fall back to buffered I/O.

Since Linux 6.1, **O_DIRECT** support and alignment restrictions for a file can be queried using *statx*(2), using the **STATX_DIOALIGN** flag. Support for **STATX_DIOALIGN** varies by filesystem; see *statx*(2).

Some filesystems provide their own interfaces for querying **O_DIRECT** alignment restrictions, for example the **XFS_IOC_DIOINFO** operation in *xfsctl*(3)**STATX_DIOALIGN** should be used instead when it is available.

If none of the above is available, then direct I/O support and alignment restrictions can only be assumed from known characteristics of the filesystem, the individual file, the underlying storage device(s), and the kernel version. In Linux 2.4, most filesystems based on block devices require that the file offset and the length and memory address of all I/O segments be multiples of the filesystem block size (typically 4096 bytes). In Linux 2.6.0, this was relaxed to the logical block size of the block device (typically 512 bytes). A block device's logical block size can be determined using the *ioctl*(2) **BLKSSZGET** operation or from the shell using the command:

```
blockdev --getss
```

O_DIRECT I/Os should never be run concurrently with the *fork*(2) system call, if the memory buffer is a private mapping (i.e., any mapping created with the *mmap*(2) **MAP_PRIVATE** flag; this includes memory allocated on the heap and statically allocated buffers). Any such I/Os, whether submitted via an asynchronous I/O interface or from another thread in the process, should be completed before *fork*(2) is called. Failure to do so can result in data corruption and undefined behavior in parent and child processes. This restriction does not apply when the memory buffer for the **O_DIRECT** I/Os was created using *shmat*(2) or *mmap*(2) with the **MAP_SHARED** flag. Nor does this restriction apply when the memory buffer has been advised as **MADV_DONTFORK** with *madvise*(2), ensuring that it will not be available to the child after *fork*(2).

The **O_DIRECT** flag was introduced in SGI IRIX, where it has alignment restrictions similar to those of Linux 2.4. IRIX has also a *fcntl*(2) call to query appropriate alignments, and sizes. FreeBSD 4.x introduced a flag of the same name, but without alignment restrictions.

O_DIRECT support was added in Linux 2.4.10. Older Linux kernels simply ignore this flag. Some filesystems may not implement the flag, in which case **open**() fails with the error **EINVAL** if it is used.

Applications should avoid mixing **O_DIRECT** and normal I/O to the same file, and especially to overlapping byte regions in the same file. Even when the filesystem correctly handles the coherency issues in this situation, overall I/O throughput is likely to be slower than using either mode alone. Likewise, applications should avoid mixing *mmap*(2) of files with direct I/O to the same files.

The behavior of **O_DIRECT** with NFS will differ from local filesystems. Older kernels, or kernels configured in certain ways, may not support this combination. The NFS protocol does not support passing the flag to the server, so **O_DIRECT** I/O will bypass the page cache only on the client; the server may still cache the I/O. The client asks the server to make the I/O synchronous to preserve the synchronous semantics of **O_DIRECT**. Some servers will perform poorly under these circumstances, especially if the I/O size is small. Some servers may also be configured to lie to clients about the I/O having reached stable storage; this will avoid the performance penalty at some risk to data integrity in the event of server power failure. The Linux NFS client places no alignment restrictions on **O_DIRECT** I/O.

In summary, **O_DIRECT** is a potentially powerful tool that should be used with caution. It is recommended that applications treat use of **O_DIRECT** as a performance option which is disabled by default.

BUGS

Currently, it is not possible to enable signal-driven I/O by specifying **O_ASYNC** when calling **open**(); use *fcntl*(2) to enable this flag.

One must check for two different error codes, **EISDIR** and **ENOENT**, when trying to determine whether the kernel supports **O_TMPFILE** functionality.

When both **O_CREAT** and **O_DIRECTORY** are specified in *flags* and the file specified by *pathname* does not exist, **open**() will create a regular file (i.e., **O_DIRECTORY** is ignored).

SEE ALSO

chmod(2), *chown*(2), *close*(2), *dup*(2), *fcntl*(2), *link*(2), *lseek*(2), *mknod*(2), *mmap*(2), *mount*(2), *open_by_handle_at*(2), *openat2*(2), *read*(2), *socket*(2), *stat*(2), *umask*(2), *unlink*(2), *write*(2), *fopen*(3), *acl*(5), *fifo*(7), *inode*(7), *path_resolution*(7), *symlink*(7)

NAME

name_to_handle_at, open_by_handle_at − obtain handle for a pathname and open file via a handle

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>

int name_to_handle_at(int dirfd, const char *pathname,
                      struct file_handle *handle,
                      int *mount_id, int flags);
int open_by_handle_at(int mount_fd, struct file_handle *handle,
                      int flags);
```

DESCRIPTION

The **name_to_handle_at**() and **open_by_handle_at**() system calls split the functionality of *openat*(2) into two parts: **name_to_handle_at**() returns an opaque handle that corresponds to a specified file; **open_by_handle_at**() opens the file corresponding to a handle returned by a previous call to **name_to_handle_at**() and returns an open file descriptor.

name_to_handle_at()

The **name_to_handle_at**() system call returns a file handle and a mount ID corresponding to the file specified by the *dirfd* and *pathname* arguments. The file handle is returned via the argument *handle*, which is a pointer to a structure of the following form:

```
struct file_handle {
    unsigned int  handle_bytes;   /* Size of f_handle [in, out] */
    int           handle_type;    /* Handle type [out] */
    unsigned char f_handle[0];    /* File identifier (sized by
                                     caller) [out] */
};
```

It is the caller's responsibility to allocate the structure with a size large enough to hold the handle returned in *f_handle*. Before the call, the *handle_bytes* field should be initialized to contain the allocated size for *f_handle*. (The constant **MAX_HANDLE_SZ**, defined in <*fcntl.h*>, specifies the maximum expected size for a file handle. It is not a guaranteed upper limit as future filesystems may require more space.) Upon successful return, the *handle_bytes* field is updated to contain the number of bytes actually written to *f_handle*.

The caller can discover the required size for the *file_handle* structure by making a call in which *handle−>handle_bytes* is zero; in this case, the call fails with the error **EOVERFLOW** and *handle−>handle_bytes* is set to indicate the required size; the caller can then use this information to allocate a structure of the correct size (see EXAMPLES below). Some care is needed here as **EOVERFLOW** can also indicate that no file handle is available for this particular name in a filesystem which does normally support file-handle lookup. This case can be detected when the **EOVERFLOW** error is returned without *handle_bytes* being increased.

Other than the use of the *handle_bytes* field, the caller should treat the *file_handle* structure as an opaque data type: the *handle_type* and *f_handle* fields can be used in a subsequent call to **open_by_handle_at**(). The caller can also use the opaque *file_handle* to compare the identity of filesystem objects that were queried at different times and possibly at different paths. The *fanotify*(7) subsystem can report events with an information record containing a *file_handle* to identify the filesystem object.

The *flags* argument is a bit mask constructed by ORing together zero or more of **AT_HANDLE_FID**, **AT_EMPTY_PATH**, and **AT_SYMLINK_FOLLOW**, described below.

When *flags* contain the **AT_HANDLE_FID** (since Linux 6.5) flag, the caller indicates that the returned *file_handle* is needed to identify the filesystem object, and not for opening the file later, so it should be expected that a subsequent call to **open_by_handle_at**() with the returned *file_handle* may fail.

Together, the *pathname* and *dirfd* arguments identify the file for which a handle is to be obtained. There are four distinct cases:

- If *pathname* is a nonempty string containing an absolute pathname, then a handle is returned for the file referred to by that pathname. In this case, *dirfd* is ignored.
- If *pathname* is a nonempty string containing a relative pathname and *dirfd* has the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the caller, and a handle is returned for the file to which it refers.
- If *pathname* is a nonempty string containing a relative pathname and *dirfd* is a file descriptor referring to a directory, then *pathname* is interpreted relative to the directory referred to by *dirfd*, and a handle is returned for the file to which it refers. (See *openat*(2) for an explanation of why "directory file descriptors" are useful.)
- If *pathname* is an empty string and *flags* specifies the value **AT_EMPTY_PATH**, then *dirfd* can be an open file descriptor referring to any type of file, or **AT_FDCWD**, meaning the current working directory, and a handle is returned for the file to which it refers.

The *mount_id* argument returns an identifier for the filesystem mount that corresponds to *pathname*. This corresponds to the first field in one of the records in */proc/self/mountinfo*. Opening the pathname in the fifth field of that record yields a file descriptor for the mount point; that file descriptor can be used in a subsequent call to **open_by_handle_at**(). *mount_id* is returned both for a successful call and for a call that results in the error **EOVERFLOW**.

By default, **name_to_handle_at**() does not dereference *pathname* if it is a symbolic link, and thus returns a handle for the link itself. If **AT_SYMLINK_FOLLOW** is specified in *flags*, *pathname* is dereferenced if it is a symbolic link (so that the call returns a handle for the file referred to by the link).

name_to_handle_at() does not trigger a mount when the final component of the pathname is an automount point. When a filesystem supports both file handles and automount points, a **name_to_handle_at**() call on an automount point will return with error **EOVERFLOW** without having increased *handle_bytes*. This can happen since Linux 4.13 with NFS when accessing a directory which is on a separate filesystem on the server. In this case, the automount can be triggered by adding a "/" to the end of the pathname.

open_by_handle_at()

The **open_by_handle_at**() system call opens the file referred to by *handle*, a file handle returned by a previous call to **name_to_handle_at**().

The *mount_fd* argument is a file descriptor for any object (file, directory, etc.) in the mounted filesystem with respect to which *handle* should be interpreted. The special value **AT_FDCWD** can be specified, meaning the current working directory of the caller.

The *flags* argument is as for *open*(2). If *handle* refers to a symbolic link, the caller must specify the **O_PATH** flag, and the symbolic link is not dereferenced; the **O_NOFOLLOW** flag, if specified, is ignored.

The caller must have the **CAP_DAC_READ_SEARCH** capability to invoke **open_by_handle_at**().

RETURN VALUE

On success, **name_to_handle_at**() returns 0, and **open_by_handle_at**() returns a file descriptor (a nonnegative integer).

In the event of an error, both system calls return −1 and set *errno* to indicate the error.

ERRORS

name_to_handle_at() and **open_by_handle_at**() can fail for the same errors as *openat*(2). In addition, they can fail with the errors noted below.

name_to_handle_at() can fail with the following errors:

EFAULT
: *pathname*, *mount_id*, or *handle* points outside your accessible address space.

EINVAL
: *flags* includes an invalid bit value.

EINVAL
: *handle*−>*handle_bytes* is greater than **MAX_HANDLE_SZ**.

ENOENT
pathname is an empty string, but **AT_EMPTY_PATH** was not specified in *flags*.

ENOTDIR
The file descriptor supplied in *dirfd* does not refer to a directory, and it is not the case that both *flags* includes **AT_EMPTY_PATH** and *pathname* is an empty string.

EOPNOTSUPP
The filesystem does not support decoding of a pathname to a file handle.

EOVERFLOW
The *handle->handle_bytes* value passed into the call was too small. When this error occurs, *handle->handle_bytes* is updated to indicate the required size for the handle.

open_by_handle_at() can fail with the following errors:

EBADF
mount_fd is not an open file descriptor.

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
handle points outside your accessible address space.

EINVAL
handle->handle_bytes is greater than **MAX_HANDLE_SZ** or is equal to zero.

ELOOP
handle refers to a symbolic link, but **O_PATH** was not specified in *flags*.

EPERM
The caller does not have the **CAP_DAC_READ_SEARCH** capability.

ESTALE
The specified *handle* is not valid for opening a file. This error will occur if, for example, the file has been deleted. This error can also occur if the *handle* was acquired using the **AT_HANDLE_FID** flag and the filesystem does not support **open_by_handle_at**().

VERSIONS
FreeBSD has a broadly similar pair of system calls in the form of **getfh**() and **openfh**().

STANDARDS
Linux.

HISTORY
Linux 2.6.39, glibc 2.14.

NOTES
A file handle can be generated in one process using **name_to_handle_at**() and later used in a different process that calls **open_by_handle_at**().

Some filesystem don't support the translation of pathnames to file handles, for example, */proc*, */sys*, and various network filesystems. Some filesystems support the translation of pathnames to file handles, but do not support using those file handles in **open_by_handle_at**().

A file handle may become invalid ("stale") if a file is deleted, or for other filesystem-specific reasons. Invalid handles are notified by an **ESTALE** error from **open_by_handle_at**().

These system calls are designed for use by user-space file servers. For example, a user-space NFS server might generate a file handle and pass it to an NFS client. Later, when the client wants to open the file, it could pass the handle back to the server. This sort of functionality allows a user-space file server to operate in a stateless fashion with respect to the files it serves.

If *pathname* refers to a symbolic link and *flags* does not specify **AT_SYMLINK_FOLLOW**, then **name_to_handle_at**() returns a handle for the link (rather than the file to which it refers). The process receiving the handle can later perform operations on the symbolic link by converting the handle to a file descriptor using **open_by_handle_at**() with the **O_PATH** flag, and then passing the file descriptor as the *dirfd* argument in system calls such as *readlinkat*(2) and *fchownat*(2).

Obtaining a persistent filesystem ID

The mount IDs in */proc/self/mountinfo* can be reused as filesystems are unmounted and mounted. Therefore, the mount ID returned by **name_to_handle_at**() (in **mount_id*) should not be treated as a persistent identifier for the corresponding mounted filesystem. However, an application can use the information in the *mountinfo* record that corresponds to the mount ID to derive a persistent identifier.

For example, one can use the device name in the fifth field of the *mountinfo* record to search for the corresponding device UUID via the symbolic links in */dev/disks/by−uuid*. (A more comfortable way of obtaining the UUID is to use the *libblkid*(3) library.) That process can then be reversed, using the UUID to look up the device name, and then obtaining the corresponding mount point, in order to produce the *mount_fd* argument used by **open_by_handle_at**().

EXAMPLES

The two programs below demonstrate the use of **name_to_handle_at**() and **open_by_handle_at**(). The first program (*t_name_to_handle_at.c*) uses **name_to_handle_at**() to obtain the file handle and mount ID for the file specified in its command-line argument; the handle and mount ID are written to standard output.

The second program (*t_open_by_handle_at.c*) reads a mount ID and file handle from standard input. The program then employs **open_by_handle_at**() to open the file using that handle. If an optional command-line argument is supplied, then the *mount_fd* argument for **open_by_handle_at**() is obtained by opening the directory named in that argument. Otherwise, *mount_fd* is obtained by scanning */proc/self/mountinfo* to find a record whose mount ID matches the mount ID read from standard input, and the mount directory specified in that record is opened. (These programs do not deal with the fact that mount IDs are not persistent.)

The following shell session demonstrates the use of these two programs:

```
$ echo 'Can you please think about it?' > cecilia.txt
$ ./t_name_to_handle_at cecilia.txt > fh
$ ./t_open_by_handle_at < fh
open_by_handle_at: Operation not permitted
$ sudo ./t_open_by_handle_at < fh      # Need CAP_SYS_ADMIN
Read 31 bytes
$ rm cecilia.txt
```

Now we delete and (quickly) re-create the file so that it has the same content and (by chance) the same inode. Nevertheless, **open_by_handle_at**() recognizes that the original file referred to by the file handle no longer exists.

```
$ stat --printf="%i\n" cecilia.txt     # Display inode number
4072121
$ rm cecilia.txt
$ echo 'Can you please think about it?' > cecilia.txt
$ stat --printf="%i\n" cecilia.txt     # Check inode number
4072121
$ sudo ./t_open_by_handle_at < fh
open_by_handle_at: Stale NFS file handle
```

Program source: t_name_to_handle_at.c

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int                mount_id, fhsize, flags, dirfd;
    char               *pathname;
```

```
        struct file_handle  *fhp;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s pathname\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        pathname = argv[1];

        /* Allocate file_handle structure. */

        fhsize = sizeof(*fhp);
        fhp = malloc(fhsize);
        if (fhp == NULL)
            err(EXIT_FAILURE, "malloc");

        /* Make an initial call to name_to_handle_at() to discover
           the size required for file handle. */

        dirfd = AT_FDCWD;           /* For name_to_handle_at() calls */
        flags = 0;                  /* For name_to_handle_at() calls */
        fhp->handle_bytes = 0;
        if (name_to_handle_at(dirfd, pathname, fhp,
                              &mount_id, flags) != -1
            || errno != EOVERFLOW)
        {
            fprintf(stderr, "Unexpected result from name_to_handle_at()\n");
            exit(EXIT_FAILURE);
        }

        /* Reallocate file_handle structure with correct size. */

        fhsize = sizeof(*fhp) + fhp->handle_bytes;
        fhp = realloc(fhp, fhsize);         /* Copies fhp->handle_bytes */
        if (fhp == NULL)
            err(EXIT_FAILURE, "realloc");

        /* Get file handle from pathname supplied on command line. */

        if (name_to_handle_at(dirfd, pathname, fhp, &mount_id, flags) == -1)
            err(EXIT_FAILURE, "name_to_handle_at");

        /* Write mount ID, file handle size, and file handle to stdout,
           for later reuse by t_open_by_handle_at.c. */

        printf("%d\n", mount_id);
        printf("%u %d   ", fhp->handle_bytes, fhp->handle_type);
        for (size_t j = 0; j < fhp->handle_bytes; j++)
            printf(" %02x", fhp->f_handle[j]);
        printf("\n");

        exit(EXIT_SUCCESS);
    }
```

Program source: t_open_by_handle_at.c

```
    #define _GNU_SOURCE
    #include <err.h>
    #include <fcntl.h>
```

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* Scan /proc/self/mountinfo to find the line whose mount ID matches
   'mount_id'. (An easier way to do this is to install and use the
   'libmount' library provided by the 'util-linux' project.)
   Open the corresponding mount path and return the resulting file
   descriptor. */

static int
open_mount_path_by_id(int mount_id)
{
    int      mi_mount_id, found;
    char     mount_path[PATH_MAX];
    char     *linep;
    FILE     *fp;
    size_t   lsize;
    ssize_t  nread;

    fp = fopen("/proc/self/mountinfo", "r");
    if (fp == NULL)
        err(EXIT_FAILURE, "fopen");

    found = 0;
    linep = NULL;
    while (!found) {
        nread = getline(&linep, &lsize, fp);
        if (nread == -1)
            break;

        nread = sscanf(linep, "%d %*d %*s %*s %s",
                       &mi_mount_id, mount_path);
        if (nread != 2) {
            fprintf(stderr, "Bad sscanf()\n");
            exit(EXIT_FAILURE);
        }

        if (mi_mount_id == mount_id)
            found = 1;
    }
    free(linep);

    fclose(fp);

    if (!found) {
        fprintf(stderr, "Could not find mount point\n");
        exit(EXIT_FAILURE);
    }

    return open(mount_path, O_RDONLY);
}

int
main(int argc, char *argv[])
{
    int                mount_id, fd, mount_fd, handle_bytes;
```

```
    char                buf[1000];
#define LINE_SIZE 100
    char                line1[LINE_SIZE], line2[LINE_SIZE];
    char                *nextp;
    ssize_t             nread;
    struct file_handle  *fhp;

    if ((argc > 1 && strcmp(argv[1], "--help") == 0) || argc > 2) {
        fprintf(stderr, "Usage: %s [mount-path]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Standard input contains mount ID and file handle information:

         Line 1: <mount_id>
         Line 2: <handle_bytes> <handle_type>   <bytes of handle in hex>
    */

    if (fgets(line1, sizeof(line1), stdin) == NULL ||
        fgets(line2, sizeof(line2), stdin) == NULL)
    {
        fprintf(stderr, "Missing mount_id / file handle\n");
        exit(EXIT_FAILURE);
    }

    mount_id = atoi(line1);

    handle_bytes = strtoul(line2, &nextp, 0);

    /* Given handle_bytes, we can now allocate file_handle structure. */

    fhp = malloc(sizeof(*fhp) + handle_bytes);
    if (fhp == NULL)
        err(EXIT_FAILURE, "malloc");

    fhp->handle_bytes = handle_bytes;

    fhp->handle_type = strtoul(nextp, &nextp, 0);

    for (size_t j = 0; j < fhp->handle_bytes; j++)
        fhp->f_handle[j] = strtoul(nextp, &nextp, 16);

    /* Obtain file descriptor for mount point, either by opening
       the pathname specified on the command line, or by scanning
       /proc/self/mounts to find a mount that matches the 'mount_id'
       that we received from stdin. */

    if (argc > 1)
        mount_fd = open(argv[1], O_RDONLY);
    else
        mount_fd = open_mount_path_by_id(mount_id);

    if (mount_fd == -1)
        err(EXIT_FAILURE, "opening mount fd");

    /* Open file using handle and mount point. */

    fd = open_by_handle_at(mount_fd, fhp, O_RDONLY);
    if (fd == -1)
```

```
            err(EXIT_FAILURE, "open_by_handle_at");

        /* Try reading a few bytes from the file. */

        nread = read(fd, buf, sizeof(buf));
        if (nread == -1)
            err(EXIT_FAILURE, "read");

        printf("Read %zd bytes\n", nread);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

open(2), *libblkid*(3), *blkid*(8), *findfs*(8), *mount*(8)

The *libblkid* and *libmount* documentation in the latest *util-linux* release at

NAME

openat2 − open and possibly create a file (extended)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>          /* Definition of O_* and S_* constants */
#include <linux/openat2.h>  /* Definition of RESOLVE_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_openat2, int dirfd, const char *pathname,
             struct open_how *how, size_t size);
```

Note: glibc provides no wrapper for **openat2**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **openat2**() system call is an extension of *openat*(2) and provides a superset of its functionality.

The **openat2**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *how.flags*) be created.

As with *openat*(2), if *pathname* is a relative pathname, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (or the current working directory of the calling process, if *dirfd* is the special value **AT_FDCWD**). If *pathname* is an absolute pathname, then *dirfd* is ignored (unless *how.resolve* contains **RESOLVE_IN_ROOT**, in which case *pathname* is resolved relative to *dirfd*).

Rather than taking a single *flags* argument, an extensible structure (*how*) is passed to allow for future extensions. The *size* argument must be specified as *sizeof(struct open_how)*.

The open_how structure

The *how* argument specifies how *pathname* should be opened, and acts as a superset of the *flags* and *mode* arguments to *openat*(2). This argument is a pointer to an *open_how* structure, described in *open_how*(2type).

Any future extensions to **openat2**() will be implemented as new fields appended to the *open_how* structure, with a zero value in a new field resulting in the kernel behaving as though that extension field was not present. Therefore, the caller *must* zero-fill this structure on initialization. (See the "Extensibility" section of the **NOTES** for more detail on why this is necessary.)

The fields of the *open_how* structure are as follows:

flags This field specifies the file creation and file status flags to use when opening the file. All of the **O_*** flags defined for *openat*(2) are valid **openat2**() flag values.

Whereas *openat*(2) ignores unknown bits in its *flags* argument, **openat2**() returns an error if unknown or conflicting flags are specified in *how.flags*.

mode This field specifies the mode for the new file, with identical semantics to the *mode* argument of *openat*(2).

Whereas *openat*(2) ignores bits other than those in the range *07777* in its *mode* argument, **openat2**() returns an error if *how.mode* contains bits other than *07777*. Similarly, an error is returned if **openat2**() is called with a nonzero *how.mode* and *how.flags* does not contain **O_CREAT** or **O_TMPFILE**.

resolve This is a bit-mask of flags that modify the way in which **all** components of *pathname* will be resolved. (See *path_resolution*(7) for background information.)

The primary use case for these flags is to allow trusted programs to restrict how untrusted paths (or paths inside untrusted directories) are resolved. The full list of *resolve* flags is as follows:

RESOLVE_BENEATH

Do not permit the path resolution to succeed if any component of the resolution is not a descendant of the directory indicated by *dirfd*. This causes absolute symbolic links (and absolute values of *pathname*) to be rejected.

Currently, this flag also disables magic-link resolution (see below). However, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify **RESOLVE_NO_MAGICLINKS**.

RESOLVE_IN_ROOT
Treat the directory referred to by *dirfd* as the root directory while resolving *pathname*. Absolute symbolic links are interpreted relative to *dirfd*. If a prefix component of *pathname* equates to *dirfd*, then an immediately following .. component likewise equates to *dirfd* (just as */..* is traditionally equivalent to */*). If *pathname* is an absolute path, it is also interpreted relative to *dirfd*.

The effect of this flag is as though the calling process had used *chroot*(2) to (temporarily) modify its root directory (to the directory referred to by *dirfd*). However, unlike *chroot*(2) (which changes the filesystem root permanently for a process), **RESOLVE_IN_ROOT** allows a program to efficiently restrict path resolution on a per-open basis.

Currently, this flag also disables magic-link resolution. However, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify **RESOLVE_NO_MAGICLINKS**.

RESOLVE_NO_MAGICLINKS
Disallow all magic-link resolution during path resolution.

Magic links are symbolic link-like objects that are most notably found in *proc*(5); examples include */proc/*pid*/exe* and */proc/*pid*/fd/**. (See *symlink*(7) for more details.)

Unknowingly opening magic links can be risky for some applications. Examples of such risks include the following:

- If the process opening a pathname is a controlling process that currently has no controlling terminal (see *credentials*(7)), then opening a magic link inside */proc/*pid*/fd* that happens to refer to a terminal would cause the process to acquire a controlling terminal.
- In a containerized environment, a magic link inside */proc* may refer to an object outside the container, and thus may provide a means to escape from the container.

Because of such risks, an application may prefer to disable magic link resolution using the **RESOLVE_NO_MAGICLINKS** flag.

If the trailing component (i.e., basename) of *pathname* is a magic link, *how.resolve* contains **RESOLVE_NO_MAGICLINKS**, and *how.flags* contains both **O_PATH** and **O_NOFOLLOW**, then an **O_PATH** file descriptor referencing the magic link will be returned.

RESOLVE_NO_SYMLINKS
Disallow resolution of symbolic links during path resolution. This option implies **RESOLVE_NO_MAGICLINKS**.

If the trailing component (i.e., basename) of *pathname* is a symbolic link, *how.resolve* contains **RESOLVE_NO_SYMLINKS**, and *how.flags* contains both **O_PATH** and **O_NOFOLLOW**, then an **O_PATH** file descriptor referencing the symbolic link will be returned.

Note that the effect of the **RESOLVE_NO_SYMLINKS** flag, which affects the treatment of symbolic links in all of the components of *pathname*, differs from the effect of the **O_NOFOLLOW** file creation flag (in *how.flags*), which affects the handling of symbolic links only in the final component of *pathname*.

Applications that employ the **RESOLVE_NO_SYMLINKS** flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as symbolic links are very widely used by end-users. Setting this flag indiscriminately—i.e., for purposes not specifically related to security—for all uses of **openat2**() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified (e.g., in a new distribution release) so that a pathname component (now) contains a symbolic link.

RESOLVE_NO_XDEV
Disallow traversal of mount points during path resolution (including all bind mounts). Consequently, *pathname* must either be on the same mount as the directory referred to by *dirfd*, or on the same mount as the current working directory if *dirfd* is specified as **AT_FDCWD**.

Applications that employ the **RESOLVE_NO_XDEV** flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as bind mounts are widely used by end-users. Setting this flag indiscriminately—i.e., for purposes not specifically related to security—for all uses of **openat2**() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified (e.g., in a new distribution release) so that a pathname component (now) contains a bind mount.

RESOLVE_CACHED
Make the open operation fail unless all path components are already present in the kernel's lookup cache. If any kind of revalidation or I/O is needed to satisfy the lookup, **openat2**() fails with the error **EAGAIN**. This is useful in providing a fast-path open that can be performed without resorting to thread offload, or other mechanisms that an application might use to offload slower operations.

If any bits other than those listed above are set in *how.resolve*, an error is returned.

RETURN VALUE

On success, a new file descriptor is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The set of errors returned by **openat2**() includes all of the errors returned by *openat*(2), as well as the following additional errors:

E2BIG An extension that this kernel does not support was specified in *how*. (See the "Extensibility" section of **NOTES** for more detail on how extensions are handled.)

EAGAIN
how.resolve contains either **RESOLVE_IN_ROOT** or **RESOLVE_BENEATH**, and the kernel could not ensure that a ".." component didn't escape (due to a race condition or potential attack). The caller may choose to retry the **openat2**() call.

EAGAIN
RESOLVE_CACHED was set, and the open operation cannot be performed using only cached information. The caller should retry without **RESOLVE_CACHED** set in *how.resolve*.

EINVAL
An unknown flag or invalid value was specified in *how*.

EINVAL
mode is nonzero, but *how.flags* does not contain **O_CREAT** or **O_TMPFILE**.

EINVAL
size was smaller than any known version of *struct open_how*.

ELOOP
how.resolve contains **RESOLVE_NO_SYMLINKS**, and one of the path components was a symbolic link (or magic link).

ELOOP
how.resolve contains **RESOLVE_NO_MAGICLINKS**, and one of the path components was a magic link.

EXDEV
how.resolve contains either **RESOLVE_IN_ROOT** or **RESOLVE_BENEATH**, and an escape from the root during path resolution was detected.

EXDEV
how.resolve contains **RESOLVE_NO_XDEV**, and a path component crosses a mount point.

STANDARDS

Linux.

HISTORY

Linux 5.6.

The semantics of **RESOLVE_BENEATH** were modeled after FreeBSD's **O_BENEATH**.

NOTES

Extensibility

In order to allow for future extensibility, **openat2**() requires the user-space application to specify the size of the *open_how* structure that it is passing. By providing this information, it is possible for **openat2**() to provide both forwards- and backwards-compatibility, with *size* acting as an implicit version number. (Because new extension fields will always be appended, the structure size will always increase.) This extensibility design is very similar to other system calls such as *sched_setattr*(2), *perf_event_open*(2), and *clone3*(2).

If we let *usize* be the size of the structure as specified by the user-space application, and *ksize* be the size of the structure which the kernel supports, then there are three cases to consider:

- If *ksize* equals *usize*, then there is no version mismatch and *how* can be used verbatim.
- If *ksize* is larger than *usize*, then there are some extension fields that the kernel supports which the user-space application is unaware of. Because a zero value in any added extension field signifies a no-op, the kernel treats all of the extension fields not provided by the user-space application as having zero values. This provides backwards-compatibility.
- If *ksize* is smaller than *usize*, then there are some extension fields which the user-space application is aware of but which the kernel does not support. Because any extension field must have its zero values signify a no-op, the kernel can safely ignore the unsupported extension fields if they are all-zero. If any unsupported extension fields are nonzero, then −1 is returned and *errno* is set to **E2BIG**. This provides forwards-compatibility.

Because the definition of *struct open_how* may change in the future (with new fields being added when system headers are updated), user-space applications should zero-fill *struct open_how* to ensure that recompiling the program with new headers will not result in spurious errors at run time. The simplest way is to use a designated initializer:

```
struct open_how how = { .flags = O_RDWR,
                        .resolve = RESOLVE_IN_ROOT };
```

or explicitly using *memset*(3) or similar:

```
struct open_how how;
memset(&how, 0, sizeof(how));
how.flags = O_RDWR;
how.resolve = RESOLVE_IN_ROOT;
```

A user-space application that wishes to determine which extensions the running kernel supports can do so by conducting a binary search on *size* with a structure which has every byte nonzero (to find the largest value which doesn't produce an error of **E2BIG**).

SEE ALSO

openat(2), *open_how*(2type), *path_resolution*(7), *symlink*(7)

NAME

outb, outw, outl, outsb, outsw, outsl, inb, inw, inl, insb, insw, insl, outb_p, outw_p, outl_p, inb_p, inw_p, inl_p − port I/O

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/io.h>

unsigned char inb(unsigned short port);
unsigned char inb_p(unsigned short port);
unsigned short inw(unsigned short port);
unsigned short inw_p(unsigned short port);
unsigned int inl(unsigned short port);
unsigned int inl_p(unsigned short port);

void outb(unsigned char value, unsigned short port);
void outb_p(unsigned char value, unsigned short port);
void outw(unsigned short value, unsigned short port);
void outw_p(unsigned short value, unsigned short port);
void outl(unsigned int value, unsigned short port);
void outl_p(unsigned int value, unsigned short port);

void insb(unsigned short port, void addr[.count],
        unsigned long count);
void insw(unsigned short port, void addr[.count],
        unsigned long count);
void insl(unsigned short port, void addr[.count],
        unsigned long count);
void outsb(unsigned short port, const void addr[.count],
        unsigned long count);
void outsw(unsigned short port, const void addr[.count],
        unsigned long count);
void outsl(unsigned short port, const void addr[.count],
        unsigned long count);
```

DESCRIPTION

This family of functions is used to do low-level port input and output. The out* functions do port output, the in* functions do port input; the b-suffix functions are byte-width and the w-suffix functions word-width; the _p-suffix functions pause until the I/O completes.

They are primarily designed for internal kernel use, but can be used from user space.

You must compile with **−O** or **−O2** or similar. The functions are defined as inline macros, and will not be substituted in without optimization enabled, causing unresolved references at link time.

You use *ioperm*(2) or alternatively *iopl*(2) to tell the kernel to allow the user space application to access the I/O ports in question. Failure to do this will cause the application to receive a segmentation fault.

VERSIONS

outb() and friends are hardware-specific. The *value* argument is passed first and the *port* argument is passed second, which is the opposite order from most DOS implementations.

STANDARDS

None.

SEE ALSO

ioperm(2), *iopl*(2)

NAME

pause − wait for signal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

pause() causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function.

RETURN VALUE

pause() returns only when a signal was caught and the signal-catching function returned. In this case, **pause**() returns −1, and *errno* is set to **EINTR**.

ERRORS

EINTR
: a signal was caught and the signal-catching function returned.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

kill(2), *select*(2), *signal*(2), *sigsuspend*(2)

NAME

pciconfig_read, pciconfig_write, pciconfig_iobase − pci device information handling

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <pci.h>

int pciconfig_read(unsigned long bus, unsigned long dfn,
                   unsigned long off, unsigned long len,
                   unsigned char *buf);
int pciconfig_write(unsigned long bus, unsigned long dfn,
                   unsigned long off, unsigned long len,
                   unsigned char *buf);
int pciconfig_iobase(int which, unsigned long bus,
                   unsigned long devfn);
```

DESCRIPTION

Most of the interaction with PCI devices is already handled by the kernel PCI layer, and thus these calls should not normally need to be accessed from user space.

pciconfig_read()
: Reads to *buf* from device *dev* at offset *off* value.

pciconfig_write()
: Writes from *buf* to device *dev* at offset *off* value.

pciconfig_iobase()
: You pass it a bus/devfn pair and get a physical address for either the memory offset (for things like prep, this is 0xc0000000), the IO base for PIO cycles, or the ISA holes if any.

RETURN VALUE

pciconfig_read()
: On success, zero is returned. On error, −1 is returned and *errno* is set to indicate the error.

pciconfig_write()
: On success, zero is returned. On error, −1 is returned and *errno* is set to indicate the error.

pciconfig_iobase()
: Returns information on locations of various I/O regions in physical memory according to the *which* value. Values for *which* are: **IOBASE_BRIDGE_NUMBER**, **IOBASE_MEMORY**, **IOBASE_IO**, **IOBASE_ISA_IO**, **IOBASE_ISA_MEM**.

ERRORS

EINVAL
: *len* value is invalid. This does not apply to **pciconfig_iobase**().

EIO I/O error.

ENODEV
: For **pciconfig_iobase**(), "hose" value is NULL. For the other calls, could not find a slot.

ENOSYS
: The system has not implemented these calls (**CONFIG_PCI** not defined).

EOPNOTSUPP
: This return value is valid only for **pciconfig_iobase**(). It is returned if the value for *which* is invalid.

EPERM
: User does not have the **CAP_SYS_ADMIN** capability. This does not apply to **pciconfig_iobase**().

STANDARDS

Linux.

HISTORY

Linux 2.0.26/2.1.11.

SEE ALSO

capabilities(7)

NAME

perf_event_open − set up performance monitoring

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/perf_event.h>    /* Definition of PERF_* constants */
#include <linux/hw_breakpoint.h> /* Definition of HW_* constants */
#include <sys/syscall.h>         /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_perf_event_open, struct perf_event_attr *attr,
            pid_t pid, int cpu, int group_fd, unsigned long flags);
```

Note: glibc provides no wrapper for **perf_event_open**(), necessitating the use of *syscall*(2).

DESCRIPTION

Given a list of parameters, **perf_event_open**() returns a file descriptor, for use in subsequent system calls (*read*(2), *mmap*(2), *prctl*(2), *fcntl*(2), etc.).

A call to **perf_event_open**() creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured; these can be grouped together to measure multiple events simultaneously.

Events can be enabled and disabled in two ways: via *ioctl*(2) and via *prctl*(2). When an event is disabled it does not count or generate overflows but does continue to exist and maintain its count value.

Events come in two flavors: counting and sampled. A *counting* event is one that is used for counting the aggregate number of events that occur. In general, counting event results are gathered with a *read*(2) call. A *sampling* event periodically writes measurements to a buffer that can then be accessed via *mmap*(2).

Arguments

The *pid* and *cpu* arguments allow specifying which process and CPU to monitor:

pid == 0 and **cpu == -1**
: This measures the calling process/thread on any CPU.

pid == 0 and **cpu >= 0**
: This measures the calling process/thread only when running on the specified CPU.

pid > 0 and **cpu == -1**
: This measures the specified process/thread on any CPU.

pid > 0 and **cpu >= 0**
: This measures the specified process/thread only when running on the specified CPU.

pid == -1 and **cpu >= 0**
: This measures all processes/threads on the specified CPU. This requires **CAP_PERFMON** (since Linux 5.8) or **CAP_SYS_ADMIN** capability or a */proc/sys/kernel/perf_event_paranoid* value of less than 1.

pid == -1 and **cpu == -1**
: This setting is invalid and will return an error.

When *pid* is greater than zero, permission to perform this system call is governed by **CAP_PERFMON** (since Linux 5.9) and a ptrace access mode **PTRACE_MODE_READ_REALCREDS** check on older Linux versions; see *ptrace*(2).

The *group_fd* argument allows event groups to be created. An event group has one event which is the group leader. The leader is created first, with *group_fd* = −1. The rest of the group members are created with subsequent **perf_event_open**() calls with *group_fd* being set to the file descriptor of the group leader. (A single event on its own is created with *group_fd* = −1 and is considered to be a group with only 1 member.) An event group is scheduled onto the CPU as a unit: it will be put onto the CPU only if all of the events in the group can be put onto the CPU. This means that the values of the member events can be meaningfully compared —added, divided (to get ratios), and so on— with each other, since they have counted events for the same set of executed instructions.

The *flags* argument is formed by ORing together zero or more of the following values:

PERF_FLAG_FD_CLOEXEC (since Linux 3.14)
This flag enables the close-on-exec flag for the created event file descriptor, so that the file descriptor is automatically closed on *execve*(2). Setting the close-on-exec flags at creation time, rather than later with *fcntl*(2), avoids potential race conditions where the calling thread invokes **perf_event_open**() and *fcntl*(2) at the same time as another thread calls *fork*(2) then *execve*(2).

PERF_FLAG_FD_NO_GROUP
This flag tells the event to ignore the *group_fd* parameter except for the purpose of setting up output redirection using the **PERF_FLAG_FD_OUTPUT** flag.

PERF_FLAG_FD_OUTPUT (broken since Linux 2.6.35)
This flag re-routes the event's sampled output to instead be included in the mmap buffer of the event specified by *group_fd*.

PERF_FLAG_PID_CGROUP (since Linux 2.6.39)
This flag activates per-container system-wide monitoring. A container is an abstraction that isolates a set of resources for finer-grained control (CPUs, memory, etc.). In this mode, the event is measured only if the thread running on the monitored CPU belongs to the designated container (cgroup). The cgroup is identified by passing a file descriptor opened on its directory in the cgroupfs filesystem. For instance, if the cgroup to monitor is called *test*, then a file descriptor opened on */dev/cgroup/test* (assuming cgroupfs is mounted on */dev/cgroup*) must be passed as the *pid* parameter. cgroup monitoring is available only for system-wide events and may therefore require extra permissions.

The *perf_event_attr* structure provides detailed configuration information for the event being created.

```
struct perf_event_attr {
    __u32 type;                 /* Type of event */
    __u32 size;                 /* Size of attribute structure */
    __u64 config;               /* Type-specific configuration */

    union {
        __u64 sample_period;    /* Period of sampling */
        __u64 sample_freq;      /* Frequency of sampling */
    };

    __u64 sample_type;  /* Specifies values included in sample */
    __u64 read_format;  /* Specifies values returned in read */

    __u64 disabled       : 1,   /* off by default */
          inherit        : 1,   /* children inherit it */
          pinned         : 1,   /* must always be on PMU */
          exclusive      : 1,   /* only group on PMU */
          exclude_user   : 1,   /* don't count user */
          exclude_kernel : 1,   /* don't count kernel */
          exclude_hv     : 1,   /* don't count hypervisor */
          exclude_idle   : 1,   /* don't count when idle */
          mmap           : 1,   /* include mmap data */
          comm           : 1,   /* include comm data */
          freq           : 1,   /* use freq, not period */
          inherit_stat   : 1,   /* per task counts */
          enable_on_exec : 1,   /* next exec enables */
          task           : 1,   /* trace fork/exit */
          watermark      : 1,   /* wakeup_watermark */
          precise_ip     : 2,   /* skid constraint */
          mmap_data      : 1,   /* non-exec mmap data */
          sample_id_all  : 1,   /* sample_type all events */
          exclude_host   : 1,   /* don't count in host */
          exclude_guest  : 1,   /* don't count in guest */
          exclude_callchain_kernel : 1,
```

```
                            /* exclude kernel callchains */
        exclude_callchain_user   : 1,
                            /* exclude user callchains */
        mmap2          :  1,  /* include mmap with inode data */
        comm_exec      :  1,  /* flag comm events that are
                                 due to exec */
        use_clockid    :  1,  /* use clockid for time fields */
        context_switch :  1,  /* context switch data */
        write_backward :  1,  /* Write ring buffer from end
                                 to beginning */
        namespaces     :  1,  /* include namespaces data */
        ksymbol        :  1,  /* include ksymbol events */
        bpf_event      :  1,  /* include bpf events */
        aux_output     :  1,  /* generate AUX records
                                 instead of events */
        cgroup         :  1,  /* include cgroup events */
        text_poke      :  1,  /* include text poke events */
        build_id       :  1,  /* use build id in mmap2 events */
        inherit_thread :  1,  /* children only inherit */
                            /* if cloned with CLONE_THREAD */
        remove_on_exec :  1,  /* event is removed from task
                                 on exec */
        sigtrap        :  1,  /* send synchronous SIGTRAP
                                 on event */

        __reserved_1   : 26;

    union {
        __u32 wakeup_events;    /* wakeup every n events */
        __u32 wakeup_watermark; /* bytes before wakeup */
    };

    __u32     bp_type;          /* breakpoint type */

    union {
        __u64 bp_addr;          /* breakpoint address */
        __u64 kprobe_func;      /* for perf_kprobe */
        __u64 uprobe_path;      /* for perf_uprobe */
        __u64 config1;          /* extension of config */
    };

    union {
        __u64 bp_len;           /* breakpoint length */
        __u64 kprobe_addr;      /* with kprobe_func == NULL */
        __u64 probe_offset;     /* for perf_[k,u]probe */
        __u64 config2;          /* extension of config1 */
    };
    __u64 branch_sample_type;   /* enum perf_branch_sample_type */
    __u64 sample_regs_user;     /* user regs to dump on samples */
    __u32 sample_stack_user;    /* size of stack to dump on
                                   samples */
    __s32 clockid;              /* clock to use for time fields */
    __u64 sample_regs_intr;     /* regs to dump on samples */
    __u32 aux_watermark;        /* aux bytes before wakeup */
    __u16 sample_max_stack;     /* max frames in callchain */
    __u16 __reserved_2;         /* align to u64 */
    __u32 aux_sample_size;      /* max aux sample size */
    __u32 __reserved_3;         /* align to u64 */
    __u64 sig_data;             /* user data for sigtrap */
```

```
};
```

The fields of the *perf_event_attr* structure are described in more detail below:

type This field specifies the overall event type. It has one of the following values:

PERF_TYPE_HARDWARE
This indicates one of the "generalized" hardware events provided by the kernel. See the *config* field definition for more details.

PERF_TYPE_SOFTWARE
This indicates one of the software-defined events provided by the kernel (even if no hardware support is available).

PERF_TYPE_TRACEPOINT
This indicates a tracepoint provided by the kernel tracepoint infrastructure.

PERF_TYPE_HW_CACHE
This indicates a hardware cache event. This has a special encoding, described in the *config* field definition.

PERF_TYPE_RAW
This indicates a "raw" implementation-specific event in the *config* field.

PERF_TYPE_BREAKPOINT (since Linux 2.6.33)
This indicates a hardware breakpoint as provided by the CPU. Breakpoints can be read/write accesses to an address as well as execution of an instruction address.

dynamic PMU
Since Linux 2.6.38, **perf_event_open**() can support multiple PMUs. To enable this, a value exported by the kernel can be used in the *type* field to indicate which PMU to use. The value to use can be found in the sysfs filesystem: there is a subdirectory per PMU instance under */sys/bus/event_source/devices*. In each subdirectory there is a *type* file whose content is an integer that can be used in the *type* field. For instance, */sys/bus/event_source/devices/cpu/type* contains the value for the core CPU PMU, which is usually 4.

kprobe and **uprobe** (since Linux 4.17)
These two dynamic PMUs create a kprobe/uprobe and attach it to the file descriptor generated by perf_event_open. The kprobe/uprobe will be destroyed on the destruction of the file descriptor. See fields *kprobe_func*, *uprobe_path*, *kprobe_addr*, and *probe_offset* for more details.

size The size of the *perf_event_attr* structure for forward/backward compatibility. Set this using *sizeof(struct perf_event_attr)* to allow the kernel to see the struct size at the time of compilation.

The related define **PERF_ATTR_SIZE_VER0** is set to 64; this was the size of the first published struct. **PERF_ATTR_SIZE_VER1** is 72, corresponding to the addition of breakpoints in Linux 2.6.33. **PERF_ATTR_SIZE_VER2** is 80 corresponding to the addition of branch sampling in Linux 3.4. **PERF_ATTR_SIZE_VER3** is 96 corresponding to the addition of *sample_regs_user* and *sample_stack_user* in Linux 3.7. **PERF_ATTR_SIZE_VER4** is 104 corresponding to the addition of *sample_regs_intr* in Linux 3.19. **PERF_ATTR_SIZE_VER5** is 112 corresponding to the addition of *aux_watermark* in Linux 4.1.

config This specifies which event you want, in conjunction with the *type* field. The *config1* and *config2* fields are also taken into account in cases where 64 bits is not enough to fully specify the event. The encoding of these fields are event dependent.

There are various ways to set the *config* field that are dependent on the value of the previously described *type* field. What follows are various possible settings for *config* separated out by *type*.

If *type* is **PERF_TYPE_HARDWARE**, we are measuring one of the generalized hardware CPU events. Not all of these are available on all platforms. Set *config* to one of the following:

PERF_COUNT_HW_CPU_CYCLES
Total cycles. Be wary of what happens during CPU frequency scaling.

PERF_COUNT_HW_INSTRUCTIONS
Retired instructions. Be careful, these can be affected by various issues, most notably hardware interrupt counts.

PERF_COUNT_HW_CACHE_REFERENCES
Cache accesses. Usually this indicates Last Level Cache accesses but this may vary depending on your CPU. This may include prefetches and coherency messages; again this depends on the design of your CPU.

PERF_COUNT_HW_CACHE_MISSES
Cache misses. Usually this indicates Last Level Cache misses; this is intended to be used in conjunction with the **PERF_COUNT_HW_CACHE_REFERENCES** event to calculate cache miss rates.

PERF_COUNT_HW_BRANCH_INSTRUCTIONS
Retired branch instructions. Prior to Linux 2.6.35, this used the wrong event on AMD processors.

PERF_COUNT_HW_BRANCH_MISSES
Mispredicted branch instructions.

PERF_COUNT_HW_BUS_CYCLES
Bus cycles, which can be different from total cycles.

PERF_COUNT_HW_STALLED_CYCLES_FRONTEND (since Linux 3.0)
Stalled cycles during issue.

PERF_COUNT_HW_STALLED_CYCLES_BACKEND (since Linux 3.0)
Stalled cycles during retirement.

PERF_COUNT_HW_REF_CPU_CYCLES (since Linux 3.3)
Total cycles; not affected by CPU frequency scaling.

If *type* is **PERF_TYPE_SOFTWARE**, we are measuring software events provided by the kernel. Set *config* to one of the following:

PERF_COUNT_SW_CPU_CLOCK
This reports the CPU clock, a high-resolution per-CPU timer.

PERF_COUNT_SW_TASK_CLOCK
This reports a clock count specific to the task that is running.

PERF_COUNT_SW_PAGE_FAULTS
This reports the number of page faults.

PERF_COUNT_SW_CONTEXT_SWITCHES
This counts context switches. Until Linux 2.6.34, these were all reported as user-space events, after that they are reported as happening in the kernel.

PERF_COUNT_SW_CPU_MIGRATIONS
This reports the number of times the process has migrated to a new CPU.

PERF_COUNT_SW_PAGE_FAULTS_MIN
This counts the number of minor page faults. These did not require disk I/O to handle.

PERF_COUNT_SW_PAGE_FAULTS_MAJ
This counts the number of major page faults. These required disk I/O to handle.

PERF_COUNT_SW_ALIGNMENT_FAULTS (since Linux 2.6.33)
This counts the number of alignment faults. These happen when unaligned memory accesses happen; the kernel can handle these but it reduces performance. This happens only on some architectures (never on x86).

PERF_COUNT_SW_EMULATION_FAULTS (since Linux 2.6.33)
This counts the number of emulation faults. The kernel sometimes traps on unimplemented instructions and emulates them for user space. This can negatively impact performance.

PERF_COUNT_SW_DUMMY (since Linux 3.12)
This is a placeholder event that counts nothing. Informational sample record types such as mmap or comm must be associated with an active event. This dummy event allows gathering such records without requiring a counting event.

PERF_COUNT_SW_BPF_OUTPUT (since Linux 4.4)
This is used to generate raw sample data from BPF. BPF programs can write to this event using **bpf_perf_event_output** helper.

PERF_COUNT_SW_CGROUP_SWITCHES (since Linux 5.13)
This counts context switches to a task in a different cgroup. In other words, if the next task is in the same cgroup, it won't count the switch.

If *type* is **PERF_TYPE_TRACEPOINT**, then we are measuring kernel tracepoints. The value to use in *config* can be obtained from under debugfs *tracing/events/*/*/id* if ftrace is enabled in the kernel.

If *type* is **PERF_TYPE_HW_CACHE**, then we are measuring a hardware CPU cache event. To calculate the appropriate *config* value, use the following equation:

```
config = (perf_hw_cache_id) |
         (perf_hw_cache_op_id << 8) |
         (perf_hw_cache_op_result_id << 16);
```

where *perf_hw_cache_id* is one of:

PERF_COUNT_HW_CACHE_L1D
for measuring Level 1 Data Cache

PERF_COUNT_HW_CACHE_L1I
for measuring Level 1 Instruction Cache

PERF_COUNT_HW_CACHE_LL
for measuring Last-Level Cache

PERF_COUNT_HW_CACHE_DTLB
for measuring the Data TLB

PERF_COUNT_HW_CACHE_ITLB
for measuring the Instruction TLB

PERF_COUNT_HW_CACHE_BPU
for measuring the branch prediction unit

PERF_COUNT_HW_CACHE_NODE (since Linux 3.1)
for measuring local memory accesses

and *perf_hw_cache_op_id* is one of:

PERF_COUNT_HW_CACHE_OP_READ
for read accesses

PERF_COUNT_HW_CACHE_OP_WRITE
for write accesses

PERF_COUNT_HW_CACHE_OP_PREFETCH
for prefetch accesses

and *perf_hw_cache_op_result_id* is one of:

PERF_COUNT_HW_CACHE_RESULT_ACCESS
to measure accesses

PERF_COUNT_HW_CACHE_RESULT_MISS
to measure misses

If *type* is **PERF_TYPE_RAW**, then a custom "raw" *config* value is needed. Most CPUs

support events that are not covered by the "generalized" events. These are implementation defined; see your CPU manual (for example the Intel Volume 3B documentation or the AMD BIOS and Kernel Developer Guide). The libpfm4 library can be used to translate from the name in the architectural manuals to the raw hex value **perf_event_open**() expects in this field.

If *type* is **PERF_TYPE_BREAKPOINT**, then leave *config* set to zero. Its parameters are set in other places.

If *type* is **kprobe** or **uprobe**, set *retprobe* (bit 0 of *config*, see */sys/bus/event_source/devices/[k,u]probe/format/retprobe*) for kretprobe/uretprobe. See fields *kprobe_func*, *uprobe_path*, *kprobe_addr*, and *probe_offset* for more details.

kprobe_func
uprobe_path
kprobe_addr
probe_offset
: These fields describe the kprobe/uprobe for dynamic PMUs **kprobe** and **uprobe**. For **kprobe**: use *kprobe_func* and *probe_offset*, or use *kprobe_addr* and leave *kprobe_func* as NULL. For **uprobe**: use *uprobe_path* and *probe_offset*.

sample_period
sample_freq
: A "sampling" event is one that generates an overflow notification every N events, where N is given by *sample_period*. A sampling event has *sample_period* > 0. When an overflow occurs, requested data is recorded in the mmap buffer. The *sample_type* field controls what data is recorded on each overflow.

sample_freq can be used if you wish to use frequency rather than period. In this case, you set the *freq* flag. The kernel will adjust the sampling period to try and achieve the desired rate. The rate of adjustment is a timer tick.

sample_type
: The various bits in this field specify which values to include in the sample. They will be recorded in a ring-buffer, which is available to user space using *mmap*(2). The order in which the values are saved in the sample are documented in the MMAP Layout subsection below; it is not the *enum perf_event_sample_format* order.

PERF_SAMPLE_IP
: Records instruction pointer.

PERF_SAMPLE_TID
: Records the process and thread IDs.

PERF_SAMPLE_TIME
: Records a timestamp.

PERF_SAMPLE_ADDR
: Records an address, if applicable.

PERF_SAMPLE_READ
: Record counter values for all events in a group, not just the group leader.

PERF_SAMPLE_CALLCHAIN
: Records the callchain (stack backtrace).

PERF_SAMPLE_ID
: Records a unique ID for the opened event's group leader.

PERF_SAMPLE_CPU
: Records CPU number.

PERF_SAMPLE_PERIOD
: Records the current sampling period.

PERF_SAMPLE_STREAM_ID
: Records a unique ID for the opened event. Unlike **PERF_SAMPLE_ID** the actual ID is returned, not the group leader. This ID is the same as the one returned by

PERF_FORMAT_ID.

PERF_SAMPLE_RAW
Records additional data, if applicable. Usually returned by tracepoint events.

PERF_SAMPLE_BRANCH_STACK (since Linux 3.4)
This provides a record of recent branches, as provided by CPU branch sampling hardware (such as Intel Last Branch Record). Not all hardware supports this feature.

See the *branch_sample_type* field for how to filter which branches are reported.

PERF_SAMPLE_REGS_USER (since Linux 3.7)
Records the current user-level CPU register state (the values in the process before the kernel was called).

PERF_SAMPLE_STACK_USER (since Linux 3.7)
Records the user level stack, allowing stack unwinding.

PERF_SAMPLE_WEIGHT (since Linux 3.10)
Records a hardware provided weight value that expresses how costly the sampled event was. This allows the hardware to highlight expensive events in a profile.

PERF_SAMPLE_DATA_SRC (since Linux 3.10)
Records the data source: where in the memory hierarchy the data associated with the sampled instruction came from. This is available only if the underlying hardware supports this feature.

PERF_SAMPLE_IDENTIFIER (since Linux 3.12)
Places the **SAMPLE_ID** value in a fixed position in the record, either at the beginning (for sample events) or at the end (if a non-sample event).

This was necessary because a sample stream may have records from various different event sources with different *sample_type* settings. Parsing the event stream properly was not possible because the format of the record was needed to find **SAMPLE_ID**, but the format could not be found without knowing what event the sample belonged to (causing a circular dependency).

The **PERF_SAMPLE_IDENTIFIER** setting makes the event stream always parsable by putting **SAMPLE_ID** in a fixed location, even though it means having duplicate **SAMPLE_ID** values in records.

PERF_SAMPLE_TRANSACTION (since Linux 3.13)
Records reasons for transactional memory abort events (for example, from Intel TSX transactional memory support).

The *precise_ip* setting must be greater than 0 and a transactional memory abort event must be measured or no values will be recorded. Also note that some perf_event measurements, such as sampled cycle counting, may cause extraneous aborts (by causing an interrupt during a transaction).

PERF_SAMPLE_REGS_INTR (since Linux 3.19)
Records a subset of the current CPU register state as specified by *sample_regs_intr*. Unlike **PERF_SAMPLE_REGS_USER** the register values will return kernel register state if the overflow happened while kernel code is running. If the CPU supports hardware sampling of register state (i.e., PEBS on Intel x86) and *precise_ip* is set higher than zero then the register values returned are those captured by hardware at the time of the sampled instruction's retirement.

PERF_SAMPLE_PHYS_ADDR (since Linux 4.13)
Records physical address of data like in **PERF_SAMPLE_ADDR**.

PERF_SAMPLE_CGROUP (since Linux 5.7)
Records (perf_event) cgroup ID of the process. This corresponds to the *id* field in the **PERF_RECORD_CGROUP** event.

PERF_SAMPLE_DATA_PAGE_SIZE (since Linux 5.11)
Records page size of data like in **PERF_SAMPLE_ADDR**.

PERF_SAMPLE_CODE_PAGE_SIZE (since Linux 5.11)
Records page size of ip like in **PERF_SAMPLE_IP**.

PERF_SAMPLE_WEIGHT_STRUCT (since Linux 5.12)
Records hardware provided weight values like in **PERF_SAMPLE_WEIGHT**, but it can represent multiple values in a struct. This shares the same space as **PERF_SAMPLE_WEIGHT**, so users can apply either of those, not both. It has the following format and the meaning of each field is dependent on the hardware implementation.

```
union perf_sample_weight {
    u64  full;           /* PERF_SAMPLE_WEIGHT */
    struct {             /* PERF_SAMPLE_WEIGHT_STRUCT */
        u32  var1_dw;
        u16  var2_w;
        u16  var3_w;
    };
};
```

read_format
This field specifies the format of the data returned by *read*(2) on a **perf_event_open**() file descriptor.

PERF_FORMAT_TOTAL_TIME_ENABLED
Adds the 64-bit *time_enabled* field. This can be used to calculate estimated totals if the PMU is overcommitted and multiplexing is happening.

PERF_FORMAT_TOTAL_TIME_RUNNING
Adds the 64-bit *time_running* field. This can be used to calculate estimated totals if the PMU is overcommitted and multiplexing is happening.

PERF_FORMAT_ID
Adds a 64-bit unique value that corresponds to the event group.

PERF_FORMAT_GROUP
Allows all counter values in an event group to be read with one read.

PERF_FORMAT_LOST (since Linux 6.0)
Adds a 64-bit value that is the number of lost samples for this event. This would be only meaningful when *sample_period* or *sample_freq* is set.

disabled
The *disabled* bit specifies whether the counter starts out disabled or enabled. If disabled, the event can later be enabled by *ioctl*(2), *prctl*(2), or *enable_on_exec*.

When creating an event group, typically the group leader is initialized with *disabled* set to 1 and any child events are initialized with *disabled* set to 0. Despite *disabled* being 0, the child events will not start until the group leader is enabled.

inherit The *inherit* bit specifies that this counter should count events of child tasks as well as the task specified. This applies only to new children, not to any existing children at the time the counter is created (nor to any new children of existing children).

Inherit does not work for some combinations of *read_format* values, such as **PERF_FORMAT_GROUP**.

pinned The *pinned* bit specifies that the counter should always be on the CPU if at all possible. It applies only to hardware counters and only to group leaders. If a pinned counter cannot be put onto the CPU (e.g., because there are not enough hardware counters or because of a conflict with some other event), then the counter goes into an 'error' state, where reads return end-of-file (i.e., *read*(2) returns 0) until the counter is subsequently enabled or disabled.

exclusive
The *exclusive* bit specifies that when this counter's group is on the CPU, it should be the only group using the CPU's counters. In the future this may allow monitoring programs to support PMU features that need to run alone so that they do not disrupt other hardware counters.

Note that many unexpected situations may prevent events with the *exclusive* bit set from ever running. This includes any users running a system-wide measurement as well as any kernel use of the performance counters (including the commonly enabled NMI Watchdog Timer interface).

exclude_user
If this bit is set, the count excludes events that happen in user space.

exclude_kernel
If this bit is set, the count excludes events that happen in kernel space.

exclude_hv
If this bit is set, the count excludes events that happen in the hypervisor. This is mainly for PMUs that have built-in support for handling this (such as POWER). Extra support is needed for handling hypervisor measurements on most machines.

exclude_idle
If set, don't count when the CPU is running the idle task. While you can currently enable this for any event type, it is ignored for all but software events.

mmap The *mmap* bit enables generation of **PERF_RECORD_MMAP** samples for every *mmap*(2) call that has **PROT_EXEC** set. This allows tools to notice new executable code being mapped into a program (dynamic shared libraries for example) so that addresses can be mapped back to the original code.

comm The *comm* bit enables tracking of process command name as modified by the *execve*(2) and *prctl*(PR_SET_NAME) system calls as well as writing to */proc/self/comm*. If the *comm_exec* flag is also successfully set (possible since Linux 3.16), then the misc flag **PERF_RECORD_MISC_COMM_EXEC** can be used to differentiate the *execve*(2) case from the others.

freq If this bit is set, then *sample_frequency* not *sample_period* is used when setting up the sampling interval.

inherit_stat
This bit enables saving of event counts on context switch for inherited tasks. This is meaningful only if the *inherit* field is set.

enable_on_exec
If this bit is set, a counter is automatically enabled after a call to *execve*(2).

task If this bit is set, then fork/exit notifications are included in the ring buffer.

watermark
If set, have an overflow notification happen when we cross the *wakeup_watermark* boundary. Otherwise, overflow notifications happen after *wakeup_events* samples.

precise_ip (since Linux 2.6.35)
This controls the amount of skid. Skid is how many instructions execute between an event of interest happening and the kernel being able to stop and record the event. Smaller skid is better and allows more accurate reporting of which events correspond to which instructions, but hardware is often limited with how small this can be.

The possible values of this field are the following:

0 **SAMPLE_IP** can have arbitrary skid.

1 **SAMPLE_IP** must have constant skid.

2 **SAMPLE_IP** requested to have 0 skid.

3 **SAMPLE_IP** must have 0 skid. See also the description of **PERF_RECORD_MISC_EXACT_IP**.

mmap_data (since Linux 2.6.36)
This is the counterpart of the *mmap* field. This enables generation of **PERF_RECORD_MMAP** samples for *mmap*(2) calls that do not have **PROT_EXEC** set (for example data and SysV shared memory).

sample_id_all (since Linux 2.6.38)
: If set, then TID, TIME, ID, STREAM_ID, and CPU can additionally be included in non-**PERF_RECORD_SAMPLE**s if the corresponding *sample_type* is selected.

  If **PERF_SAMPLE_IDENTIFIER** is specified, then an additional ID value is included as the last value to ease parsing the record stream. This may lead to the *id* value appearing twice.

  The layout is described by this pseudo-structure:

```
struct sample_id {
    { u32 pid, tid; }   /* if PERF_SAMPLE_TID set */
    { u64 time;     }   /* if PERF_SAMPLE_TIME set */
    { u64 id;       }   /* if PERF_SAMPLE_ID set */
    { u64 stream_id;}   /* if PERF_SAMPLE_STREAM_ID set  */
    { u32 cpu, res; }   /* if PERF_SAMPLE_CPU set */
    { u64 id;       }   /* if PERF_SAMPLE_IDENTIFIER set */
};
```

exclude_host (since Linux 3.2)
: When conducting measurements that include processes running VM instances (i.e., have executed a **KVM_RUN** *ioctl*(2)), only measure events happening inside a guest instance. This is only meaningful outside the guests; this setting does not change counts gathered inside of a guest. Currently, this functionality is x86 only.

exclude_guest (since Linux 3.2)
: When conducting measurements that include processes running VM instances (i.e., have executed a **KVM_RUN** *ioctl*(2)), do not measure events happening inside guest instances. This is only meaningful outside the guests; this setting does not change counts gathered inside of a guest. Currently, this functionality is x86 only.

exclude_callchain_kernel (since Linux 3.7)
: Do not include kernel callchains.

exclude_callchain_user (since Linux 3.7)
: Do not include user callchains.

mmap2 (since Linux 3.16)
: Generate an extended executable mmap record that contains enough additional information to uniquely identify shared mappings. The *mmap* flag must also be set for this to work.

comm_exec (since Linux 3.16)
: This is purely a feature-detection flag, it does not change kernel behavior. If this flag can successfully be set, then, when *comm* is enabled, the **PERF_RECORD_MISC_COMM_EXEC** flag will be set in the *misc* field of a comm record header if the rename event being reported was caused by a call to *execve*(2). This allows tools to distinguish between the various types of process renaming.

use_clockid (since Linux 4.1)
: This allows selecting which internal Linux clock to use when generating timestamps via the *clockid* field. This can make it easier to correlate perf sample times with timestamps generated by other tools.

context_switch (since Linux 4.3)
: This enables the generation of **PERF_RECORD_SWITCH** records when a context switch occurs. It also enables the generation of **PERF_RECORD_SWITCH_CPU_WIDE** records when sampling in CPU-wide mode. This functionality is in addition to existing tracepoint and software events for measuring context switches. The advantage of this method is that it will give full information even with strict *perf_event_paranoid* settings.

write_backward (since Linux 4.6)
: This causes the ring buffer to be written from the end to the beginning. This is to support reading from overwritable ring buffer.

namespaces (since Linux 4.11)
: This enables the generation of **PERF_RECORD_NAMESPACES** records when a task enters a new namespace. Each namespace has a combination of device and inode numbers.

ksymbol (since Linux 5.0)
This enables the generation of **PERF_RECORD_KSYMBOL** records when new kernel symbols are registered or unregistered. This is analyzing dynamic kernel functions like eBPF.

bpf_event (since Linux 5.0)
This enables the generation of **PERF_RECORD_BPF_EVENT** records when an eBPF program is loaded or unloaded.

aux_output (since Linux 5.4)
This allows normal (non-AUX) events to generate data for AUX events if the hardware supports it.

cgroup (since Linux 5.7)
This enables the generation of **PERF_RECORD_CGROUP** records when a new cgroup is created (and activated).

text_poke (since Linux 5.8)
This enables the generation of **PERF_RECORD_TEXT_POKE** records when there's a change to the kernel text (i.e., self-modifying code).

build_id (since Linux 5.12)
This changes the contents in the **PERF_RECORD_MMAP2** to have a build-id instead of device and inode numbers.

inherit_thread (since Linux 5.13)
This disables the inheritance of the event to a child process. Only new threads in the same process (which is cloned with **CLONE_THREAD**) will inherit the event.

remove_on_exec (since Linux 5.13)
This closes the event when it starts a new process image by *execve*(2).

sigtrap (since Linux 5.13)
This enables synchronous signal delivery of **SIGTRAP** on event overflow.

wakeup_events
wakeup_watermark
This union sets how many samples (*wakeup_events*) or bytes (*wakeup_watermark*) happen before an overflow notification happens. Which one is used is selected by the *watermark* bit flag.

wakeup_events counts only **PERF_RECORD_SAMPLE** record types. To receive overflow notification for all **PERF_RECORD** types choose watermark and set *wakeup_watermark* to 1.

Prior to Linux 3.0, setting *wakeup_events* to 0 resulted in no overflow notifications; more recent kernels treat 0 the same as 1.

bp_type (since Linux 2.6.33)
This chooses the breakpoint type. It is one of:

HW_BREAKPOINT_EMPTY
No breakpoint.

HW_BREAKPOINT_R
Count when we read the memory location.

HW_BREAKPOINT_W
Count when we write the memory location.

HW_BREAKPOINT_RW
Count when we read or write the memory location.

HW_BREAKPOINT_X
Count when we execute code at the memory location.

The values can be combined via a bitwise or, but the combination of **HW_BREAKPOINT_R** or **HW_BREAKPOINT_W** with **HW_BREAKPOINT_X** is not allowed.

bp_addr (since Linux 2.6.33)
This is the address of the breakpoint. For execution breakpoints, this is the memory address of the instruction of interest; for read and write breakpoints, it is the memory address of the memory location of interest.

config1 (since Linux 2.6.39)
config1 is used for setting events that need an extra register or otherwise do not fit in the regular config field. Raw OFFCORE_EVENTS on Nehalem/Westmere/SandyBridge use this field on Linux 3.3 and later kernels.

bp_len (since Linux 2.6.33)
bp_len is the length of the breakpoint being measured if *type* is **PERF_TYPE_BREAKPOINT**. Options are **HW_BREAKPOINT_LEN_1**, **HW_BREAKPOINT_LEN_2**, **HW_BREAKPOINT_LEN_4**, and **HW_BREAKPOINT_LEN_8**. For an execution breakpoint, set this to *sizeof(long)*.

config2 (since Linux 2.6.39)
config2 is a further extension of the *config1* field.

branch_sample_type (since Linux 3.4)
If **PERF_SAMPLE_BRANCH_STACK** is enabled, then this specifies what branches to include in the branch record.

The first part of the value is the privilege level, which is a combination of one of the values listed below. If the user does not set privilege level explicitly, the kernel will use the event's privilege level. Event and branch privilege levels do not have to match.

PERF_SAMPLE_BRANCH_USER
Branch target is in user space.

PERF_SAMPLE_BRANCH_KERNEL
Branch target is in kernel space.

PERF_SAMPLE_BRANCH_HV
Branch target is in hypervisor.

PERF_SAMPLE_BRANCH_PLM_ALL
A convenience value that is the three preceding values ORed together.

In addition to the privilege value, at least one or more of the following bits must be set.

PERF_SAMPLE_BRANCH_ANY
Any branch type.

PERF_SAMPLE_BRANCH_ANY_CALL
Any call branch (includes direct calls, indirect calls, and far jumps).

PERF_SAMPLE_BRANCH_IND_CALL
Indirect calls.

PERF_SAMPLE_BRANCH_CALL (since Linux 4.4)
Direct calls.

PERF_SAMPLE_BRANCH_ANY_RETURN
Any return branch.

PERF_SAMPLE_BRANCH_IND_JUMP (since Linux 4.2)
Indirect jumps.

PERF_SAMPLE_BRANCH_COND (since Linux 3.16)
Conditional branches.

PERF_SAMPLE_BRANCH_ABORT_TX (since Linux 3.11)
Transactional memory aborts.

PERF_SAMPLE_BRANCH_IN_TX (since Linux 3.11)
Branch in transactional memory transaction.

PERF_SAMPLE_BRANCH_NO_TX (since Linux 3.11)
Branch not in transactional memory transaction. **PERF_SAMPLE_BRANCH_CALL_STACK** (since Linux 4.1) Branch is part of a hardware-

generated call stack. This requires hardware support, currently only found on Intel x86 Haswell or newer.

sample_regs_user (since Linux 3.7)
This bit mask defines the set of user CPU registers to dump on samples. The layout of the register mask is architecture-specific and is described in the kernel header file *arch/ARCH/include/uapi/asm/perf_regs.h*.

sample_stack_user (since Linux 3.7)
This defines the size of the user stack to dump if **PERF_SAMPLE_STACK_USER** is specified.

clockid (since Linux 4.1)
If *use_clockid* is set, then this field selects which internal Linux timer to use for timestamps. The available timers are defined in *linux/time.h*, with **CLOCK_MONOTONIC**, **CLOCK_MONOTONIC_RAW**, **CLOCK_REALTIME**, **CLOCK_BOOTTIME**, and **CLOCK_TAI** currently supported.

aux_watermark (since Linux 4.1)
This specifies how much data is required to trigger a **PERF_RECORD_AUX** sample.

sample_max_stack (since Linux 4.8)
When *sample_type* includes **PERF_SAMPLE_CALLCHAIN**, this field specifies how many stack frames to report when generating the callchain.

aux_sample_size (since Linux 5.5)
When **PERF_SAMPLE_AUX** flag is set, specify the desired size of AUX data. Note that it can get smaller data than the specified size.

sig_data (since Linux 5.13)
This data will be copied to user's signal handler (through *si_perf* in the *siginfo_t*) to disambiguate which event triggered the signal.

Reading results

Once a **perf_event_open**() file descriptor has been opened, the values of the events can be read from the file descriptor. The values that are there are specified by the *read_format* field in the *attr* structure at open time.

If you attempt to read into a buffer that is not big enough to hold the data, the error **ENOSPC** results.

Here is the layout of the data returned by a read:

- If **PERF_FORMAT_GROUP** was specified to allow reading all events in a group at once:

```
struct read_format {
    u64 nr;            /* The number of events */
    u64 time_enabled;  /* if PERF_FORMAT_TOTAL_TIME_ENABLED */
    u64 time_running;  /* if PERF_FORMAT_TOTAL_TIME_RUNNING */
    struct {
        u64 value;     /* The value of the event */
        u64 id;        /* if PERF_FORMAT_ID */
        u64 lost;      /* if PERF_FORMAT_LOST */
    } values[nr];
};
```

- If **PERF_FORMAT_GROUP** was *not* specified:

```
struct read_format {
    u64 value;         /* The value of the event */
    u64 time_enabled;  /* if PERF_FORMAT_TOTAL_TIME_ENABLED */
    u64 time_running;  /* if PERF_FORMAT_TOTAL_TIME_RUNNING */
    u64 id;            /* if PERF_FORMAT_ID */
    u64 lost;          /* if PERF_FORMAT_LOST */
};
```

The values read are as follows:

nr The number of events in this file descriptor. Available only if **PERF_FORMAT_GROUP** was specified.

time_enabled
time_running
Total time the event was enabled and running. Normally these values are the same. Multiplexing happens if the number of events is more than the number of available PMU counter slots. In that case the events run only part of the time and the *time_enabled* and *time running* values can be used to scale an estimated value for the count.

value An unsigned 64-bit value containing the counter result.

id A globally unique value for this particular event; only present if **PERF_FORMAT_ID** was specified in *read_format*.

lost The number of lost samples of this event; only present if **PERF_FORMAT_LOST** was specified in *read_format*.

MMAP layout

When using **perf_event_open**() in sampled mode, asynchronous events (like counter overflow or **PROT_EXEC** mmap tracking) are logged into a ring-buffer. This ring-buffer is created and accessed through *mmap*(2).

The mmap size should be 1+2^n pages, where the first page is a metadata page (*struct perf_event_mmap_page*) that contains various bits of information such as where the ring-buffer head is.

Before Linux 2.6.39, there is a bug that means you must allocate an mmap ring buffer when sampling even if you do not plan to access it.

The structure of the first metadata mmap page is as follows:

```
struct perf_event_mmap_page {
    __u32 version;        /* version number of this structure */
    __u32 compat_version; /* lowest version this is compat with */
    __u32 lock;           /* seqlock for synchronization */
    __u32 index;          /* hardware counter identifier */
    __s64 offset;         /* add to hardware counter value */
    __u64 time_enabled;   /* time event active */
    __u64 time_running;   /* time event on CPU */
    union {
        __u64   capabilities;
        struct {
            __u64 cap_usr_time / cap_usr_rdpmc / cap_bit0 : 1,
                  cap_bit0_is_deprecated : 1,
                  cap_user_rdpmc         : 1,
                  cap_user_time          : 1,
                  cap_user_time_zero     : 1,
        };
    };
    __u16 pmc_width;
    __u16 time_shift;
    __u32 time_mult;
    __u64 time_offset;
    __u64 __reserved[120];   /* Pad to 1 k */
    __u64 data_head;         /* head in the data section */
    __u64 data_tail;         /* user-space written tail */
    __u64 data_offset;       /* where the buffer starts */
    __u64 data_size;         /* data buffer size */
    __u64 aux_head;
    __u64 aux_tail;
    __u64 aux_offset;
    __u64 aux_size;

}
```

The following list describes the fields in the *perf_event_mmap_page* structure in more detail:

version Version number of this structure.

compat_version
The lowest version this is compatible with.

lock A seqlock for synchronization.

index A unique hardware counter identifier.

offset When using rdpmc for reads this offset value must be added to the one returned by rdpmc to get the current total event count.

time_enabled
Time the event was active.

time_running
Time the event was running.

cap_usr_time / *cap_usr_rdpmc* / *cap_bit0* (since Linux 3.4)
There was a bug in the definition of *cap_usr_time* and *cap_usr_rdpmc* from Linux 3.4 until Linux 3.11. Both bits were defined to point to the same location, so it was impossible to know if *cap_usr_time* or *cap_usr_rdpmc* were actually set.

Starting with Linux 3.12, these are renamed to *cap_bit0* and you should use the *cap_user_time* and *cap_user_rdpmc* fields instead.

cap_bit0_is_deprecated (since Linux 3.12)
If set, this bit indicates that the kernel supports the properly separated *cap_user_time* and *cap_user_rdpmc* bits.

If not-set, it indicates an older kernel where *cap_usr_time* and *cap_usr_rdpmc* map to the same bit and thus both features should be used with caution.

cap_user_rdpmc (since Linux 3.12)
If the hardware supports user-space read of performance counters without syscall (this is the "rdpmc" instruction on x86), then the following code can be used to do a read:

```
u32 seq, time_mult, time_shift, idx, width;
u64 count, enabled, running;
u64 cyc, time_offset;

do {
    seq = pc->lock;
    barrier();
    enabled = pc->time_enabled;
    running = pc->time_running;

    if (pc->cap_usr_time && enabled != running) {
        cyc = rdtsc();
        time_offset = pc->time_offset;
        time_mult   = pc->time_mult;
        time_shift  = pc->time_shift;
    }

    idx = pc->index;
    count = pc->offset;

    if (pc->cap_usr_rdpmc && idx) {
        width = pc->pmc_width;
        count += rdpmc(idx - 1);
    }

    barrier();
} while (pc->lock != seq);
```

cap_user_time (since Linux 3.12)
This bit indicates the hardware has a constant, nonstop timestamp counter (TSC on x86).

cap_user_time_zero (since Linux 3.12)
Indicates the presence of *time_zero* which allows mapping timestamp values to the hardware clock.

pmc_width
If *cap_usr_rdpmc*, this field provides the bit-width of the value read using the rdpmc or equivalent instruction. This can be used to sign extend the result like:

```
pmc <<= 64 - pmc_width;
pmc >>= 64 - pmc_width; // signed shift right
count += pmc;
```

time_shift
time_mult
time_offset

If *cap_usr_time*, these fields can be used to compute the time delta since *time_enabled* (in nanoseconds) using rdtsc or similar.

```
u64 quot, rem;
u64 delta;

quot  = cyc >> time_shift;
rem   = cyc & (((u64)1 << time_shift) - 1);
delta = time_offset + quot * time_mult +
        ((rem * time_mult) >> time_shift);
```

Where *time_offset*, *time_mult*, *time_shift*, and *cyc* are read in the seqcount loop described above. This delta can then be added to enabled and possible running (if idx), improving the scaling:

```
enabled += delta;
if (idx)
    running += delta;
quot  = count / running;
rem   = count % running;
count = quot * enabled + (rem * enabled) / running;
```

time_zero (since Linux 3.12)

If *cap_usr_time_zero* is set, then the hardware clock (the TSC timestamp counter on x86) can be calculated from the *time_zero*, *time_mult*, and *time_shift* values:

```
time = timestamp - time_zero;
quot = time / time_mult;
rem  = time % time_mult;
cyc  = (quot << time_shift) + (rem << time_shift) / time_mult;
```

And vice versa:

```
quot = cyc >> time_shift;
rem  = cyc & (((u64)1 << time_shift) - 1);
timestamp = time_zero + quot * time_mult +
            ((rem * time_mult) >> time_shift);
```

data_head
This points to the head of the data section. The value continuously increases, it does not wrap. The value needs to be manually wrapped by the size of the mmap buffer before accessing the samples.

On SMP-capable platforms, after reading the *data_head* value, user space should issue an rmb().

data_tail
: When the mapping is **PROT_WRITE**, the *data_tail* value should be written by user space to reflect the last read data. In this case, the kernel will not overwrite unread data.

data_offset (since Linux 4.1)
: Contains the offset of the location in the mmap buffer where perf sample data begins.

data_size (since Linux 4.1)
: Contains the size of the perf sample region within the mmap buffer.

aux_head
aux_tail
aux_offset
aux_size (since Linux 4.1)
: The AUX region allows *mmap*(2)-ing a separate sample buffer for high-bandwidth data streams (separate from the main perf sample buffer). An example of a high-bandwidth stream is instruction tracing support, as is found in newer Intel processors.

  To set up an AUX area, first *aux_offset* needs to be set with an offset greater than *data_offset+data_size* and *aux_size* needs to be set to the desired buffer size. The desired offset and size must be page aligned, and the size must be a power of two. These values are then passed to mmap in order to map the AUX buffer. Pages in the AUX buffer are included as part of the **RLIMIT_MEMLOCK** resource limit (see *setrlimit*(2)), and also as part of the *perf_event_mlock_kb* allowance.

  By default, the AUX buffer will be truncated if it will not fit in the available space in the ring buffer. If the AUX buffer is mapped as a read only buffer, then it will operate in ring buffer mode where old data will be overwritten by new. In overwrite mode, it might not be possible to infer where the new data began, and it is the consumer's job to disable measurement while reading to avoid possible data races.

  The *aux_head* and *aux_tail* ring buffer pointers have the same behavior and ordering rules as the previous described *data_head* and *data_tail*.

The following 2^n ring-buffer pages have the layout described below.

If *perf_event_attr.sample_id_all* is set, then all event types will have the sample_type selected fields related to where/when (identity) an event took place (TID, TIME, ID, CPU, STREAM_ID) described in **PERF_RECORD_SAMPLE** below, it will be stashed just after the *perf_event_header* and the fields already present for the existing fields, that is, at the end of the payload. This allows a newer perf.data file to be supported by older perf tools, with the new optional fields being ignored.

The mmap values start with a header:

```
struct perf_event_header {
    __u32   type;
    __u16   misc;
    __u16   size;
};
```

Below, we describe the *perf_event_header* fields in more detail. For ease of reading, the fields with shorter descriptions are presented first.

size This indicates the size of the record.

misc The *misc* field contains additional information about the sample.

The CPU mode can be determined from this value by masking with **PERF_RECORD_MISC_CPUMODE_MASK** and looking for one of the following (note these are not bit masks, only one can be set at a time):

PERF_RECORD_MISC_CPUMODE_UNKNOWN
: Unknown CPU mode.

PERF_RECORD_MISC_KERNEL
: Sample happened in the kernel.

PERF_RECORD_MISC_USER
Sample happened in user code.

PERF_RECORD_MISC_HYPERVISOR
Sample happened in the hypervisor.

PERF_RECORD_MISC_GUEST_KERNEL (since Linux 2.6.35)
Sample happened in the guest kernel.

PERF_RECORD_MISC_GUEST_USER (since Linux 2.6.35)
Sample happened in guest user code.

Since the following three statuses are generated by different record types, they alias to the same bit:

PERF_RECORD_MISC_MMAP_DATA (since Linux 3.10)
This is set when the mapping is not executable; otherwise the mapping is executable.

PERF_RECORD_MISC_COMM_EXEC (since Linux 3.16)
This is set for a **PERF_RECORD_COMM** record on kernels more recent than Linux 3.16 if a process name change was caused by an *execve*(2) system call.

PERF_RECORD_MISC_SWITCH_OUT (since Linux 4.3)
When a **PERF_RECORD_SWITCH** or **PERF_RECORD_SWITCH_CPU_WIDE** record is generated, this bit indicates that the context switch is away from the current process (instead of into the current process).

In addition, the following bits can be set:

PERF_RECORD_MISC_EXACT_IP
This indicates that the content of **PERF_SAMPLE_IP** points to the actual instruction that triggered the event. See also *perf_event_attr.precise_ip*.

PERF_RECORD_MISC_SWITCH_OUT_PREEMPT (since Linux 4.17)
When a **PERF_RECORD_SWITCH** or **PERF_RECORD_SWITCH_CPU_WIDE** record is generated, this indicates the context switch was a preemption.

PERF_RECORD_MISC_MMAP_BUILD_ID (since Linux 5.12)
This indicates that the content of **PERF_SAMPLE_MMAP2** contains build-ID data instead of device major and minor numbers as well as the inode number.

PERF_RECORD_MISC_EXT_RESERVED (since Linux 2.6.35)
This indicates there is extended data available (currently not used).

PERF_RECORD_MISC_PROC_MAP_PARSE_TIMEOUT
This bit is not set by the kernel. It is reserved for the user-space perf utility to indicate that */proc/*pid*/maps* parsing was taking too long and was stopped, and thus the mmap records may be truncated.

type The *type* value is one of the below. The values in the corresponding record (that follows the header) depend on the *type* selected as shown.

PERF_RECORD_MMAP
The MMAP events record the **PROT_EXEC** mappings so that we can correlate user-space IPs to code. They have the following structure:

```
struct {
    struct perf_event_header header;
    u32    pid, tid;
    u64    addr;
    u64    len;
    u64    pgoff;
    char   filename[];
};
```

pid is the process ID.

tid is the thread ID.

addr is the address of the allocated memory. *len* is the length of the allocated memory. *pgoff* is the page offset of the allocated memory. *filename* is a string describing the backing of the allocated memory.

PERF_RECORD_LOST

This record indicates when events are lost.

```
struct {
    struct perf_event_header header;
    u64    id;
    u64    lost;
    struct sample_id sample_id;
};
```

id is the unique event ID for the samples that were lost.

lost is the number of events that were lost.

PERF_RECORD_COMM

This record indicates a change in the process name.

```
struct {
    struct perf_event_header header;
    u32    pid;
    u32    tid;
    char   comm[];
    struct sample_id sample_id;
};
```

pid is the process ID.

tid is the thread ID.

comm is a string containing the new name of the process.

PERF_RECORD_EXIT

This record indicates a process exit event.

```
struct {
    struct perf_event_header header;
    u32    pid, ppid;
    u32    tid, ptid;
    u64    time;
    struct sample_id sample_id;
};
```

PERF_RECORD_THROTTLE
PERF_RECORD_UNTHROTTLE

This record indicates a throttle/unthrottle event.

```
struct {
    struct perf_event_header header;
    u64    time;
    u64    id;
    u64    stream_id;
    struct sample_id sample_id;
};
```

PERF_RECORD_FORK

This record indicates a fork event.

```
struct {
    struct perf_event_header header;
    u32    pid, ppid;
    u32    tid, ptid;
```

```
        u64    time;
        struct sample_id sample_id;
    };
```

PERF_RECORD_READ
This record indicates a read event.

```
    struct {
        struct perf_event_header header;
        u32    pid, tid;
        struct read_format values;
        struct sample_id sample_id;
    };
```

PERF_RECORD_SAMPLE
This record indicates a sample.

```
    struct {
        struct perf_event_header header;
        u64    sample_id;   /* if PERF_SAMPLE_IDENTIFIER */
        u64    ip;          /* if PERF_SAMPLE_IP */
        u32    pid, tid;    /* if PERF_SAMPLE_TID */
        u64    time;        /* if PERF_SAMPLE_TIME */
        u64    addr;        /* if PERF_SAMPLE_ADDR */
        u64    id;          /* if PERF_SAMPLE_ID */
        u64    stream_id;   /* if PERF_SAMPLE_STREAM_ID */
        u32    cpu, res;    /* if PERF_SAMPLE_CPU */
        u64    period;      /* if PERF_SAMPLE_PERIOD */
        struct read_format v;
                            /* if PERF_SAMPLE_READ */
        u64    nr;          /* if PERF_SAMPLE_CALLCHAIN */
        u64    ips[nr];     /* if PERF_SAMPLE_CALLCHAIN */
        u32    size;        /* if PERF_SAMPLE_RAW */
        char   data[size];  /* if PERF_SAMPLE_RAW */
        u64    bnr;         /* if PERF_SAMPLE_BRANCH_STACK */
        struct perf_branch_entry lbr[bnr];
                            /* if PERF_SAMPLE_BRANCH_STACK */
        u64    abi;         /* if PERF_SAMPLE_REGS_USER */
        u64    regs[weight(mask)];
                            /* if PERF_SAMPLE_REGS_USER */
        u64    size;        /* if PERF_SAMPLE_STACK_USER */
        char   data[size];  /* if PERF_SAMPLE_STACK_USER */
        u64    dyn_size;    /* if PERF_SAMPLE_STACK_USER &&
                               size != 0 */
        union perf_sample_weight weight;
                            /* if PERF_SAMPLE_WEIGHT */
                            /* || PERF_SAMPLE_WEIGHT_STRUCT */
        u64    data_src;    /* if PERF_SAMPLE_DATA_SRC */
        u64    transaction; /* if PERF_SAMPLE_TRANSACTION */
        u64    abi;         /* if PERF_SAMPLE_REGS_INTR */
        u64    regs[weight(mask)];
                            /* if PERF_SAMPLE_REGS_INTR */
        u64    phys_addr;   /* if PERF_SAMPLE_PHYS_ADDR */
        u64    cgroup;      /* if PERF_SAMPLE_CGROUP */
        u64    data_page_size;
                           /* if PERF_SAMPLE_DATA_PAGE_SIZE */
        u64    code_page_size;
                          /* if PERF_SAMPLE_CODE_PAGE_SIZE */
        u64    size;        /* if PERF_SAMPLE_AUX */
        char   data[size];  /* if PERF_SAMPLE_AUX */
    };
```

sample_id
: If **PERF_SAMPLE_IDENTIFIER** is enabled, a 64-bit unique ID is included. This is a duplication of the **PERF_SAMPLE_ID** *id* value, but included at the beginning of the sample so parsers can easily obtain the value.

ip
: If **PERF_SAMPLE_IP** is enabled, then a 64-bit instruction pointer value is included.

pid
tid
: If **PERF_SAMPLE_TID** is enabled, then a 32-bit process ID and 32-bit thread ID are included.

time
: If **PERF_SAMPLE_TIME** is enabled, then a 64-bit timestamp is included. This is obtained via local_clock() which is a hardware timestamp if available and the jiffies value if not.

addr
: If **PERF_SAMPLE_ADDR** is enabled, then a 64-bit address is included. This is usually the address of a tracepoint, breakpoint, or software event; otherwise the value is 0.

id
: If **PERF_SAMPLE_ID** is enabled, a 64-bit unique ID is included. If the event is a member of an event group, the group leader ID is returned. This ID is the same as the one returned by **PERF_FORMAT_ID**.

stream_id
: If **PERF_SAMPLE_STREAM_ID** is enabled, a 64-bit unique ID is included. Unlike **PERF_SAMPLE_ID** the actual ID is returned, not the group leader. This ID is the same as the one returned by **PERF_FORMAT_ID**.

cpu
res
: If **PERF_SAMPLE_CPU** is enabled, this is a 32-bit value indicating which CPU was being used, in addition to a reserved (unused) 32-bit value.

period
: If **PERF_SAMPLE_PERIOD** is enabled, a 64-bit value indicating the current sampling period is written.

v
: If **PERF_SAMPLE_READ** is enabled, a structure of type read_format is included which has values for all events in the event group. The values included depend on the *read_format* value used at **perf_event_open**() time.

nr
ips[nr]
: If **PERF_SAMPLE_CALLCHAIN** is enabled, then a 64-bit number is included which indicates how many following 64-bit instruction pointers will follow. This is the current callchain.

size
data[size]
: If **PERF_SAMPLE_RAW** is enabled, then a 32-bit value indicating size is included followed by an array of 8-bit values of length size. The values are padded with 0 to have 64-bit alignment.

  This RAW record data is opaque with respect to the ABI. The ABI doesn't make any promises with respect to the stability of its content, it may vary depending on event, hardware, and kernel version.

bnr
lbr[bnr]
: If **PERF_SAMPLE_BRANCH_STACK** is enabled, then a 64-bit value indicating the number of records is included, followed by *bnr* *perf_branch_entry* structures which each include the fields:

from This indicates the source instruction (may not be a branch).

to The branch target.

mispred
The branch target was mispredicted.

predicted
The branch target was predicted.

in_tx (since Linux 3.11)
The branch was in a transactional memory transaction.

abort (since Linux 3.11)
The branch was in an aborted transactional memory transaction.

cycles (since Linux 4.3)
This reports the number of cycles elapsed since the previous branch stack update.

The entries are from most to least recent, so the first entry has the most recent branch.

Support for *mispred*, *predicted*, and *cycles* is optional; if not supported, those values will be 0.

The type of branches recorded is specified by the *branch_sample_type* field.

abi
regs[weight(mask)]
If **PERF_SAMPLE_REGS_USER** is enabled, then the user CPU registers are recorded.

The *abi* field is one of **PERF_SAMPLE_REGS_ABI_NONE**, **PERF_SAMPLE_REGS_ABI_32**, or **PERF_SAMPLE_REGS_ABI_64**.

The *regs* field is an array of the CPU registers that were specified by the *sample_regs_user* attr field. The number of values is the number of bits set in the *sample_regs_user* bit mask.

size
data[size]
dyn_size
If **PERF_SAMPLE_STACK_USER** is enabled, then the user stack is recorded. This can be used to generate stack backtraces. *size* is the size requested by the user in *sample_stack_user* or else the maximum record size. *data* is the stack data (a raw dump of the memory pointed to by the stack pointer at the time of sampling). *dyn_size* is the amount of data actually dumped (can be less than *size*). Note that *dyn_size* is omitted if *size* is 0.

weight
If **PERF_SAMPLE_WEIGHT** or **PERF_SAMPLE_WEIGHT_STRUCT** is enabled, then a 64-bit value provided by the hardware is recorded that indicates how costly the event was. This allows expensive events to stand out more clearly in profiles.

data_src
If **PERF_SAMPLE_DATA_SRC** is enabled, then a 64-bit value is recorded that is made up of the following fields:

mem_op
Type of opcode, a bitwise combination of:

PERF_MEM_OP_NA Not available
PERF_MEM_OP_LOAD Load instruction
PERF_MEM_OP_STORE Store instruction
PERF_MEM_OP_PFETCH
Prefetch

PERF_MEM_OP_EXEC Executable code

mem_lvl
Memory hierarchy level hit or miss, a bitwise combination of the following, shifted left by **PERF_MEM_LVL_SHIFT**:

PERF_MEM_LVL_NA Not available
PERF_MEM_LVL_HIT Hit
PERF_MEM_LVL_MISS Miss
PERF_MEM_LVL_L1 Level 1 cache
PERF_MEM_LVL_LFB Line fill buffer
PERF_MEM_LVL_L2 Level 2 cache
PERF_MEM_LVL_L3 Level 3 cache
PERF_MEM_LVL_LOC_RAM
Local DRAM
PERF_MEM_LVL_REM_RAM1
Remote DRAM 1 hop
PERF_MEM_LVL_REM_RAM2
Remote DRAM 2 hops
PERF_MEM_LVL_REM_CCE1
Remote cache 1 hop
PERF_MEM_LVL_REM_CCE2
Remote cache 2 hops
PERF_MEM_LVL_IO I/O memory
PERF_MEM_LVL_UNC Uncached memory

mem_snoop
Snoop mode, a bitwise combination of the following, shifted left by **PERF_MEM_SNOOP_SHIFT**:

PERF_MEM_SNOOP_NA
Not available
PERF_MEM_SNOOP_NONE
No snoop
PERF_MEM_SNOOP_HIT
Snoop hit
PERF_MEM_SNOOP_MISS
Snoop miss
PERF_MEM_SNOOP_HITM
Snoop hit modified

mem_lock
Lock instruction, a bitwise combination of the following, shifted left by **PERF_MEM_LOCK_SHIFT**:

PERF_MEM_LOCK_NA Not available
PERF_MEM_LOCK_LOCKED
Locked transaction

mem_dtlb
TLB access hit or miss, a bitwise combination of the following, shifted left by **PERF_MEM_TLB_SHIFT**:

PERF_MEM_TLB_NA Not available
PERF_MEM_TLB_HIT Hit
PERF_MEM_TLB_MISS Miss
PERF_MEM_TLB_L1 Level 1 TLB
PERF_MEM_TLB_L2 Level 2 TLB
PERF_MEM_TLB_WK Hardware walker
PERF_MEM_TLB_OS OS fault handler

transaction
If the **PERF_SAMPLE_TRANSACTION** flag is set, then a 64-bit field is recorded describing the sources of any transactional memory aborts.

The field is a bitwise combination of the following values:

PERF_TXN_ELISION
Abort from an elision type transaction (Intel-CPU-specific).

PERF_TXN_TRANSACTION
Abort from a generic transaction.

PERF_TXN_SYNC
Synchronous abort (related to the reported instruction).

PERF_TXN_ASYNC
Asynchronous abort (not related to the reported instruction).

PERF_TXN_RETRY
Retryable abort (retrying the transaction may have succeeded).

PERF_TXN_CONFLICT
Abort due to memory conflicts with other threads.

PERF_TXN_CAPACITY_WRITE
Abort due to write capacity overflow.

PERF_TXN_CAPACITY_READ
Abort due to read capacity overflow.

In addition, a user-specified abort code can be obtained from the high 32 bits of the field by shifting right by **PERF_TXN_ABORT_SHIFT** and masking with the value **PERF_TXN_ABORT_MASK**.

abi
regs[weight(mask)]
If **PERF_SAMPLE_REGS_INTR** is enabled, then the user CPU registers are recorded.

The *abi* field is one of **PERF_SAMPLE_REGS_ABI_NONE**, **PERF_SAMPLE_REGS_ABI_32**, or **PERF_SAMPLE_REGS_ABI_64**.

The *regs* field is an array of the CPU registers that were specified by the *sample_regs_intr* attr field. The number of values is the number of bits set in the *sample_regs_intr* bit mask.

phys_addr
If the **PERF_SAMPLE_PHYS_ADDR** flag is set, then the 64-bit physical address is recorded.

cgroup
If the **PERF_SAMPLE_CGROUP** flag is set, then the 64-bit cgroup ID (for the perf_event subsystem) is recorded. To get the pathname of the cgroup, the ID should match to one in a **PERF_RECORD_CGROUP**.

data_page_size
If the **PERF_SAMPLE_DATA_PAGE_SIZE** flag is set, then the 64-bit page size value of the **data** address is recorded.

code_page_size
If the **PERF_SAMPLE_CODE_PAGE_SIZE** flag is set, then the 64-bit page size value of the **ip** address is recorded.

size
data[*size*]
If **PERF_SAMPLE_AUX** is enabled, a snapshot of the aux buffer is recorded.

PERF_RECORD_MMAP2
This record includes extended information on *mmap*(2) calls returning executable mappings. The format is similar to that of the **PERF_RECORD_MMAP** record, but includes extra values that allow uniquely identifying shared mappings. Depending on the **PERF_RECORD_MISC_MMAP_BUILD_ID** bit in the header, the extra values have different layout and meanings.

```
struct {
    struct perf_event_header header;
    u32    pid;
    u32    tid;
    u64    addr;
    u64    len;
    u64    pgoff;
    union {
        struct {
            u32    maj;
            u32    min;
            u64    ino;
            u64    ino_generation;
        };
        struct {   /* if PERF_RECORD_MISC_MMAP_BUILD_ID */
            u8     build_id_size;
            u8     __reserved_1;
            u16    __reserved_2;
            u8     build_id[20];
        };
    };
    u32    prot;
    u32    flags;
    char   filename[];
    struct sample_id sample_id;
};
```

pid is the process ID.

tid is the thread ID.

addr is the address of the allocated memory.

len is the length of the allocated memory.

pgoff is the page offset of the allocated memory.

maj is the major ID of the underlying device.

min is the minor ID of the underlying device.

ino is the inode number.

ino_generation
is the inode generation.

build_id_size
is the actual size of *build_id* field (up to 20).

build_id
is a raw data to identify a binary.

prot is the protection information.

flags is the flags information.

filename
is a string describing the backing of the allocated memory.

PERF_RECORD_AUX (since Linux 4.1)
This record reports that new data is available in the separate AUX buffer region.

```
struct {
    struct perf_event_header header;
    u64    aux_offset;
    u64    aux_size;
    u64    flags;
    struct sample_id sample_id;
```

```
};
```

aux_offset
: offset in the AUX mmap region where the new data begins.

aux_size
: size of the data made available.

flags describes the AUX update.

PERF_AUX_FLAG_TRUNCATED
: if set, then the data returned was truncated to fit the available buffer size.

PERF_AUX_FLAG_OVERWRITE
: if set, then the data returned has overwritten previous data.

PERF_RECORD_ITRACE_START (since Linux 4.1)
: This record indicates which process has initiated an instruction trace event, allowing tools to properly correlate the instruction addresses in the AUX buffer with the proper executable.

```
struct {
    struct perf_event_header header;
    u32    pid;
    u32    tid;
};
```

pid process ID of the thread starting an instruction trace.

tid thread ID of the thread starting an instruction trace.

PERF_RECORD_LOST_SAMPLES (since Linux 4.2)
: When using hardware sampling (such as Intel PEBS) this record indicates some number of samples that may have been lost.

```
struct {
    struct perf_event_header header;
    u64    lost;
    struct sample_id sample_id;
};
```

lost the number of potentially lost samples.

PERF_RECORD_SWITCH (since Linux 4.3)
: This record indicates a context switch has happened. The **PERF_RECORD_MISC_SWITCH_OUT** bit in the *misc* field indicates whether it was a context switch into or away from the current process.

```
struct {
    struct perf_event_header header;
    struct sample_id sample_id;
};
```

PERF_RECORD_SWITCH_CPU_WIDE (since Linux 4.3)
: As with **PERF_RECORD_SWITCH** this record indicates a context switch has happened, but it only occurs when sampling in CPU-wide mode and provides additional information on the process being switched to/from. The **PERF_RECORD_MISC_SWITCH_OUT** bit in the *misc* field indicates whether it was a context switch into or away from the current process.

```
struct {
    struct perf_event_header header;
    u32 next_prev_pid;
    u32 next_prev_tid;
    struct sample_id sample_id;
};
```

next_prev_pid
The process ID of the previous (if switching in) or next (if switching out) process on the CPU.

next_prev_tid
The thread ID of the previous (if switching in) or next (if switching out) thread on the CPU.

PERF_RECORD_NAMESPACES (since Linux 4.11)
This record includes various namespace information of a process.

```
struct {
    struct perf_event_header header;
    u32    pid;
    u32    tid;
    u64    nr_namespaces;
    struct { u64 dev, inode } [nr_namespaces];
    struct sample_id sample_id;
};
```

pid is the process ID

tid is the thread ID

nr_namespace
is the number of namespaces in this record

Each namespace has *dev* and *inode* fields and is recorded in the fixed position like below:

NET_NS_INDEX=**0**
Network namespace

UTS_NS_INDEX=**1**
UTS namespace

IPC_NS_INDEX=**2**
IPC namespace

PID_NS_INDEX=**3**
PID namespace

USER_NS_INDEX=**4**
User namespace

MNT_NS_INDEX=**5**
Mount namespace

CGROUP_NS_INDEX=**6**
Cgroup namespace

PERF_RECORD_KSYMBOL (since Linux 5.0)
This record indicates kernel symbol register/unregister events.

```
struct {
    struct perf_event_header header;
    u64    addr;
    u32    len;
    u16    ksym_type;
    u16    flags;
    char   name[];
    struct sample_id sample_id;
};
```

addr is the address of the kernel symbol.

len is the length of the kernel symbol.

ksym_type
is the type of the kernel symbol. Currently the following types are available:

PERF_RECORD_KSYMBOL_TYPE_BPF
The kernel symbol is a BPF function.

flags If the **PERF_RECORD_KSYMBOL_FLAGS_UNREGISTER** is set, then this event is for unregistering the kernel symbol.

PERF_RECORD_BPF_EVENT (since Linux 5.0)
This record indicates BPF program is loaded or unloaded.

```
struct {
    struct perf_event_header header;
    u16 type;
    u16 flags;
    u32 id;
    u8 tag[BPF_TAG_SIZE];
    struct sample_id sample_id;
};
```

type is one of the following values:

PERF_BPF_EVENT_PROG_LOAD
A BPF program is loaded

PERF_BPF_EVENT_PROG_UNLOAD
A BPF program is unloaded

id is the ID of the BPF program.

tag is the tag of the BPF program. Currently, **BPF_TAG_SIZE** is defined as 8.

PERF_RECORD_CGROUP (since Linux 5.7)
This record indicates a new cgroup is created and activated.

```
struct {
    struct perf_event_header header;
    u64    id;
    char   path[];
    struct sample_id sample_id;
};
```

id is the cgroup identifier. This can be also retrieved by *name_to_handle_at*(2) on the cgroup path (as a file handle).

path is the path of the cgroup from the root.

PERF_RECORD_TEXT_POKE (since Linux 5.8)
This record indicates a change in the kernel text. This includes addition and removal of the text and the corresponding length is zero in this case.

```
struct {
    struct perf_event_header header;
    u64    addr;
    u16    old_len;
    u16    new_len;
    u8     bytes[];
    struct sample_id sample_id;
};
```

addr is the address of the change

old_len is the old length

new_len
is the new length

bytes contains old bytes immediately followed by new bytes.

Overflow handling
Events can be set to notify when a threshold is crossed, indicating an overflow. Overflow conditions can be captured by monitoring the event file descriptor with *poll*(2), *select*(2), or *epoll*(7).

Alternatively, the overflow events can be captured via sa signal handler, by enabling I/O signaling on the file descriptor; see the discussion of the **F_SETOWN** and **F_SETSIG** operations in *fcntl*(2).

Overflows are generated only by sampling events (*sample_period* must have a nonzero value).

There are two ways to generate overflow notifications.

The first is to set a *wakeup_events* or *wakeup_watermark* value that will trigger if a certain number of samples or bytes have been written to the mmap ring buffer. In this case, **POLL_IN** is indicated.

The other way is by use of the **PERF_EVENT_IOC_REFRESH** ioctl. This ioctl adds to a counter that decrements each time the event overflows. When nonzero, **POLL_IN** is indicated, but once the counter reaches 0 **POLL_HUP** is indicated and the underlying event is disabled.

Refreshing an event group leader refreshes all siblings and refreshing with a parameter of 0 currently enables infinite refreshes; these behaviors are unsupported and should not be relied on.

Starting with Linux 3.18, **POLL_HUP** is indicated if the event being monitored is attached to a different process and that process exits.

rdpmc instruction

Starting with Linux 3.4 on x86, you can use the *rdpmc* instruction to get low-latency reads without having to enter the kernel. Note that using *rdpmc* is not necessarily faster than other methods for reading event values.

Support for this can be detected with the *cap_usr_rdpmc* field in the mmap page; documentation on how to calculate event values can be found in that section.

Originally, when rdpmc support was enabled, any process (not just ones with an active perf event) could use the rdpmc instruction to access the counters. Starting with Linux 4.0, rdpmc support is only allowed if an event is currently enabled in a process's context. To restore the old behavior, write the value 2 to */sys/devices/cpu/rdpmc*.

perf_event ioctl calls

Various ioctls act on **perf_event_open**() file descriptors:

PERF_EVENT_IOC_ENABLE

This enables the individual event or event group specified by the file descriptor argument.

If the **PERF_IOC_FLAG_GROUP** bit is set in the ioctl argument, then all events in a group are enabled, even if the event specified is not the group leader (but see BUGS).

PERF_EVENT_IOC_DISABLE

This disables the individual counter or event group specified by the file descriptor argument.

Enabling or disabling the leader of a group enables or disables the entire group; that is, while the group leader is disabled, none of the counters in the group will count. Enabling or disabling a member of a group other than the leader affects only that counter; disabling a non-leader stops that counter from counting but doesn't affect any other counter.

If the **PERF_IOC_FLAG_GROUP** bit is set in the ioctl argument, then all events in a group are disabled, even if the event specified is not the group leader (but see BUGS).

PERF_EVENT_IOC_REFRESH

Non-inherited overflow counters can use this to enable a counter for a number of overflows specified by the argument, after which it is disabled. Subsequent calls of this ioctl add the argument value to the current count. An overflow notification with **POLL_IN** set will happen on each overflow until the count reaches 0; when that happens a notification with **POLL_HUP** set is sent and the event is disabled. Using an argument of 0 is considered undefined behavior.

PERF_EVENT_IOC_RESET

Reset the event count specified by the file descriptor argument to zero. This resets only the counts; there is no way to reset the multiplexing *time_enabled* or *time_running* values.

If the **PERF_IOC_FLAG_GROUP** bit is set in the ioctl argument, then all events in a group are reset, even if the event specified is not the group leader (but see BUGS).

PERF_EVENT_IOC_PERIOD
This updates the overflow period for the event.

Since Linux 3.7 (on ARM) and Linux 3.14 (all other architectures), the new period takes effect immediately. On older kernels, the new period did not take effect until after the next overflow.

The argument is a pointer to a 64-bit value containing the desired new period.

Prior to Linux 2.6.36, this ioctl always failed due to a bug in the kernel.

PERF_EVENT_IOC_SET_OUTPUT
This tells the kernel to report event notifications to the specified file descriptor rather than the default one. The file descriptors must all be on the same CPU.

The argument specifies the desired file descriptor, or −1 if output should be ignored.

PERF_EVENT_IOC_SET_FILTER (since Linux 2.6.33)
This adds an ftrace filter to this event.

The argument is a pointer to the desired ftrace filter.

PERF_EVENT_IOC_ID (since Linux 3.12)
This returns the event ID value for the given event file descriptor.

The argument is a pointer to a 64-bit unsigned integer to hold the result.

PERF_EVENT_IOC_SET_BPF (since Linux 4.1)
This allows attaching a Berkeley Packet Filter (BPF) program to an existing kprobe tracepoint event. You need **CAP_PERFMON** (since Linux 5.8) or **CAP_SYS_ADMIN** privileges to use this ioctl.

The argument is a BPF program file descriptor that was created by a previous *bpf*(2) system call.

PERF_EVENT_IOC_PAUSE_OUTPUT (since Linux 4.7)
This allows pausing and resuming the event's ring-buffer. A paused ring-buffer does not prevent generation of samples, but simply discards them. The discarded samples are considered lost, and cause a **PERF_RECORD_LOST** sample to be generated when possible. An overflow signal may still be triggered by the discarded sample even though the ring-buffer remains empty.

The argument is an unsigned 32-bit integer. A nonzero value pauses the ring-buffer, while a zero value resumes the ring-buffer.

PERF_EVENT_MODIFY_ATTRIBUTES (since Linux 4.17)
This allows modifying an existing event without the overhead of closing and reopening a new event. Currently this is supported only for breakpoint events.

The argument is a pointer to a *perf_event_attr* structure containing the updated event settings.

PERF_EVENT_IOC_QUERY_BPF (since Linux 4.16)
This allows querying which Berkeley Packet Filter (BPF) programs are attached to an existing kprobe tracepoint. You can only attach one BPF program per event, but you can have multiple events attached to a tracepoint. Querying this value on one tracepoint event returns the ID of all BPF programs in all events attached to the tracepoint. You need **CAP_PERFMON** (since Linux 5.8) or **CAP_SYS_ADMIN** privileges to use this ioctl.

The argument is a pointer to a structure

```
struct perf_event_query_bpf {
    __u32    ids_len;
    __u32    prog_cnt;
    __u32    ids[0];
};
```

The *ids_len* field indicates the number of ids that can fit in the provided *ids* array. The *prog_cnt* value is filled in by the kernel with the number of attached BPF programs. The *ids* array is filled with the ID of each attached BPF program. If there are more programs than will fit in the array, then the kernel will return **ENOSPC** and *ids_len* will indicate the number of

program IDs that were successfully copied.

Using prctl(2)

A process can enable or disable all currently open event groups using the *prctl*(2) **PR_TASK_PERF_EVENTS_ENABLE** and **PR_TASK_PERF_EVENTS_DISABLE** operations. This applies only to events created locally by the calling process. This does not apply to events created by other processes attached to the calling process or inherited events from a parent process. Only group leaders are enabled and disabled, not any other members of the groups.

perf_event related configuration files

Files in */proc/sys/kernel/*

/proc/sys/kernel/perf_event_paranoid

The */proc/sys/kernel/perf_event_paranoid* file can be set to restrict access to the performance counters.

2 allow only user-space measurements (default since Linux 4.6).
1 allow both kernel and user measurements (default before Linux 4.6).
0 allow access to CPU-specific data but not raw tracepoint samples.
−1 no restrictions.

The existence of the */proc/sys/kernel/perf_event_paranoid* file is the official method for determining if a kernel supports **perf_event_open**().

/proc/sys/kernel/perf_event_max_sample_rate

This sets the maximum sample rate. Setting this too high can allow users to sample at a rate that impacts overall machine performance and potentially lock up the machine. The default value is 100000 (samples per second).

/proc/sys/kernel/perf_event_max_stack

This file sets the maximum depth of stack frame entries reported when generating a call trace.

/proc/sys/kernel/perf_event_mlock_kb

Maximum number of pages an unprivileged user can *mlock*(2). The default is 516 (kB).

Files in */sys/bus/event_source/devices/*

Since Linux 2.6.34, the kernel supports having multiple PMUs available for monitoring. Information on how to program these PMUs can be found under */sys/bus/event_source/devices/*. Each subdirectory corresponds to a different PMU.

/sys/bus/event_source/devices//type* (since Linux 2.6.38)

This contains an integer that can be used in the *type* field of *perf_event_attr* to indicate that you wish to use this PMU.

/sys/bus/event_source/devices/cpu/rdpmc (since Linux 3.4)

If this file is 1, then direct user-space access to the performance counter registers is allowed via the rdpmc instruction. This can be disabled by echoing 0 to the file.

As of Linux 4.0 the behavior has changed, so that 1 now means only allow access to processes with active perf events, with 2 indicating the old allow-anyone-access behavior.

/sys/bus/event_source/devices//format/* (since Linux 3.4)

This subdirectory contains information on the architecture-specific subfields available for programming the various *config* fields in the *perf_event_attr* struct.

The content of each file is the name of the config field, followed by a colon, followed by a series of integer bit ranges separated by commas. For example, the file *event* may contain the value *config1:1,6−10,44* which indicates that event is an attribute that occupies bits 1,6–10, and 44 of *perf_event_attr::config1*.

/sys/bus/event_source/devices//events/* (since Linux 3.4)

This subdirectory contains files with predefined events. The contents are strings describing the event settings expressed in terms of the fields found in the previously mentioned *./format/* directory. These are not necessarily complete lists of all events supported by a PMU, but usually a subset of events deemed useful or interesting.

The content of each file is a list of attribute names separated by commas. Each entry has an optional value (either hex or decimal). If no value is specified, then it is assumed to be

a single-bit field with a value of 1. An example entry may look like this: *event=0x2,inv,ldlat=3*.

/sys/bus/event_source/devices//uevent*
This file is the standard kernel device interface for injecting hotplug events.

/sys/bus/event_source/devices//cpumask* (since Linux 3.7)
The *cpumask* file contains a comma-separated list of integers that indicate a representative CPU number for each socket (package) on the motherboard. This is needed when setting up uncore or northbridge events, as those PMUs present socket-wide events.

RETURN VALUE

On success, **perf_event_open**() returns the new file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

The errors returned by **perf_event_open**() can be inconsistent, and may vary across processor architectures and performance monitoring units.

E2BIG Returned if the *perf_event_attr size* value is too small (smaller than **PERF_ATTR_SIZE_VER0**), too big (larger than the page size), or larger than the kernel supports and the extra bytes are not zero. When **E2BIG** is returned, the *perf_event_attr size* field is overwritten by the kernel to be the size of the structure it was expecting.

EACCES
Returned when the requested event requires **CAP_PERFMON** (since Linux 5.8) or **CAP_SYS_ADMIN** permissions (or a more permissive perf_event paranoid setting). Some common cases where an unprivileged process may encounter this error: attaching to a process owned by a different user; monitoring all processes on a given CPU (i.e., specifying the *pid* argument as −1); and not setting *exclude_kernel* when the paranoid setting requires it.

EBADF
Returned if the *group_fd* file descriptor is not valid, or, if **PERF_FLAG_PID_CGROUP** is set, the cgroup file descriptor in *pid* is not valid.

EBUSY (since Linux 4.1)
Returned if another event already has exclusive access to the PMU.

EFAULT
Returned if the *attr* pointer points at an invalid memory address.

EINTR
Returned when trying to mix perf and ftrace handling for a uprobe.

EINVAL
Returned if the specified event is invalid. There are many possible reasons for this. A not-exhaustive list: *sample_freq* is higher than the maximum setting; the *cpu* to monitor does not exist; *read_format* is out of range; *sample_type* is out of range; the *flags* value is out of range; *exclusive* or *pinned* set and the event is not a group leader; the event *config* values are out of range or set reserved bits; the generic event selected is not supported; or there is not enough room to add the selected event.

EMFILE
Each opened event uses one file descriptor. If a large number of events are opened, the per-process limit on the number of open file descriptors will be reached, and no more events can be created.

ENODEV
Returned when the event involves a feature not supported by the current CPU.

ENOENT
Returned if the *type* setting is not valid. This error is also returned for some unsupported generic events.

ENOSPC
Prior to Linux 3.3, if there was not enough room for the event, **ENOSPC** was returned. In Linux 3.3, this was changed to **EINVAL**. **ENOSPC** is still returned if you try to add more breakpoint events than supported by the hardware.

ENOSYS
Returned if **PERF_SAMPLE_STACK_USER** is set in *sample_type* and it is not supported by hardware.

EOPNOTSUPP
Returned if an event requiring a specific hardware feature is requested but there is no hardware support. This includes requesting low-skid events if not supported, branch tracing if it is not available, sampling if no PMU interrupt is available, and branch stacks for software events.

EOVERFLOW (since Linux 4.8)
Returned if **PERF_SAMPLE_CALLCHAIN** is requested and *sample_max_stack* is larger than the maximum specified in */proc/sys/kernel/perf_event_max_stack*.

EPERM
Returned on many (but not all) architectures when an unsupported *exclude_hv*, *exclude_idle*, *exclude_user*, or *exclude_kernel* setting is specified.

It can also happen, as with **EACCES**, when the requested event requires **CAP_PERFMON** (since Linux 5.8) or **CAP_SYS_ADMIN** permissions (or a more permissive perf_event paranoid setting). This includes setting a breakpoint on a kernel address, and (since Linux 3.13) setting a kernel function-trace tracepoint.

ESRCH
Returned if attempting to attach to a process that does not exist.

STANDARDS
Linux.

HISTORY
perf_event_open() was introduced in Linux 2.6.31 but was called **perf_counter_open**(). It was renamed in Linux 2.6.32.

NOTES
The official way of knowing if **perf_event_open**() support is enabled is checking for the existence of the file */proc/sys/kernel/perf_event_paranoid*.

CAP_PERFMON capability (since Linux 5.8) provides secure approach to performance monitoring and observability operations in a system according to the principal of least privilege (POSIX IEEE 1003.1e). Accessing system performance monitoring and observability operations using **CAP_PERFMON** rather than the much more powerful **CAP_SYS_ADMIN** excludes chances to misuse credentials and makes operations more secure. **CAP_SYS_ADMIN** usage for secure system performance monitoring and observability is discouraged in favor of the **CAP_PERFMON** capability.

BUGS
The **F_SETOWN_EX** option to *fcntl*(2) is needed to properly get overflow signals in threads. This was introduced in Linux 2.6.32.

Prior to Linux 2.6.33 (at least for x86), the kernel did not check if events could be scheduled together until read time. The same happens on all known kernels if the NMI watchdog is enabled. This means to see if a given set of events works you have to **perf_event_open**(), start, then read before you know for sure you can get valid measurements.

Prior to Linux 2.6.34, event constraints were not enforced by the kernel. In that case, some events would silently return "0" if the kernel scheduled them in an improper counter slot.

Prior to Linux 2.6.34, there was a bug when multiplexing where the wrong results could be returned.

Kernels from Linux 2.6.35 to Linux 2.6.39 can quickly crash the kernel if "inherit" is enabled and many threads are started.

Prior to Linux 2.6.35, **PERF_FORMAT_GROUP** did not work with attached processes.

There is a bug in the kernel code between Linux 2.6.36 and Linux 3.0 that ignores the "watermark" field and acts as if a wakeup_event was chosen if the union has a nonzero value in it.

From Linux 2.6.31 to Linux 3.4, the **PERF_IOC_FLAG_GROUP** ioctl argument was broken and would repeatedly operate on the event specified rather than iterating across all sibling events in a group.

From Linux 3.4 to Linux 3.11, the mmap *cap_usr_rdpmc* and *cap_usr_time* bits mapped to the same

location. Code should migrate to the new *cap_user_rdpmc* and *cap_user_time* fields instead.

Always double-check your results! Various generalized events have had wrong values. For example, retired branches measured the wrong thing on AMD machines until Linux 2.6.35.

EXAMPLES

The following is a short example that measures the total instruction count of a call to *printf*(3).

```
#include <linux/perf_event.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/syscall.h>
#include <unistd.h>

static long
perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                int cpu, int group_fd, unsigned long flags)
{
    int ret;

    ret = syscall(SYS_perf_event_open, hw_event, pid, cpu,
                  group_fd, flags);
    return ret;
}

int
main(void)
{
    int                     fd;
    long long               count;
    struct perf_event_attr  pe;

    memset(&pe, 0, sizeof(pe));
    pe.type = PERF_TYPE_HARDWARE;
    pe.size = sizeof(pe);
    pe.config = PERF_COUNT_HW_INSTRUCTIONS;
    pe.disabled = 1;
    pe.exclude_kernel = 1;
    pe.exclude_hv = 1;

    fd = perf_event_open(&pe, 0, -1, -1, 0);
    if (fd == -1) {
       fprintf(stderr, "Error opening leader %llx\n", pe.config);
       exit(EXIT_FAILURE);
    }

    ioctl(fd, PERF_EVENT_IOC_RESET, 0);
    ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

    printf("Measuring instruction count for this printf\n");

    ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
    read(fd, &count, sizeof(count));

    printf("Used %lld instructions\n", count);

    close(fd);
}
```

SEE ALSO

perf (1), *fcntl*(2), *mmap*(2), *open*(2), *prctl*(2), *read*(2)

Documentation/admin−guide/perf−security.rst in the kernel source tree

NAME

perfmonctl − interface to IA-64 performance monitoring unit

SYNOPSIS

#include <syscall.h>
#include <perfmon.h>

long perfmonctl(int *fd***, int** *cmd***, void** *arg***[.***narg***], int** *narg***);**

Note: There is no glibc wrapper for this system call; see HISTORY.

DESCRIPTION

The IA-64-specific **perfmonctl**() system call provides an interface to the PMU (performance monitoring unit). The PMU consists of PMD (performance monitoring data) registers and PMC (performance monitoring control) registers, which gather hardware statistics.

perfmonctl() applies the operation *cmd* to the input arguments specified by *arg*. The number of arguments is defined by *narg*. The *fd* argument specifies the perfmon context to operate on.

Supported values for *cmd* are:

PFM_CREATE_CONTEXT
perfmonctl(int *fd***, PFM_CREATE_CONTEXT, pfarg_context_t *** *ctxt***, 1);**
Set up a context.

The *fd* parameter is ignored. A new perfmon context is created as specified in *ctxt* and its file descriptor is returned in *ctxt−>ctx_fd*.

The file descriptor can be used in subsequent calls to **perfmonctl**() and can be used to read event notifications (type *pfm_msg_t*) using *read*(2). The file descriptor is pollable using *select*(2), *poll*(2), and *epoll*(7).

The context can be destroyed by calling *close*(2) on the file descriptor.

PFM_WRITE_PMCS
perfmonctl(int *fd***, PFM_WRITE_PMCS, pfarg_reg_t *** *pmcs***, n);**
Set PMC registers.

PFM_WRITE_PMDS
perfmonctl(int *fd***, PFM_WRITE_PMDS, pfarg_reg_t *** *pmds***, n);**
Set PMD registers.

PFM_READ_PMDS
perfmonctl(int *fd***, PFM_READ_PMDS, pfarg_reg_t *** *pmds***, n);**
Read PMD registers.

PFM_START
perfmonctl(int *fd***, PFM_START, NULL, 0);**
Start monitoring.

PFM_STOP
perfmonctl(int *fd***, PFM_STOP, NULL, 0);**
Stop monitoring.

PFM_LOAD_CONTEXT
perfmonctl(int *fd***, PFM_LOAD_CONTEXT, pfarg_load_t *** *largs***, 1);**
Attach the context to a thread.

PFM_UNLOAD_CONTEXT
perfmonctl(int *fd***, PFM_UNLOAD_CONTEXT, NULL, 0);**
Detach the context from a thread.

PFM_RESTART
perfmonctl(int *fd***, PFM_RESTART, NULL, 0);**
Restart monitoring after receiving an overflow notification.

PFM_GET_FEATURES
perfmonctl(int *fd***, PFM_GET_FEATURES, pfarg_features_t *** *arg***, 1);**

PFM_DEBUG

perfmonctl(int *fd***, PFM_DEBUG,** *val***, 0);**
If *val* is nonzero, enable debugging mode, otherwise disable.

PFM_GET_PMC_RESET_VAL

perfmonctl(int *fd***, PFM_GET_PMC_RESET_VAL, pfarg_reg_t ****req***, n);**
Reset PMC registers to default values.

RETURN VALUE

perfmonctl() returns zero when the operation is successful. On error, −1 is returned and *errno* is set to indicate the error.

STANDARDS

Linux on IA-64.

HISTORY

Added in Linux 2.4; removed in Linux 5.10.

This system call was broken for many years, and ultimately removed in Linux 5.10.

glibc does not provide a wrapper for this system call; on kernels where it exists, call it using *syscall*(2).

SEE ALSO

gprof (1)

The perfmon2 interface specification

NAME

personality − set the process execution domain

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/personality.h>

int personality(unsigned long *persona***);**

DESCRIPTION

Linux supports different execution domains, or personalities, for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other UNIX-like operating systems.

If *persona* is not 0xffffffff, then **personality**() sets the caller's execution domain to the value specified by *persona*. Specifying *persona* as 0xffffffff provides a way of retrieving the current persona without changing it.

A list of the available execution domains can be found in *<sys/personality.h>*. The execution domain is a 32-bit value in which the top three bytes are set aside for flags that cause the kernel to modify the behavior of certain system calls so as to emulate historical or architectural quirks. The least significant byte is a value defining the personality the kernel should assume. The flag values are as follows:

ADDR_COMPAT_LAYOUT (since Linux 2.6.9)
With this flag set, provide legacy virtual address space layout.

ADDR_NO_RANDOMIZE (since Linux 2.6.12)
With this flag set, disable address-space-layout randomization.

ADDR_LIMIT_32BIT (since Linux 2.2)
Limit the address space to 32 bits.

ADDR_LIMIT_3GB (since Linux 2.4.0)
With this flag set, use 0xc0000000 as the offset at which to search a virtual memory chunk on *mmap*(2); otherwise use 0xffffe000. Applies to 32-bit x86 processes only.

FDPIC_FUNCPTRS (since Linux 2.6.11)
User-space function pointers to signal handlers point to descriptors. Applies only to ARM if BINFMT_ELF_FDPIC and SuperH.

MMAP_PAGE_ZERO (since Linux 2.4.0)
Map page 0 as read-only (to support binaries that depend on this SVr4 behavior).

READ_IMPLIES_EXEC (since Linux 2.6.8)
With this flag set, **PROT_READ** implies **PROT_EXEC** for *mmap*(2).

SHORT_INODE (since Linux 2.4.0)
No effect.

STICKY_TIMEOUTS (since Linux 1.2.0)
With this flag set, *select*(2), *pselect*(2), and *ppoll*(2) do not modify the returned timeout argument when interrupted by a signal handler.

UNAME26 (since Linux 3.1)
Have *uname*(2) report a 2.6.(40+x) version number rather than a MAJOR.x version number. Added as a stopgap measure to support broken applications that could not handle the kernel version-numbering switch from Linux 2.6.x to Linux 3.x.

WHOLE_SECONDS (since Linux 1.2.0)
No effect.

The available execution domains are:

PER_BSD (since Linux 1.2.0)
BSD. (No effects.)

PER_HPUX (since Linux 2.4)
Support for 32-bit HP/UX. This support was never complete, and was dropped so that since Linux 4.0, this value has no effect.

PER_IRIX32 (since Linux 2.2)
IRIX 5 32-bit. Never fully functional; support dropped in Linux 2.6.27. Implies **STICKY_TIMEOUTS**.

PER_IRIX64 (since Linux 2.2)
IRIX 6 64-bit. Implies **STICKY_TIMEOUTS**; otherwise no effect.

PER_IRIXN32 (since Linux 2.2)
IRIX 6 new 32-bit. Implies **STICKY_TIMEOUTS**; otherwise no effect.

PER_ISCR4 (since Linux 1.2.0)
Implies **STICKY_TIMEOUTS**; otherwise no effect.

PER_LINUX (since Linux 1.2.0)
Linux.

PER_LINUX32 (since Linux 2.2)
uname(2) returns the name of the 32-bit architecture in the *machine* field ("i686" instead of "x86_64", &c.).

Under ia64 (Itanium), processes with this personality don't have the O_LARGEFILE *open*(2) flag forced.

Under 64-bit ARM, setting this personality is forbidden if *execve*(2)ing a 32-bit process would also be forbidden (cf. the allow_mismatched_32bit_el0 kernel parameter and *Documentation/arm64/asymmetric-32bit.rst*).

PER_LINUX32_3GB (since Linux 2.4)
Same as **PER_LINUX32**, but implies **ADDR_LIMIT_3GB**.

PER_LINUX_32BIT (since Linux 2.0)
Same as **PER_LINUX**, but implies **ADDR_LIMIT_32BIT**.

PER_LINUX_FDPIC (since Linux 2.6.11)
Same as **PER_LINUX**, but implies **FDPIC_FUNCPTRS**.

PER_OSF4 (since Linux 2.4)
OSF/1 v4. No effect since Linux 6.1, which removed a.out binary support. Before, on alpha, would clear top 32 bits of iov_len in the user's buffer for compatibility with old versions of OSF/1 where iov_len was defined as. *int*.

PER_OSR5 (since Linux 2.4)
SCO OpenServer 5. Implies **STICKY_TIMEOUTS** and **WHOLE_SECONDS**; otherwise no effect.

PER_RISCOS (since Linux 2.3.7; macro since Linux 2.3.13)
Acorn RISC OS/Arthur (MIPS). No effect. Up to Linux v4.0, would set the emulation altroot to */usr/gnemul/riscos* (cf. **PER_SUNOS**, below). Before then, up to Linux 2.6.3, just Arthur emulation.

PER_SCOSVR3 (since Linux 1.2.0)
SCO UNIX System V Release 3. Same as **PER_OSR5**, but also implies **SHORT_INODE**.

PER_SOLARIS (since Linux 2.4)
Solaris. Implies **STICKY_TIMEOUTS**; otherwise no effect.

PER_SUNOS (since Linux 2.4.0)
Sun OS. Same as **PER_BSD**, but implies **STICKY_TIMEOUTS**. Prior to Linux 2.6.26, diverted library and dynamic linker searches to */usr/gnemul*. Buggy, largely unmaintained, and almost entirely unused.

PER_SVR3 (since Linux 1.2.0)
AT&T UNIX System V Release 3. Implies **STICKY_TIMEOUTS** and **SHORT_INODE**; otherwise no effect.

PER_SVR4 (since Linux 1.2.0)
AT&T UNIX System V Release 4. Implies **STICKY_TIMEOUTS** and **MMAP_PAGE_ZERO**; otherwise no effect.

PER_UW7 (since Linux 2.4)
UnixWare 7. Implies **STICKY_TIMEOUTS** and **MMAP_PAGE_ZERO**; otherwise no effect.

PER_WYSEV386 (since Linux 1.2.0)
WYSE UNIX System V/386. Implies **STICKY_TIMEOUTS** and **SHORT_INODE**; otherwise no effect.

PER_XENIX (since Linux 1.2.0)
XENIX. Implies **STICKY_TIMEOUTS** and **SHORT_INODE**; otherwise no effect.

RETURN VALUE

On success, the previous *persona* is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
The kernel was unable to change the personality.

STANDARDS

Linux.

HISTORY

Linux 1.1.20, glibc 2.3.

SEE ALSO

setarch(8)

NAME

pidfd_getfd − obtain a duplicate of another process's file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_pidfd_getfd, int pidfd, int targetfd,
            unsigned int flags);
```

Note: glibc provides no wrapper for **pidfd_getfd**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **pidfd_getfd**() system call allocates a new file descriptor in the calling process. This new file descriptor is a duplicate of an existing file descriptor, *targetfd*, in the process referred to by the PID file descriptor *pidfd*.

The duplicate file descriptor refers to the same open file description (see *open*(2)) as the original file descriptor in the process referred to by *pidfd*. The two file descriptors thus share file status flags and file offset. Furthermore, operations on the underlying file object (for example, assigning an address to a socket object using *bind*(2)) can equally be performed via the duplicate file descriptor.

The close-on-exec flag (**FD_CLOEXEC**; see *fcntl*(2)) is set on the file descriptor returned by **pidfd_getfd**().

The *flags* argument is reserved for future use. Currently, it must be specified as 0.

Permission to duplicate another process's file descriptor is governed by a ptrace access mode **PTRACE_MODE_ATTACH_REALCREDS** check (see *ptrace*(2)).

RETURN VALUE

On success, **pidfd_getfd**() returns a file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
: *pidfd* is not a valid PID file descriptor.

EBADF
: *targetfd* is not an open file descriptor in the process referred to by *pidfd*.

EINVAL
: *flags* is not 0.

EMFILE
: The per-process limit on the number of open file descriptors has been reached (see the description of **RLIMIT_NOFILE** in *getrlimit*(2)).

ENFILE
: The system-wide limit on the total number of open files has been reached.

EPERM
: The calling process did not have **PTRACE_MODE_ATTACH_REALCREDS** permissions (see *ptrace*(2)) over the process referred to by *pidfd*.

ESRCH
: The process referred to by *pidfd* does not exist (i.e., it has terminated and been waited on).

STANDARDS

Linux.

HISTORY

Linux 5.6.

NOTES

For a description of PID file descriptors, see *pidfd_open*(2).

The effect of **pidfd_getfd**() is similar to the use of **SCM_RIGHTS** messages described in *unix*(7), but

differs in the following respects:

- In order to pass a file descriptor using an **SCM_RIGHTS** message, the two processes must first establish a UNIX domain socket connection.
- The use of **SCM_RIGHTS** requires cooperation on the part of the process whose file descriptor is being copied. By contrast, no such cooperation is necessary when using **pidfd_getfd**().
- The ability to use **pidfd_getfd**() is restricted by a **PTRACE_MODE_ATTACH_REALCREDS** ptrace access mode check.

SEE ALSO

clone3(2), *dup*(2), *kcmp*(2), *pidfd_open*(2)

NAME

pidfd_open − obtain a file descriptor that refers to a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_pidfd_open, pid_t *pid***, unsigned int** *flags***);**

Note: glibc provides no wrapper for **pidfd_open**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **pidfd_open**() system call creates a file descriptor that refers to the process whose PID is specified in *pid*. The file descriptor is returned as the function result; the close-on-exec flag is set on the file descriptor.

The *flags* argument either has the value 0, or contains the following flag:

PIDFD_NONBLOCK (since Linux 5.10)
: Return a nonblocking file descriptor. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid*(2) will immediately return the error **EAGAIN** rather than blocking.

RETURN VALUE

On success, **pidfd_open**() returns a file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
: *flags* is not valid.

EINVAL
: *pid* is not valid.

EMFILE
: The per-process limit on the number of open file descriptors has been reached (see the description of **RLIMIT_NOFILE** in *getrlimit*(2)).

ENFILE
: The system-wide limit on the total number of open files has been reached.

ENODEV
: The anonymous inode filesystem is not available in this kernel.

ENOMEM
: Insufficient kernel memory was available.

ESRCH
: The process specified by *pid* does not exist.

STANDARDS

Linux.

HISTORY

Linux 5.3.

NOTES

The following code sequence can be used to obtain a file descriptor for the child of *fork*(2):

```
pid = fork();
if (pid > 0) {     /* If parent */
    pidfd = pidfd_open(pid, 0);
    ...
}
```

Even if the child has already terminated by the time of the **pidfd_open**() call, its PID will not have been recycled and the returned file descriptor will refer to the resulting zombie process. Note, however, that this is guaranteed only if the following conditions hold true:

- the disposition of **SIGCHLD** has not been explicitly set to **SIG_IGN** (see *sigaction*(2));
- the **SA_NOCLDWAIT** flag was not specified while establishing a handler for **SIGCHLD** or while setting the disposition of that signal to **SIG_DFL** (see *sigaction*(2)); and
- the zombie process was not reaped elsewhere in the program (e.g., either by an asynchronously executed signal handler or by *wait*(2) or similar in another thread).

If any of these conditions does not hold, then the child process (along with a PID file descriptor that refers to it) should instead be created using *clone*(2) with the **CLONE_PIDFD** flag.

Use cases for PID file descriptors

A PID file descriptor returned by **pidfd_open**() (or by *clone*(2) with the **CLONE_PID** flag) can be used for the following purposes:

- The *pidfd_send_signal*(2) system call can be used to send a signal to the process referred to by a PID file descriptor.
- A PID file descriptor can be monitored using *poll*(2), *select*(2), and *epoll*(7). When the process that it refers to terminates, these interfaces indicate the file descriptor as readable. Note, however, that in the current implementation, nothing can be read from the file descriptor (**read**(2) on the file descriptor fails with the error **EINVAL**).
- If the PID file descriptor refers to a child of the calling process, then it can be waited on using *waitid*(2).
- The *pidfd_getfd*(2) system call can be used to obtain a duplicate of a file descriptor of another process referred to by a PID file descriptor.
- A PID file descriptor can be used as the argument of *setns*(2) in order to move into one or more of the same namespaces as the process referred to by the file descriptor.
- A PID file descriptor can be used as the argument of *process_madvise*(2) in order to provide advice on the memory usage patterns of the process referred to by the file descriptor.

The **pidfd_open**() system call is the preferred way of obtaining a PID file descriptor for an already existing process. The alternative is to obtain a file descriptor by opening a */proc/*pid directory. However, the latter technique is possible only if the *proc*(5) filesystem is mounted; furthermore, the file descriptor obtained in this way is *not* pollable and can't be waited on with *waitid*(2).

EXAMPLES

The program below opens a PID file descriptor for the process whose PID is specified as its command-line argument. It then uses *poll*(2) to monitor the file descriptor for process exit, as indicated by an **EPOLLIN** event.

Program source

```
#define _GNU_SOURCE
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>

static int
pidfd_open(pid_t pid, unsigned int flags)
{
    return syscall(SYS_pidfd_open, pid, flags);
}

int
main(int argc, char *argv[])
{
    int            pidfd, ready;
    struct pollfd  pollfd;
```

```
        if (argc != 2) {
            fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
            exit(EXIT_SUCCESS);
        }

        pidfd = pidfd_open(atoi(argv[1]), 0);
        if (pidfd == -1) {
            perror("pidfd_open");
            exit(EXIT_FAILURE);
        }

        pollfd.fd = pidfd;
        pollfd.events = POLLIN;

        ready = poll(&pollfd, 1, -1);
        if (ready == -1) {
            perror("poll");
            exit(EXIT_FAILURE);
        }

        printf("Events (%#x): POLLIN is %sset\n", pollfd.revents,
               (pollfd.revents & POLLIN) ? "" : "not ");

        close(pidfd);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

clone(2), *kill*(2), *pidfd_getfd*(2), *pidfd_send_signal*(2), *poll*(2), *process_madvise*(2), *select*(2), *setns*(2), *waitid*(2), *epoll*(7)

NAME

pidfd_send_signal − send a signal to a process specified by a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/signal.h>      /* Definition of SIG* constants */
#include <signal.h>            /* Definition of SI_* constants */
#include <sys/syscall.h>       /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_pidfd_send_signal, int pidfd, int sig,
            siginfo_t *_Nullable info, unsigned int flags);
```

Note: glibc provides no wrapper for **pidfd_send_signal**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **pidfd_send_signal**() system call sends the signal *sig* to the target process referred to by *pidfd*, a PID file descriptor that refers to a process.

If the *info* argument points to a *siginfo_t* buffer, that buffer should be populated as described in *rt_sigqueueinfo*(2).

If the *info* argument is a null pointer, this is equivalent to specifying a pointer to a *siginfo_t* buffer whose fields match the values that are implicitly supplied when a signal is sent using *kill*(2):

- *si_signo* is set to the signal number;
- *si_errno* is set to 0;
- *si_code* is set to **SI_USER**;
- *si_pid* is set to the caller's PID; and
- *si_uid* is set to the caller's real user ID.

The calling process must either be in the same PID namespace as the process referred to by *pidfd*, or be in an ancestor of that namespace.

The *flags* argument is reserved for future use; currently, this argument must be specified as 0.

RETURN VALUE

On success, **pidfd_send_signal**() returns 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
: *pidfd* is not a valid PID file descriptor.

EINVAL
: *sig* is not a valid signal.

EINVAL
: The calling process is not in a PID namespace from which it can send a signal to the target process.

EINVAL
: *flags* is not 0.

EPERM
: The calling process does not have permission to send the signal to the target process.

EPERM
: *pidfd* doesn't refer to the calling process, and *info.si_code* is invalid (see *rt_sigqueueinfo*(2)).

ESRCH
: The target process does not exist (i.e., it has terminated and been waited on).

STANDARDS

Linux.

HISTORY

Linux 5.1.

NOTES

PID file descriptors

The *pidfd* argument is a PID file descriptor, a file descriptor that refers to process. Such a file descriptor can be obtained in any of the following ways:

- by opening a */proc/* pid directory;
- using *pidfd_open*(2); or
- via the PID file descriptor that is returned by a call to *clone*(2) or *clone3*(2) that specifies the **CLONE_PIDFD** flag.

The **pidfd_send_signal**() system call allows the avoidance of race conditions that occur when using traditional interfaces (such as *kill*(2)) to signal a process. The problem is that the traditional interfaces specify the target process via a process ID (PID), with the result that the sender may accidentally send a signal to the wrong process if the originally intended target process has terminated and its PID has been recycled for another process. By contrast, a PID file descriptor is a stable reference to a specific process; if that process terminates, **pidfd_send_signal**() fails with the error **ESRCH**.

EXAMPLES

```
#define _GNU_SOURCE
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

static int
pidfd_send_signal(int pidfd, int sig, siginfo_t *info,
                  unsigned int flags)
{
    return syscall(SYS_pidfd_send_signal, pidfd, sig, info, flags);
}

int
main(int argc, char *argv[])
{
    int        pidfd, sig;
    char       path[PATH_MAX];
    siginfo_t  info;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <pid> <signal>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    sig = atoi(argv[2]);

    /* Obtain a PID file descriptor by opening the /proc/PID directory
       of the target process. */

    snprintf(path, sizeof(path), "/proc/%s", argv[1]);

    pidfd = open(path, O_RDONLY);
    if (pidfd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
```

```
        /* Populate a 'siginfo_t' structure for use with
           pidfd_send_signal(). */

        memset(&info, 0, sizeof(info));
        info.si_code = SI_QUEUE;
        info.si_signo = sig;
        info.si_errno = 0;
        info.si_uid = getuid();
        info.si_pid = getpid();
        info.si_value.sival_int = 1234;

        /* Send the signal. */

        if (pidfd_send_signal(pidfd, sig, &info, 0) == -1) {
            perror("pidfd_send_signal");
            exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

clone(2), *kill*(2), *pidfd_open*(2), *rt_sigqueueinfo*(2), *sigaction*(2), *pid_namespaces*(7), *signal*(7)

NAME

pipe, pipe2 − create pipe

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int pipe(int pipefd[2]);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>          /* Definition of O_* constants */
#include <unistd.h>

int pipe2(int pipefd[2], int flags);

/* On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64, pipe() has the
   following prototype; see VERSIONS */

#include <unistd.h>

struct fd_pair {
    long fd[2];
};
struct fd_pair pipe(void);
```

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see *pipe*(7).

If *flags* is 0, then **pipe2**() is the same as **pipe**(). The following values can be bitwise ORed in *flags* to obtain different behavior:

O_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the two new file descriptors. See the description of the same flag in *open*(2) for reasons why this may be useful.

O_DIRECT (since Linux 3.4)
: Create a pipe that performs I/O in "packet" mode. Each *write*(2) to the pipe is dealt with as a separate packet, and *read*(2)s from the pipe will read one packet at a time. Note the following points:

  - Writes of greater than **PIPE_BUF** bytes (see *pipe*(7)) will be split into multiple packets. The constant **PIPE_BUF** is defined in <*limits.h*>.
  - If a *read*(2) specifies a buffer size that is smaller than the next packet, then the requested number of bytes are read, and the excess bytes in the packet are discarded. Specifying a buffer size of **PIPE_BUF** will be sufficient to read the largest possible packets (see the previous point).
  - Zero-length packets are not supported. (A *read*(2) that specifies a buffer size of zero is a no-op, and returns 0.)

  Older kernels that do not support this flag will indicate this via an **EINVAL** error.

  Since Linux 4.5, it is possible to change the **O_DIRECT** setting of a pipe file descriptor using *fcntl*(2).

O_NONBLOCK
: Set the **O_NONBLOCK** file status flag on the open file descriptions referred to by the new file descriptors. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

O_NOTIFICATION_PIPE
: Since Linux 5.8, general notification mechanism is built on the top of the pipe where kernel splices notification messages into pipes opened by user space. The owner of the pipe has to tell the kernel which sources of events to watch and filters can also be applied to select which

subevents should be placed into the pipe.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, *errno* is set to indicate the error, and *pipefd* is left unchanged.

On Linux (and other systems), **pipe**() does not modify *pipefd* on failure. A requirement standardizing this behavior was added in POSIX.1-2008 TC2. The Linux-specific **pipe2**() system call likewise does not modify *pipefd* on failure.

ERRORS

EFAULT
pipefd is not valid.

EINVAL
(**pipe2**()) Invalid value in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENFILE
The user hard limit on memory that can be allocated for pipes has been reached and the caller is not privileged; see *pipe*(7).

ENOPKG
(**pipe2**()) **O_NOTIFICATION_PIPE** was passed in *flags* and support for notifications (**CONFIG_WATCH_QUEUE**) is not compiled into the kernel.

VERSIONS

The System V ABI on some architectures allows the use of more than one register for returning multiple values; several architectures (namely, Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64) (ab)use this feature in order to implement the **pipe**() system call in a functional manner: the call doesn't take any arguments and returns a pair of file descriptors as the return value on success. The glibc **pipe**() wrapper function transparently deals with this. See *syscall*(2) for information regarding registers used for storing second file descriptor.

STANDARDS

pipe() POSIX.1-2008.

pipe2() Linux.

HISTORY

pipe() POSIX.1-2001.

pipe2() Linux 2.6.27, glibc 2.9.

EXAMPLES

The following program creates a pipe, and then *fork*(2)s to create a child process; the child inherits a duplicate set of file descriptors that refer to the same pipe. After the *fork*(2), each process closes the file descriptors that it doesn't need for the pipe (see *pipe*(7)). The parent then writes the string contained in the program's command-line argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int    pipefd[2];
    char   buf;
```

```
        pid_t  cpid;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s <string>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        if (pipe(pipefd) == -1) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }

        cpid = fork();
        if (cpid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if (cpid == 0) {    /* Child reads from pipe */
            close(pipefd[1]);          /* Close unused write end */

            while (read(pipefd[0], &buf, 1) > 0)
                write(STDOUT_FILENO, &buf, 1);

            write(STDOUT_FILENO, "\n", 1);
            close(pipefd[0]);
            _exit(EXIT_SUCCESS);

        } else {            /* Parent writes argv[1] to pipe */
            close(pipefd[0]);          /* Close unused read end */
            write(pipefd[1], argv[1], strlen(argv[1]));
            close(pipefd[1]);          /* Reader will see EOF */
            wait(NULL);                /* Wait for child */
            exit(EXIT_SUCCESS);
        }
    }
```

SEE ALSO

fork(2), *read*(2), *socketpair*(2), *splice*(2), *tee*(2), *vmsplice*(2), *write*(2), *popen*(3), *pipe*(7)

NAME

pivot_root − change the root mount

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_pivot_root, const char **new_root*, const char **put_old*);

Note: glibc provides no wrapper for **pivot_root**(), necessitating the use of *syscall*(2).

DESCRIPTION

pivot_root() changes the root mount in the mount namespace of the calling process. More precisely, it moves the root mount to the directory *put_old* and makes *new_root* the new root mount. The calling process must have the **CAP_SYS_ADMIN** capability in the user namespace that owns the caller's mount namespace.

pivot_root() changes the root directory and the current working directory of each process or thread in the same mount namespace to *new_root* if they point to the old root directory. (See also NOTES.) On the other hand, **pivot_root**() does not change the caller's current working directory (unless it is on the old root directory), and thus it should be followed by a **chdir("/")** call.

The following restrictions apply:

- *new_root* and *put_old* must be directories.
- *new_root* and *put_old* must not be on the same mount as the current root.
- *put_old* must be at or underneath *new_root*; that is, adding some nonnegative number of "/.." suffixes to the pathname pointed to by *put_old* must yield the same directory as *new_root*.
- *new_root* must be a path to a mount point, but can't be *"/"*. A path that is not already a mount point can be converted into one by bind mounting the path onto itself.
- The propagation type of the parent mount of *new_root* and the parent mount of the current root directory must not be **MS_SHARED**; similarly, if *put_old* is an existing mount point, its propagation type must not be **MS_SHARED**. These restrictions ensure that **pivot_root**() never propagates any changes to another mount namespace.
- The current root directory must be a mount point.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

pivot_root() may fail with any of the same errors as *stat*(2). Additionally, it may fail with the following errors:

EBUSY
new_root or *put_old* is on the current root mount. (This error covers the pathological case where *new_root* is *"/"*.)

EINVAL
new_root is not a mount point.

EINVAL
put_old is not at or underneath *new_root*.

EINVAL
The current root directory is not a mount point (because of an earlier *chroot*(2)).

EINVAL
The current root is on the rootfs (initial ramfs) mount; see NOTES.

EINVAL
Either the mount point at *new_root*, or the parent mount of that mount point, has propagation type **MS_SHARED**.

EINVAL
put_old is a mount point and has the propagation type **MS_SHARED**.

ENOTDIR
new_root or *put_old* is not a directory.

EPERM
The calling process does not have the **CAP_SYS_ADMIN** capability.

STANDARDS
Linux.

HISTORY
Linux 2.3.41.

NOTES
A command-line interface for this system call is provided by *pivot_root*(8)

pivot_root() allows the caller to switch to a new root filesystem while at the same time placing the old root mount at a location under *new_root* from where it can subsequently be unmounted. (The fact that it moves all processes that have a root directory or current working directory on the old root directory to the new root frees the old root directory of users, allowing the old root mount to be unmounted more easily.)

One use of **pivot_root**() is during system startup, when the system mounts a temporary root filesystem (e.g., an *initrd*(4)), then mounts the real root filesystem, and eventually turns the latter into the root directory of all relevant processes and threads. A modern use is to set up a root filesystem during the creation of a container.

The fact that **pivot_root**() modifies process root and current working directories in the manner noted in DESCRIPTION is necessary in order to prevent kernel threads from keeping the old root mount busy with their root and current working directories, even if they never access the filesystem in any way.

The rootfs (initial ramfs) cannot be **pivot_root**()ed. The recommended method of changing the root filesystem in this case is to delete everything in rootfs, overmount rootfs with the new root, attach *stdin*/*stdout*/*stderr* to the new */dev/console*, and exec the new *init*(1)Helper programs for this process exist; see *switch_root*(8)

pivot_root(".", ".")
new_root and *put_old* may be the same directory. In particular, the following sequence allows a pivot-root operation without needing to create and remove a temporary directory:

```
chdir(new_root);
pivot_root(".", ".");
umount2(".", MNT_DETACH);
```

This sequence succeeds because the **pivot_root**() call stacks the old root mount point on top of the new root mount point at `/`. At that point, the calling process's root directory and current working directory refer to the new root mount point (*new_root*). During the subsequent **umount**() call, resolution of *"."* starts with *new_root* and then moves up the list of mounts stacked at `/`, with the result that old root mount point is unmounted.

Historical notes
For many years, this manual page carried the following text:

> **pivot_root**() may or may not change the current root and the current working directory of any processes or threads which use the old root directory. The caller of **pivot_root**() must ensure that processes with root or current working directory at the old root operate correctly in either case. An easy way to ensure this is to change their root and current working directory to *new_root* before invoking **pivot_root**().

This text, written before the system call implementation was even finalized in the kernel, was probably intended to warn users at that time that the implementation might change before final release. However, the behavior stated in DESCRIPTION has remained consistent since this system call was first implemented and will not change now.

EXAMPLES

The program below demonstrates the use of **pivot_root**() inside a mount namespace that is created using *clone*(2). After pivoting to the root directory named in the program's first command-line argument, the child created by *clone*(2) then executes the program named in the remaining command-line arguments.

We demonstrate the program by creating a directory that will serve as the new root filesystem and placing a copy of the (statically linked) *busybox*(1) executable in that directory.

```
$ mkdir /tmp/rootfs
$ ls -id /tmp/rootfs    # Show inode number of new root directory
319459 /tmp/rootfs
$ cp $(which busybox) /tmp/rootfs
$ PS1='bbsh$ ' sudo ./pivot_root_demo /tmp/rootfs /busybox sh
bbsh$ PATH=/
bbsh$ busybox ln busybox ln
bbsh$ ln busybox echo
bbsh$ ln busybox ls
bbsh$ ls
busybox  echo     ln       ls
bbsh$ ls -id /          # Compare with inode number above
319459 /
bbsh$ echo 'hello world'
hello world
```

Program source

```
/* pivot_root_demo.c */

#define _GNU_SOURCE
#include <err.h>
#include <limits.h>
#include <sched.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <unistd.h>

static int
pivot_root(const char *new_root, const char *put_old)
{
    return syscall(SYS_pivot_root, new_root, put_old);
}

#define STACK_SIZE (1024 * 1024)

static int              /* Startup function for cloned child */
child(void *arg)
{
    char        path[PATH_MAX];
    char        **args = arg;
    char        *new_root = args[0];
    const char  *put_old = "/oldrootfs";

    /* Ensure that 'new_root' and its parent mount don't have
```

```
           shared propagation (which would cause pivot_root() to
           return an error), and prevent propagation of mount
           events to the initial mount namespace. */

        if (mount(NULL, "/", NULL, MS_REC | MS_PRIVATE, NULL) == -1)
            err(EXIT_FAILURE, "mount-MS_PRIVATE");

        /* Ensure that 'new_root' is a mount point. */

        if (mount(new_root, new_root, NULL, MS_BIND, NULL) == -1)
            err(EXIT_FAILURE, "mount-MS_BIND");

        /* Create directory to which old root will be pivoted. */

        snprintf(path, sizeof(path), "%s/%s", new_root, put_old);
        if (mkdir(path, 0777) == -1)
            err(EXIT_FAILURE, "mkdir");

        /* And pivot the root filesystem. */

        if (pivot_root(new_root, path) == -1)
            err(EXIT_FAILURE, "pivot_root");

        /* Switch the current working directory to "/". */

        if (chdir("/") == -1)
            err(EXIT_FAILURE, "chdir");

        /* Unmount old root and remove mount point. */

        if (umount2(put_old, MNT_DETACH) == -1)
            perror("umount2");
        if (rmdir(put_old) == -1)
            perror("rmdir");

        /* Execute the command specified in argv[1]... */

        execv(args[1], &args[1]);
        err(EXIT_FAILURE, "execv");
    }

    int
    main(int argc, char *argv[])
    {
        char *stack;

        /* Create a child process in a new mount namespace. */

        stack = mmap(NULL, STACK_SIZE, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
        if (stack == MAP_FAILED)
            err(EXIT_FAILURE, "mmap");

        if (clone(child, stack + STACK_SIZE,
                  CLONE_NEWNS | SIGCHLD, &argv[1]) == -1)
            err(EXIT_FAILURE, "clone");

        /* Parent falls through to here; wait for child. */
```

```
        if (wait(NULL) == -1)
            err(EXIT_FAILURE, "wait");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

chdir(2), *chroot*(2), *mount*(2), *stat*(2), *initrd*(4), *mount_namespaces*(7), *pivot_root*(8), *switch_root*(8)

NAME

pkey_alloc, pkey_free − allocate or free a protection key

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_alloc(unsigned int flags, unsigned int access_rights);
int pkey_free(int pkey);
```

DESCRIPTION

pkey_alloc() allocates a protection key (pkey) and allows it to be passed to *pkey_mprotect*(2).

The **pkey_alloc**() *flags* is reserved for future use and currently must always be specified as 0.

The **pkey_alloc**() *access_rights* argument may contain zero or more disable operations:

PKEY_DISABLE_ACCESS
: Disable all data access to memory covered by the returned protection key.

PKEY_DISABLE_WRITE
: Disable write access to memory covered by the returned protection key.

pkey_free() frees a protection key and makes it available for later allocations. After a protection key has been freed, it may no longer be used in any protection-key-related operations.

An application should not call **pkey_free**() on any protection key which has been assigned to an address range by *pkey_mprotect*(2) and which is still in use. The behavior in this case is undefined and may result in an error.

RETURN VALUE

On success, **pkey_alloc**() returns a positive protection key value. On success, **pkey_free**() returns zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: *pkey*, *flags*, or *access_rights* is invalid.

ENOSPC
: (**pkey_alloc**()) All protection keys available for the current process have been allocated. The number of keys available is architecture-specific and implementation-specific and may be reduced by kernel-internal use of certain keys. There are currently 15 keys available to user programs on x86.

  This error will also be returned if the processor or operating system does not support protection keys. Applications should always be prepared to handle this error, since factors outside of the application's control can reduce the number of available pkeys.

STANDARDS

Linux.

HISTORY

Linux 4.9, glibc 2.27.

NOTES

pkey_alloc() is always safe to call regardless of whether or not the operating system supports protection keys. It can be used in lieu of any other mechanism for detecting pkey support and will simply fail with the error **ENOSPC** if the operating system has no pkey support.

The kernel guarantees that the contents of the hardware rights register (PKRU) will be preserved only for allocated protection keys. Any time a key is unallocated (either before the first call returning that key from **pkey_alloc**() or after it is freed via *pkey_free*()), the kernel may make arbitrary changes to the parts of the rights register affecting access to that key.

EXAMPLES

See *pkeys*(7).

SEE ALSO

pkey_mprotect(2), *pkeys*(7)

NAME

poll, ppoll − wait for some event on a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <poll.h>

int poll(struct pollfd * fds, nfds_t nfds, int timeout);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <poll.h>

int ppoll(struct pollfd * fds, nfds_t nfds,
      const struct timespec *_Nullable tmo_p,
      const sigset_t *_Nullable sigmask);
```

DESCRIPTION

poll() performs a similar task to *select*(2): it waits for one of a set of file descriptors to become ready to perform I/O. The Linux-specific *epoll*(7) API performs a similar task, but offers features beyond those found in **poll**().

The set of file descriptors to be monitored is specified in the *fds* argument, which is an array of structures of the following form:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

The caller should specify the number of items in the *fds* array in *nfds*.

The field *fd* contains a file descriptor for an open file. If this field is negative, then the corresponding *events* field is ignored and the *revents* field returns zero. (This provides an easy way of ignoring a file descriptor for a single **poll**() call: simply set the *fd* field to its bitwise complement.)

The field *events* is an input parameter, a bit mask specifying the events the application is interested in for the file descriptor *fd*. This field may be specified as zero, in which case the only events that can be returned in *revents* are **POLLHUP**, **POLLERR**, and **POLLNVAL** (see below).

The field *revents* is an output parameter, filled by the kernel with the events that actually occurred. The bits returned in *revents* can include any of those specified in *events*, or one of the values **POLLERR**, **POLLHUP**, or **POLLNVAL**. (These three bits are meaningless in the *events* field, and will be set in the *revents* field whenever the corresponding condition is true.)

If none of the events requested (and no error) has occurred for any of the file descriptors, then **poll**() blocks until one of the events occurs.

The *timeout* argument specifies the number of milliseconds that **poll**() should block waiting for a file descriptor to become ready. The call will block until either:

- a file descriptor becomes ready;
- the call is interrupted by a signal handler; or
- the timeout expires.

Being "ready" means that the requested operation will not block; thus, **poll**()ing regular files, block devices, and other files with no reasonable polling semantic *always* returns instantly as ready to read and write.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. Specifying a negative value in *timeout* means an infinite timeout. Specifying a *timeout* of zero causes **poll**() to return immediately, even if no file descriptors are ready.

The bits that may be set/returned in *events* and *revents* are defined in <*poll.h*>:

POLLIN
There is data to read.

POLLPRI
There is some exceptional condition on the file descriptor. Possibilities include:

- There is out-of-band data on a TCP socket (see *tcp*(7)).
- A pseudoterminal master in packet mode has seen a state change on the slave (see *ioctl_tty*(2)).
- A *cgroup.events* file has been modified (see *cgroups*(7)).

POLLOUT
Writing is now possible, though a write larger than the available space in a socket or pipe will still block (unless **O_NONBLOCK** is set).

POLLRDHUP (since Linux 2.6.17)
Stream socket peer closed connection, or shut down writing half of connection. The **_GNU_SOURCE** feature test macro must be defined (before including *any* header files) in order to obtain this definition.

POLLERR
Error condition (only returned in *revents*; ignored in *events*). This bit is also set for a file descriptor referring to the write end of a pipe when the read end has been closed.

POLLHUP
Hang up (only returned in *revents*; ignored in *events*). Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer closed its end of the channel. Subsequent reads from the channel will return 0 (end of file) only after all outstanding data in the channel has been consumed.

POLLNVAL
Invalid request: *fd* not open (only returned in *revents*; ignored in *events*).

When compiling with **_XOPEN_SOURCE** defined, one also has the following, which convey no further information beyond the bits listed above:

POLLRDNORM
Equivalent to **POLLIN**.

POLLRDBAND
Priority band data can be read (generally unused on Linux).

POLLWRNORM
Equivalent to **POLLOUT**.

POLLWRBAND
Priority data may be written.

Linux also knows about, but does not use **POLLMSG**.

ppoll()

The relationship between **poll**() and **ppoll**() is analogous to the relationship between *select*(2) and *pselect*(2): like *pselect*(2), **ppoll**() allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

Other than the difference in the precision of the *timeout* argument, the following **ppoll**() call:

```
ready = ppoll(&fds, nfds, tmo_p, &sigmask);
```

is nearly equivalent to *atomically* executing the following calls:

```
sigset_t origmask;
int timeout;

timeout = (tmo_p == NULL) ? -1 :
          (tmo_p->tv_sec * 1000 + tmo_p->tv_nsec / 1000000);
pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = poll(&fds, nfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The above code segment is described as *nearly* equivalent because whereas a negative *timeout* value for **poll**() is interpreted as an infinite timeout, a negative value expressed in **tmo_p* results in an error from **ppoll**().

See the description of *pselect*(2) for an explanation of why **ppoll**() is necessary.

If the *sigmask* argument is specified as NULL, then no signal mask manipulation is performed (and thus **ppoll**() differs from **poll**() only in the precision of the *timeout* argument).

The *tmo_p* argument specifies an upper limit on the amount of time that **ppoll**() will block. This argument is a pointer to a *timespec*(3) structure.

If *tmo_p* is specified as NULL, then **ppoll**() can block indefinitely.

RETURN VALUE

On success, **poll**() returns a nonnegative value which is the number of elements in the *pollfds* whose *revents* fields have been set to a nonzero value (indicating an event or an error). A return value of zero indicates that the system call timed out before any file descriptors became ready.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: *fds* points outside the process's accessible address space. The array given as argument was not contained in the calling program's address space.

EINTR
: A signal occurred before any requested event; see *signal*(7).

EINVAL
: The *nfds* value exceeds the **RLIMIT_NOFILE** value.

EINVAL
: (**ppoll**()) The timeout value expressed in **tmo_p* is invalid (negative).

ENOMEM
: Unable to allocate memory for kernel data structures.

VERSIONS

On some other UNIX systems, **poll**() can fail with the error **EAGAIN** if the system fails to allocate kernel-internal resources, rather than **ENOMEM** as Linux does. POSIX permits this behavior. Portable programs may wish to check for **EAGAIN** and loop, just as with **EINTR**.

Some implementations define the nonstandard constant **INFTIM** with the value −1 for use as a *timeout* for **poll**(). This constant is not provided in glibc.

C library/kernel differences

The Linux **ppoll**() system call modifies its *tmo_p* argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc **ppoll**() function does not modify its *tmo_p* argument.

The raw **ppoll**() system call has a fifth argument, *size_t sigsetsize*, which specifies the size in bytes of the *sigmask* argument. The glibc **ppoll**() wrapper function specifies this argument as a fixed value (equal to *sizeof(kernel_sigset_t)*). See *sigprocmask*(2) for a discussion on the differences between the kernel and the libc notion of the sigset.

STANDARDS

poll() POSIX.1-2008.

ppoll() Linux.

HISTORY

poll() POSIX.1-2001. Linux 2.1.23.

On older kernels that lack this system call, the glibc **poll**() wrapper function provides emulation using *select*(2).

ppoll() Linux 2.6.16, glibc 2.4.

NOTES

The operation of **poll**() and **ppoll**() is not affected by the **O_NONBLOCK** flag.

For a discussion of what may happen if a file descriptor being monitored by **poll**() is closed in another thread, see *select*(2).

BUGS

See the discussion of spurious readiness notifications under the BUGS section of *select*(2).

EXAMPLES

The program below opens each of the files named in its command-line arguments and monitors the resulting file descriptors for readiness to read (**POLLIN**). The program loops, repeatedly using **poll**() to monitor the file descriptors, printing the number of ready file descriptors on return. For each ready file descriptor, the program:

- displays the returned *revents* field in a human-readable form;
- if the file descriptor is readable, reads some data from it, and displays that data on standard output; and
- if the file descriptor was not readable, but some other event occurred (presumably **POLLHUP**), closes the file descriptor.

Suppose we run the program in one terminal, asking it to open a FIFO:

```
$ mkfifo myfifo
$ ./poll_input myfifo
```

In a second terminal window, we then open the FIFO for writing, write some data to it, and close the FIFO:

```
$ echo aaaaabbbbbccccc > myfifo
```

In the terminal where we are running the program, we would then see:

```
Opened "myfifo" on fd 3
About to poll()
Ready: 1
  fd=3; events: POLLIN POLLHUP
    read 10 bytes: aaaaabbbbb
About to poll()
Ready: 1
  fd=3; events: POLLIN POLLHUP
    read 6 bytes: ccccc

About to poll()
Ready: 1
  fd=3; events: POLLHUP
    closing fd 3
All file descriptors closed; bye
```

In the above output, we see that **poll**() returned three times:

- On the first return, the bits returned in the *revents* field were **POLLIN**, indicating that the file descriptor is readable, and **POLLHUP**, indicating that the other end of the FIFO has been closed. The program then consumed some of the available input.
- The second return from **poll**() also indicated **POLLIN** and **POLLHUP**; the program then consumed the last of the available input.
- On the final return, **poll**() indicated only **POLLHUP** on the FIFO, at which point the file descriptor was closed and the program terminated.

Program source

```
/* poll_input.c

   Licensed under GNU General Public License v2 or later.
*/
```

```
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    int            ready;
    char           buf[10];
    nfds_t         num_open_fds, nfds;
    ssize_t        s;
    struct pollfd  *pfds;

    if (argc < 2) {
       fprintf(stderr, "Usage: %s file...\n", argv[0]);
       exit(EXIT_FAILURE);
    }

    num_open_fds = nfds = argc - 1;
    pfds = calloc(nfds, sizeof(struct pollfd));
    if (pfds == NULL)
        errExit("malloc");

    /* Open each file on command line, and add it to 'pfds' array. */

    for (nfds_t j = 0; j < nfds; j++) {
        pfds[j].fd = open(argv[j + 1], O_RDONLY);
        if (pfds[j].fd == -1)
            errExit("open");

        printf("Opened \"%s\" on fd %d\n", argv[j + 1], pfds[j].fd);

        pfds[j].events = POLLIN;
    }

    /* Keep calling poll() as long as at least one file descriptor is
       open. */

    while (num_open_fds > 0) {
        printf("About to poll()\n");
        ready = poll(pfds, nfds, -1);
        if (ready == -1)
            errExit("poll");

        printf("Ready: %d\n", ready);

        /* Deal with array returned by poll(). */

        for (nfds_t j = 0; j < nfds; j++) {
            if (pfds[j].revents != 0) {
                printf("  fd=%d; events: %s%s%s\n", pfds[j].fd,
                       (pfds[j].revents & POLLIN)  ? "POLLIN "  : "",
                       (pfds[j].revents & POLLHUP) ? "POLLHUP " : "",
                       (pfds[j].revents & POLLERR) ? "POLLERR " : "");
```

```
                    if (pfds[j].revents & POLLIN) {
                        s = read(pfds[j].fd, buf, sizeof(buf));
                        if (s == -1)
                            errExit("read");
                        printf("    read %zd bytes: %.*s\n",
                               s, (int) s, buf);
                    } else {                /* POLLERR | POLLHUP */
                        printf("    closing fd %d\n", pfds[j].fd);
                        if (close(pfds[j].fd) == -1)
                            errExit("close");
                        num_open_fds--;
                    }
                }
            }
        }

        printf("All file descriptors closed; bye\n");
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

restart_syscall(2), *select*(2), *select_tut*(2), *timespec*(3), *epoll*(7), *time*(7)

NAME

posix_fadvise − predeclare an access pattern for file data

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

posix_fadvise():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

Programs can use **posix_fadvise**() to announce an intention to access file data in a specific pattern in the future, thus allowing the kernel to perform appropriate optimizations.

The *advice* applies to a (not necessarily existent) region starting at *offset* and extending for *len* bytes (or until the end of the file if *len* is 0) within the file referred to by *fd*. The *advice* is not binding; it merely constitutes an expectation on behalf of the application.

Permissible values for *advice* include:

POSIX_FADV_NORMAL
: Indicates that the application has no advice to give about its access pattern for the specified data. If no advice is given for an open file, this is the default assumption.

POSIX_FADV_SEQUENTIAL
: The application expects to access the specified data sequentially (with lower offsets read before higher ones).

POSIX_FADV_RANDOM
: The specified data will be accessed in random order.

POSIX_FADV_NOREUSE
: The specified data will be accessed only once.

  Before Linux 2.6.18, **POSIX_FADV_NOREUSE** had the same semantics as **POSIX_FADV_WILLNEED**. This was probably a bug; since Linux 2.6.18, this flag is a no-op.

POSIX_FADV_WILLNEED
: The specified data will be accessed in the near future.

  POSIX_FADV_WILLNEED initiates a nonblocking read of the specified region into the page cache. The amount of data read may be decreased by the kernel depending on virtual memory load. (A few megabytes will usually be fully satisfied, and more is rarely useful.)

POSIX_FADV_DONTNEED
: The specified data will not be accessed in the near future.

  POSIX_FADV_DONTNEED attempts to free cached pages associated with the specified region. This is useful, for example, while streaming large files. A program may periodically request the kernel to free cached data that has already been used, so that more useful cached pages are not discarded instead.

  Requests to discard partial pages are ignored. It is preferable to preserve needed data than discard unneeded data. If the application requires that data be considered for discarding, then *offset* and *len* must be page-aligned.

  The implementation *may* attempt to write back dirty pages in the specified region, but this is not guaranteed. Any unwritten dirty pages will not be freed. If the application wishes to ensure that dirty pages will be released, it should call *fsync*(2) or *fdatasync*(2) first.

RETURN VALUE

On success, zero is returned. On error, an error number is returned.

ERRORS

EBADF
The *fd* argument was not a valid file descriptor.

EINVAL
An invalid value was specified for *advice*.

ESPIPE
The specified file descriptor refers to a pipe or FIFO. (**ESPIPE** is the error specified by POSIX, but before Linux 2.6.16, Linux returned **EINVAL** in this case.)

VERSIONS

Under Linux, **POSIX_FADV_NORMAL** sets the readahead window to the default size for the backing device; **POSIX_FADV_SEQUENTIAL** doubles this size, and **POSIX_FADV_RANDOM** disables file readahead entirely. These changes affect the entire file, not just the specified region (but other open file handles to the same file are unaffected).

C library/kernel differences

The name of the wrapper function in the C library is **posix_fadvise**(). The underlying system call is called **fadvise64**() (or, on some architectures, *fadvise64_64*()); the difference between the two is that the former system call assumes that the type of the *len* argument is *size_t*, while the latter expects *loff_t* there.

Architecture-specific variants

Some architectures require 64-bit arguments to be aligned in a suitable pair of registers (see *syscall*(2) for further detail). On such architectures, the call signature of **posix_fadvise**() shown in the SYNOPSIS would force a register to be wasted as padding between the *fd* and *offset* arguments. Therefore, these architectures define a version of the system call that orders the arguments suitably, but is otherwise exactly the same as **posix_fadvise**().

For example, since Linux 2.6.14, ARM has the following system call:

```
long arm_fadvise64_64(int fd, int advice,
                      loff_t offset, loff_t len);
```

These architecture-specific details are generally hidden from applications by the glibc **posix_fadvise**() wrapper function, which invokes the appropriate architecture-specific system call.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Kernel support first appeared in Linux 2.5.60; the underlying system call is called **fadvise64**(). Library support has been provided since glibc 2.2, via the wrapper function **posix_fadvise**().

Since Linux 3.18, support for the underlying system call is optional, depending on the setting of the **CONFIG_ADVISE_SYSCALLS** configuration option.

The type of the *len* argument was changed from *size_t* to *off_t* in POSIX.1-2001 TC1.

NOTES

The contents of the kernel buffer cache can be cleared via the */proc/sys/vm/drop_caches* interface described in *proc*(5).

One can obtain a snapshot of which pages of a file are resident in the buffer cache by opening a file, mapping it with *mmap*(2), and then applying *mincore*(2) to the mapping.

BUGS

Before Linux 2.6.6, if *len* was specified as 0, then this was interpreted literally as "zero bytes", rather than as meaning "all bytes through to the end of the file".

SEE ALSO

fincore(1), *mincore*(2), *readahead*(2), *sync_file_range*(2), *posix_fallocate*(3), *posix_madvise*(3)

NAME

prctl − operations on a process or thread

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/prctl.h>

int prctl(int *op***, ...**
/* **unsigned long** *arg2***, unsigned long** *arg3***,**
unsigned long *arg4***, unsigned long** *arg5* */ **);**

DESCRIPTION

prctl() manipulates various aspects of the behavior of the calling thread or process.

Note that careless use of some **prctl**() operations can confuse the user-space run-time environment, so these operations should be used with care.

prctl() is called with a first argument describing what to do (with values defined in *<linux/prctl.h>*), and further arguments with a significance depending on the first one. The first argument can be:

PR_CAP_AMBIENT (since Linux 4.3)
Reads or changes the ambient capability set of the calling thread, according to the value of *arg2*, which must be one of the following:

PR_CAP_AMBIENT_RAISE
The capability specified in *arg3* is added to the ambient set. The specified capability must already be present in both the permitted and the inheritable sets of the process. This operation is not permitted if the **SECBIT_NO_CAP_AMBIENT_RAISE** securebit is set.

PR_CAP_AMBIENT_LOWER
The capability specified in *arg3* is removed from the ambient set.

PR_CAP_AMBIENT_IS_SET
The **prctl**() call returns 1 if the capability in *arg3* is in the ambient set and 0 if it is not.

PR_CAP_AMBIENT_CLEAR_ALL
All capabilities will be removed from the ambient set. This operation requires setting *arg3* to zero.

In all of the above operations, *arg4* and *arg5* must be specified as 0.

Higher-level interfaces layered on top of the above operations are provided in the *libcap*(3) library in the form of *cap_get_ambient*(3), *cap_set_ambient*(3), and *cap_reset_ambient*(3)

PR_CAPBSET_READ (since Linux 2.6.25)
Return (as the function result) 1 if the capability specified in *arg2* is in the calling thread's capability bounding set, or 0 if it is not. (The capability constants are defined in *<linux/capability.h>*.) The capability bounding set dictates whether the process can receive the capability through a file's permitted capability set on a subsequent call to *execve*(2).

If the capability specified in *arg2* is not valid, then the call fails with the error **EINVAL**.

A higher-level interface layered on top of this operation is provided in the *libcap*(3) library in the form of *cap_get_bound*(3)

PR_CAPBSET_DROP (since Linux 2.6.25)
If the calling thread has the **CAP_SETPCAP** capability within its user namespace, then drop the capability specified by *arg2* from the calling thread's capability bounding set. Any children of the calling thread will inherit the newly reduced bounding set.

The call fails with the error: **EPERM** if the calling thread does not have the **CAP_SETPCAP**; **EINVAL** if *arg2* does not represent a valid capability; or **EINVAL** if file capabilities are not enabled in the kernel, in which case bounding sets are not supported.

A higher-level interface layered on top of this operation is provided in the *libcap*(3) library in the form of *cap_drop_bound*(3)

PR_SET_CHILD_SUBREAPER (since Linux 3.4)
If *arg2* is nonzero, set the "child subreaper" attribute of the calling process; if *arg2* is zero, unset the attribute.

A subreaper fulfills the role of *init*(1) for its descendant processes. When a process becomes orphaned (i.e., its immediate parent terminates), then that process will be reparented to the nearest still living ancestor subreaper. Subsequently, calls to *getppid*(2) in the orphaned process will now return the PID of the subreaper process, and when the orphan terminates, it is the subreaper process that will receive a **SIGCHLD** signal and will be able to *wait*(2) on the process to discover its termination status.

The setting of the "child subreaper" attribute is not inherited by children created by *fork*(2) and *clone*(2). The setting is preserved across *execve*(2).

Establishing a subreaper process is useful in session management frameworks where a hierarchical group of processes is managed by a subreaper process that needs to be informed when one of the processes—for example, a double-forked daemon—terminates (perhaps so that it can restart that process). Some *init*(1) frameworks (e.g., *systemd*(1)) employ a subreaper process for similar reasons.

PR_GET_CHILD_SUBREAPER (since Linux 3.4)
Return the "child subreaper" setting of the caller, in the location pointed to by *(int *) arg2*.

PR_SET_DUMPABLE (since Linux 2.3.20)
Set the state of the "dumpable" attribute, which determines whether core dumps are produced for the calling process upon delivery of a signal whose default behavior is to produce a core dump.

Up to and including Linux 2.6.12, *arg2* must be either 0 (**SUID_DUMP_DISABLE**, process is not dumpable) or 1 (**SUID_DUMP_USER**, process is dumpable). Between Linux 2.6.13 and Linux 2.6.17, the value 2 was also permitted, which caused any binary which normally would not be dumped to be dumped readable by root only; for security reasons, this feature has been removed. (See also the description of */proc/sys/fs/suid_dumpable* in *proc*(5).)

Normally, the "dumpable" attribute is set to 1. However, it is reset to the current value contained in the file */proc/sys/fs/suid_dumpable* (which by default has the value 0), in the following circumstances:

- The process's effective user or group ID is changed.
- The process's filesystem user or group ID is changed (see *credentials*(7)).
- The process executes (**execve**(2)) a set-user-ID or set-group-ID program, resulting in a change of either the effective user ID or the effective group ID.
- The process executes (**execve**(2)) a program that has file capabilities (see *capabilities*(7)), but only if the permitted capabilities gained exceed those already permitted for the process.

Processes that are not dumpable can not be attached via *ptrace*(2) **PTRACE_ATTACH**; see *ptrace*(2) for further details.

If a process is not dumpable, the ownership of files in the process's */proc/*pid directory is affected as described in *proc*(5).

PR_GET_DUMPABLE (since Linux 2.3.20)
Return (as the function result) the current state of the calling process's dumpable attribute.

PR_SET_ENDIAN (since Linux 2.6.18, PowerPC only)
Set the endian-ness of the calling process to the value given in *arg2*, which should be one of the following: **PR_ENDIAN_BIG**, **PR_ENDIAN_LITTLE**, or **PR_ENDIAN_PPC_LITTLE** (PowerPC pseudo little endian).

PR_GET_ENDIAN (since Linux 2.6.18, PowerPC only)
Return the endian-ness of the calling process, in the location pointed to by *(int *) arg2*.

PR_SET_FP_MODE (since Linux 4.0, only on MIPS)
On the MIPS architecture, user-space code can be built using an ABI which permits linking with code that has more restrictive floating-point (FP) requirements. For example, user-space

code may be built to target the O32 FPXX ABI and linked with code built for either one of the more restrictive FP32 or FP64 ABIs. When more restrictive code is linked in, the overall requirement for the process is to use the more restrictive floating-point mode.

Because the kernel has no means of knowing in advance which mode the process should be executed in, and because these restrictions can change over the lifetime of the process, the **PR_SET_FP_MODE** operation is provided to allow control of the floating-point mode from user space.

The *(unsigned int) arg2* argument is a bit mask describing the floating-point mode used:

PR_FP_MODE_FR
When this bit is *unset* (so called **FR=0** or **FR0** mode), the 32 floating-point registers are 32 bits wide, and 64-bit registers are represented as a pair of registers (even- and odd- numbered, with the even-numbered register containing the lower 32 bits, and the odd-numbered register containing the higher 32 bits).

When this bit is *set* (on supported hardware), the 32 floating-point registers are 64 bits wide (so called **FR=1** or **FR1** mode). Note that modern MIPS implementations (MIPS R6 and newer) support **FR=1** mode only.

Applications that use the O32 FP32 ABI can operate only when this bit is *unset* (**FR=0**; or they can be used with FRE enabled, see below). Applications that use the O32 FP64 ABI (and the O32 FP64A ABI, which exists to provide the ability to operate with existing FP32 code; see below) can operate only when this bit is *set* (**FR=1**). Applications that use the O32 FPXX ABI can operate with either **FR=0** or **FR=1**.

PR_FP_MODE_FRE
Enable emulation of 32-bit floating-point mode. When this mode is enabled, it emulates 32-bit floating-point operations by raising a reserved-instruction exception on every instruction that uses 32-bit formats and the kernel then handles the instruction in software. (The problem lies in the discrepancy of handling odd-numbered registers which are the high 32 bits of 64-bit registers with even numbers in **FR=0** mode and the lower 32-bit parts of odd-numbered 64-bit registers in **FR=1** mode.) Enabling this bit is necessary when code with the O32 FP32 ABI should operate with code with compatible the O32 FPXX or O32 FP64A ABIs (which require **FR=1** FPU mode) or when it is executed on newer hardware (MIPS R6 onwards) which lacks **FR=0** mode support when a binary with the FP32 ABI is used.

Note that this mode makes sense only when the FPU is in 64-bit mode (**FR=1**).

Note that the use of emulation inherently has a significant performance hit and should be avoided if possible.

In the N32/N64 ABI, 64-bit floating-point mode is always used, so FPU emulation is not required and the FPU always operates in **FR=1** mode.

This operation is mainly intended for use by the dynamic linker (**ld.so**(8)).

The arguments *arg3*, *arg4*, and *arg5* are ignored.

PR_GET_FP_MODE (since Linux 4.0, only on MIPS)
Return (as the function result) the current floating-point mode (see the description of **PR_SET_FP_MODE** for details).

On success, the call returns a bit mask which represents the current floating-point mode.

The arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

PR_SET_FPEMU (since Linux 2.4.18, 2.5.9, only on ia64)
Set floating-point emulation control bits to *arg2*. Pass **PR_FPEMU_NOPRINT** to silently emulate floating-point operation accesses, or **PR_FPEMU_SIGFPE** to not emulate floating-point operations and send **SIGFPE** instead.

PR_GET_FPEMU (since Linux 2.4.18, 2.5.9, only on ia64)
Return floating-point emulation control bits, in the location pointed to by *(int *) arg2*.

PR_SET_FPEXC (since Linux 2.4.21, 2.5.32, only on PowerPC)
Set floating-point exception mode to *arg2*. Pass **PR_FP_EXC_SW_ENABLE** to use FPEXC for FP exception enables, **PR_FP_EXC_DIV** for floating-point divide by zero, **PR_FP_EXC_OVF** for floating-point overflow, **PR_FP_EXC_UND** for floating-point underflow, **PR_FP_EXC_RES** for floating-point inexact result, **PR_FP_EXC_INV** for floating-point invalid operation, **PR_FP_EXC_DISABLED** for FP exceptions disabled, **PR_FP_EXC_NONRECOV** for async nonrecoverable exception mode, **PR_FP_EXC_ASYNC** for async recoverable exception mode, **PR_FP_EXC_PRECISE** for precise exception mode.

PR_GET_FPEXC (since Linux 2.4.21, 2.5.32, only on PowerPC)
Return floating-point exception mode, in the location pointed to by *(int *) arg2*.

PR_SET_IO_FLUSHER (since Linux 5.6)
If a user process is involved in the block layer or filesystem I/O path, and can allocate memory while processing I/O requests it must set *arg2* to 1. This will put the process in the IO_FLUSHER state, which allows it special treatment to make progress when allocating memory. If *arg2* is 0, the process will clear the IO_FLUSHER state, and the default behavior will be used.

The calling process must have the **CAP_SYS_RESOURCE** capability.

arg3, *arg4*, and *arg5* must be zero.

The IO_FLUSHER state is inherited by a child process created via *fork*(2) and is preserved across *execve*(2).

Examples of IO_FLUSHER applications are FUSE daemons, SCSI device emulation daemons, and daemons that perform error handling like multipath path recovery applications.

PR_GET_IO_FLUSHER (Since Linux 5.6)
Return (as the function result) the IO_FLUSHER state of the caller. A value of 1 indicates that the caller is in the IO_FLUSHER state; 0 indicates that the caller is not in the IO_FLUSHER state.

The calling process must have the **CAP_SYS_RESOURCE** capability.

arg2, *arg3*, *arg4*, and *arg5* must be zero.

PR_SET_KEEPCAPS (since Linux 2.2.18)
Set the state of the calling thread's "keep capabilities" flag. The effect of this flag is described in *capabilities*(7). *arg2* must be either 0 (clear the flag) or 1 (set the flag). The "keep capabilities" value will be reset to 0 on subsequent calls to *execve*(2).

PR_GET_KEEPCAPS (since Linux 2.2.18)
Return (as the function result) the current state of the calling thread's "keep capabilities" flag. See *capabilities*(7) for a description of this flag.

PR_MCE_KILL (since Linux 2.6.32)
Set the machine check memory corruption kill policy for the calling thread. If *arg2* is **PR_MCE_KILL_CLEAR**, clear the thread memory corruption kill policy and use the system-wide default. (The system-wide default is defined by */proc/sys/vm/memory_failure_early_kill*; see *proc*(5).) If *arg2* is **PR_MCE_KILL_SET**, use a thread-specific memory corruption kill policy. In this case, *arg3* defines whether the policy is *early kill* (**PR_MCE_KILL_EARLY**), *late kill* (**PR_MCE_KILL_LATE**), or the system-wide default (**PR_MCE_KILL_DEFAULT**). Early kill means that the thread receives a **SIGBUS** signal as soon as hardware memory corruption is detected inside its address space. In late kill mode, the process is killed only when it accesses a corrupted page. See *sigaction*(2) for more information on the **SIGBUS** signal. The policy is inherited by children. The remaining unused **prctl**() arguments must be zero for future compatibility.

PR_MCE_KILL_GET (since Linux 2.6.32)
Return (as the function result) the current per-process machine check kill policy. All unused **prctl**() arguments must be zero.

PR_SET_MM (since Linux 3.3)
Modify certain kernel memory map descriptor fields of the calling process. Usually these fields are set by the kernel and dynamic loader (see *ld.so*(8) for more information) and a regular application should not use this feature. However, there are cases, such as self-modifying programs, where a program might find it useful to change its own memory map.

The calling process must have the **CAP_SYS_RESOURCE** capability. The value in *arg2* is one of the options below, while *arg3* provides a new value for the option. The *arg4* and *arg5* arguments must be zero if unused.

Before Linux 3.10, this feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_MM_START_CODE
Set the address above which the program text can run. The corresponding memory area must be readable and executable, but not writable or shareable (see *mprotect*(2) and *mmap*(2) for more information).

PR_SET_MM_END_CODE
Set the address below which the program text can run. The corresponding memory area must be readable and executable, but not writable or shareable.

PR_SET_MM_START_DATA
Set the address above which initialized and uninitialized (bss) data are placed. The corresponding memory area must be readable and writable, but not executable or shareable.

PR_SET_MM_END_DATA
Set the address below which initialized and uninitialized (bss) data are placed. The corresponding memory area must be readable and writable, but not executable or shareable.

PR_SET_MM_START_STACK
Set the start address of the stack. The corresponding memory area must be readable and writable.

PR_SET_MM_START_BRK
Set the address above which the program heap can be expanded with *brk*(2) call. The address must be greater than the ending address of the current program data segment. In addition, the combined size of the resulting heap and the size of the data segment can't exceed the **RLIMIT_DATA** resource limit (see *setrlimit*(2)).

PR_SET_MM_BRK
Set the current *brk*(2) value. The requirements for the address are the same as for the **PR_SET_MM_START_BRK** option.

The following options are available since Linux 3.5.

PR_SET_MM_ARG_START
Set the address above which the program command line is placed.

PR_SET_MM_ARG_END
Set the address below which the program command line is placed.

PR_SET_MM_ENV_START
Set the address above which the program environment is placed.

PR_SET_MM_ENV_END
Set the address below which the program environment is placed.

The address passed with **PR_SET_MM_ARG_START**, **PR_SET_MM_ARG_END**, **PR_SET_MM_ENV_START**, and **PR_SET_MM_ENV_END** should belong to a process stack area. Thus, the corresponding memory area must be readable, writable, and (depending on the kernel configuration) have the **MAP_GROWSDOWN** attribute set (see *mmap*(2)).

PR_SET_MM_AUXV
Set a new auxiliary vector. The *arg3* argument should provide the address of the vector. The *arg4* is the size of the vector.

PR_SET_MM_EXE_FILE
Supersede the */proc/*pid*/exe* symbolic link with a new one pointing to a new executable file identified by the file descriptor provided in *arg3* argument. The file descriptor should be obtained with a regular *open*(2) call.

To change the symbolic link, one needs to unmap all existing executable memory areas, including those created by the kernel itself (for example the kernel usually creates at least one executable memory area for the ELF *.text* section).

In Linux 4.9 and earlier, the **PR_SET_MM_EXE_FILE** operation can be performed only once in a process's lifetime; attempting to perform the operation a second time results in the error **EPERM**. This restriction was enforced for security reasons that were subsequently deemed specious, and the restriction was removed in Linux 4.10 because some user-space applications needed to perform this operation more than once.

The following options are available since Linux 3.18.

PR_SET_MM_MAP
Provides one-shot access to all the addresses by passing in a *struct prctl_mm_map* (as defined in <*linux/prctl.h*>). The *arg4* argument should provide the size of the struct.

This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_MM_MAP_SIZE
Returns the size of the *struct prctl_mm_map* the kernel expects. This allows user space to find a compatible struct. The *arg4* argument should be a pointer to an unsigned int.

This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_VMA (since Linux 5.17)
Sets an attribute specified in *arg2* for virtual memory areas starting from the address specified in *arg3* and spanning the size specified in *arg4*. *arg5* specifies the value of the attribute to be set.

Note that assigning an attribute to a virtual memory area might prevent it from being merged with adjacent virtual memory areas due to the difference in that attribute's value.

Currently, *arg2* must be one of:

PR_SET_VMA_ANON_NAME
Set a name for anonymous virtual memory areas. *arg5* should be a pointer to a null-terminated string containing the name. The name length including null byte cannot exceed 80 bytes. If *arg5* is NULL, the name of the appropriate anonymous virtual memory areas will be reset. The name can contain only printable ascii characters (including space), except '[', ']', '\', '$', and '`'.

PR_MPX_ENABLE_MANAGEMENT
PR_MPX_DISABLE_MANAGEMENT (since Linux 3.19, removed in Linux 5.4; only on x86)
Enable or disable kernel management of Memory Protection eXtensions (MPX) bounds tables. The *arg2*, *arg3*, *arg4*, and *arg5* arguments must be zero.

MPX is a hardware-assisted mechanism for performing bounds checking on pointers. It consists of a set of registers storing bounds information and a set of special instruction prefixes that tell the CPU on which instructions it should do bounds enforcement. There is a limited number of these registers and when there are more pointers than registers, their contents must be "spilled" into a set of tables. These tables are called "bounds tables" and the MPX **prctl**() operations control whether the kernel manages their allocation and freeing.

When management is enabled, the kernel will take over allocation and freeing of the bounds tables. It does this by trapping the #BR exceptions that result at first use of missing bounds

tables and instead of delivering the exception to user space, it allocates the table and populates the bounds directory with the location of the new table. For freeing, the kernel checks to see if bounds tables are present for memory which is not allocated, and frees them if so.

Before enabling MPX management using **PR_MPX_ENABLE_MANAGEMENT**, the application must first have allocated a user-space buffer for the bounds directory and placed the location of that directory in the *bndcfgu* register.

These calls fail if the CPU or kernel does not support MPX. Kernel support for MPX is enabled via the **CONFIG_X86_INTEL_MPX** configuration option. You can check whether the CPU supports MPX by looking for the *mpx* CPUID bit, like with the following command:

```
cat /proc/cpuinfo | grep ' mpx '
```

A thread may not switch in or out of long (64-bit) mode while MPX is enabled.

All threads in a process are affected by these calls.

The child of a *fork*(2) inherits the state of MPX management. During *execve*(2), MPX management is reset to a state as if **PR_MPX_DISABLE_MANAGEMENT** had been called.

For further information on Intel MPX, see the kernel source file *Documentation/x86/intel_mpx.txt*.

Due to a lack of toolchain support, **PR_MPX_ENABLE_MANAGEMENT** and **PR_MPX_DISABLE_MANAGEMENT** are not supported in Linux 5.4 and later.

PR_SET_NAME (since Linux 2.6.9)
Set the name of the calling thread, using the value in the location pointed to by *(char *) arg2*. The name can be up to 16 bytes long, including the terminating null byte. (If the length of the string, including the terminating null byte, exceeds 16 bytes, the string is silently truncated.) This is the same attribute that can be set via *pthread_setname_np*(3) and retrieved using *pthread_getname_np*(3). The attribute is likewise accessible via */proc/self/task/*tid*/comm* (see *proc*(5)), where *tid* is the thread ID of the calling thread, as returned by *gettid*(2).

PR_GET_NAME (since Linux 2.6.11)
Return the name of the calling thread, in the buffer pointed to by *(char *) arg2*. The buffer should allow space for up to 16 bytes; the returned string will be null-terminated.

PR_SET_NO_NEW_PRIVS (since Linux 3.5)
Set the calling thread's *no_new_privs* attribute to the value in *arg2*. With *no_new_privs* set to 1, *execve*(2) promises not to grant privileges to do anything that could not have been done without the *execve*(2) call (for example, rendering the set-user-ID and set-group-ID mode bits, and file capabilities non-functional). Once set, the *no_new_privs* attribute cannot be unset. The setting of this attribute is inherited by children created by *fork*(2) and *clone*(2), and preserved across *execve*(2).

Since Linux 4.10, the value of a thread's *no_new_privs* attribute can be viewed via the *NoNewPrivs* field in the */proc/*pid*/status* file.

For more information, see the kernel source file *Documentation/user-space−api/no_new_privs.rst* (or *Documentation/prctl/no_new_privs.txt* before Linux 4.13). See also *seccomp*(2).

PR_GET_NO_NEW_PRIVS (since Linux 3.5)
Return (as the function result) the value of the *no_new_privs* attribute for the calling thread. A value of 0 indicates the regular *execve*(2) behavior. A value of 1 indicates *execve*(2) will operate in the privilege-restricting mode described above.

PR_PAC_RESET_KEYS (since Linux 5.0, only on arm64)
Securely reset the thread's pointer authentication keys to fresh random values generated by the kernel.

The set of keys to be reset is specified by *arg2*, which must be a logical OR of zero or more of the following:

PR_PAC_APIAKEY
instruction authentication key A

PR_PAC_APIBKEY
instruction authentication key B

PR_PAC_APDAKEY
data authentication key A

PR_PAC_APDBKEY
data authentication key B

PR_PAC_APGAKEY
generic authentication "A" key.

(Yes folks, there really is no generic B key.)

As a special case, if *arg2* is zero, then all the keys are reset. Since new keys could be added in future, this is the recommended way to completely wipe the existing keys when establishing a clean execution context. Note that there is no need to use **PR_PAC_RESET_KEYS** in preparation for calling *execve*(2), since *execve*(2) resets all the pointer authentication keys.

The remaining arguments *arg3*, *arg4*, and *arg5* must all be zero.

If the arguments are invalid, and in particular if *arg2* contains set bits that are unrecognized or that correspond to a key not available on this platform, then the call fails with error **EINVAL**.

Warning: Because the compiler or run-time environment may be using some or all of the keys, a successful **PR_PAC_RESET_KEYS** may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you know what you are doing.

For more information, see the kernel source file *Documentation/arm64/pointer−authentication.rst* (or *Documentation/arm64/pointer−authentication.txt* before Linux 5.3).

PR_SET_PDEATHSIG (since Linux 2.1.57)
Set the parent-death signal of the calling process to *arg2* (either a signal value in the range [**1**, *NSIG − 1*], or **0** to clear). This is the signal that the calling process will get when its parent dies.

Warning: the "parent" in this case is considered to be the *thread* that created this process. In other words, the signal will be sent when that thread terminates (via, for example, *pthread_exit*(3)), rather than after all of the threads in the parent process terminate.

The parent-death signal is sent upon subsequent termination of the parent thread and also upon termination of each subreaper process (see the description of **PR_SET_CHILD_SUBREAPER** above) to which the caller is subsequently reparented. If the parent thread and all ancestor subreapers have already terminated by the time of the **PR_SET_PDEATHSIG** operation, then no parent-death signal is sent to the caller.

The parent-death signal is process-directed (see *signal*(7)) and, if the child installs a handler using the *sigaction*(2) **SA_SIGINFO** flag, the *si_pid* field of the *siginfo_t* argument of the handler contains the PID of the terminating parent process.

The parent-death signal setting is cleared for the child of a *fork*(2). It is also (since Linux 2.4.36 / 2.6.23) cleared when executing a set-user-ID or set-group-ID binary, or a binary that has associated capabilities (see *capabilities*(7)); otherwise, this value is preserved across *execve*(2). The parent-death signal setting is also cleared upon changes to any of the following thread credentials: effective user ID, effective group ID, filesystem user ID, or filesystem group ID.

PR_GET_PDEATHSIG (since Linux 2.3.15)
Return the current value of the parent process death signal, in the location pointed to by *(int *) arg2*.

PR_SET_PTRACER (since Linux 3.4)
This is meaningful only when the Yama LSM is enabled and in mode 1 ("restricted ptrace", visible via */proc/sys/kernel/yama/ptrace_scope*). When a "ptracer process ID" is passed in *arg2*, the caller is declaring that the ptracer process can *ptrace*(2) the calling process as if it were a direct process ancestor. Each **PR_SET_PTRACER** operation replaces the previous "ptracer process ID". Employing **PR_SET_PTRACER** with *arg2* set to 0 clears the caller's

"ptracer process ID". If *arg2* is **PR_SET_PTRACER_ANY**, the ptrace restrictions introduced by Yama are effectively disabled for the calling process.

For further information, see the kernel source file *Documentation/admin−guide/LSM/Yama.rst* (or *Documentation/security/Yama.txt* before Linux 4.13).

PR_SET_SECCOMP (since Linux 2.6.23)
Set the secure computing (seccomp) mode for the calling thread, to limit the available system calls. The more recent *seccomp*(2) system call provides a superset of the functionality of **PR_SET_SECCOMP**, and is the preferred interface for new applications.

The seccomp mode is selected via *arg2*. (The seccomp constants are defined in <*linux/seccomp.h*>.) The following values can be specified:

SECCOMP_MODE_STRICT (since Linux 2.6.23)
See the description of **SECCOMP_SET_MODE_STRICT** in *seccomp*(2).

This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

SECCOMP_MODE_FILTER (since Linux 3.5)
The allowed system calls are defined by a pointer to a Berkeley Packet Filter passed in *arg3*. This argument is a pointer to *struct sock_fprog*; it can be designed to filter arbitrary system calls and system call arguments. See the description of **SECCOMP_SET_MODE_FILTER** in *seccomp*(2).

This operation is available only if the kernel is configured with **CONFIG_SECCOMP_FILTER** enabled.

For further details on seccomp filtering, see *seccomp*(2).

PR_GET_SECCOMP (since Linux 2.6.23)
Return (as the function result) the secure computing mode of the calling thread. If the caller is not in secure computing mode, this operation returns 0; if the caller is in strict secure computing mode, then the **prctl**() call will cause a **SIGKILL** signal to be sent to the process. If the caller is in filter mode, and this system call is allowed by the seccomp filters, it returns 2; otherwise, the process is killed with a **SIGKILL** signal.

This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

Since Linux 3.8, the *Seccomp* field of the */proc/*pid*/status* file provides a method of obtaining the same information, without the risk that the process is killed; see *proc*(5).

PR_SET_SECUREBITS (since Linux 2.6.26)
Set the "securebits" flags of the calling thread to the value supplied in *arg2*. See *capabilities*(7).

PR_GET_SECUREBITS (since Linux 2.6.26)
Return (as the function result) the "securebits" flags of the calling thread. See *capabilities*(7).

PR_GET_SPECULATION_CTRL (since Linux 4.17)
Return (as the function result) the state of the speculation misfeature specified in *arg2*. Currently, the only permitted value for this argument is **PR_SPEC_STORE_BYPASS** (otherwise the call fails with the error **ENODEV**).

The return value uses bits 0-3 with the following meaning:

PR_SPEC_PRCTL
Mitigation can be controlled per thread by **PR_SET_SPECULATION_CTRL**.

PR_SPEC_ENABLE
The speculation feature is enabled, mitigation is disabled.

PR_SPEC_DISABLE
The speculation feature is disabled, mitigation is enabled.

PR_SPEC_FORCE_DISABLE
Same as **PR_SPEC_DISABLE** but cannot be undone.

PR_SPEC_DISABLE_NOEXEC (since Linux 5.1)
Same as **PR_SPEC_DISABLE**, but the state will be cleared on *execve*(2).

If all bits are 0, then the CPU is not affected by the speculation misfeature.

If **PR_SPEC_PRCTL** is set, then per-thread control of the mitigation is available. If not set, **prctl**() for the speculation misfeature will fail.

The *arg3*, *arg4*, and *arg5* arguments must be specified as 0; otherwise the call fails with the error **EINVAL**.

PR_SET_SPECULATION_CTRL (since Linux 4.17)
Sets the state of the speculation misfeature specified in *arg2*. The speculation-misfeature settings are per-thread attributes.

Currently, *arg2* must be one of:

PR_SPEC_STORE_BYPASS
Set the state of the speculative store bypass misfeature.

PR_SPEC_INDIRECT_BRANCH (since Linux 4.20)
Set the state of the indirect branch speculation misfeature.

If *arg2* does not have one of the above values, then the call fails with the error **ENODEV**.

The *arg3* argument is used to hand in the control value, which is one of the following:

PR_SPEC_ENABLE
The speculation feature is enabled, mitigation is disabled.

PR_SPEC_DISABLE
The speculation feature is disabled, mitigation is enabled.

PR_SPEC_FORCE_DISABLE
Same as **PR_SPEC_DISABLE**, but cannot be undone. A subsequent **prctl**(*arg2*, **PR_SPEC_ENABLE**) with the same value for *arg2* will fail with the error **EPERM**.

PR_SPEC_DISABLE_NOEXEC (since Linux 5.1)
Same as **PR_SPEC_DISABLE**, but the state will be cleared on *execve*(2). Currently only supported for *arg2* equal to **PR_SPEC_STORE_BYPASS.**

Any unsupported value in *arg3* will result in the call failing with the error **ERANGE**.

The *arg4* and *arg5* arguments must be specified as 0; otherwise the call fails with the error **EINVAL**.

The speculation feature can also be controlled by the **spec_store_bypass_disable** boot parameter. This parameter may enforce a read-only policy which will result in the **prctl**() call failing with the error **ENXIO**. For further details, see the kernel source file *Documentation/admin−guide/kernel−parameters.txt*.

PR_SVE_SET_VL (since Linux 4.15, only on arm64)
Configure the thread's SVE vector length, as specified by *(int) arg2*. Arguments *arg3*, *arg4*, and *arg5* are ignored.

The bits of *arg2* corresponding to **PR_SVE_VL_LEN_MASK** must be set to the desired vector length in bytes. This is interpreted as an upper bound: the kernel will select the greatest available vector length that does not exceed the value specified. In particular, specifying **SVE_VL_MAX** (defined in *<asm/sigcontext.h>)* for the **PR_SVE_VL_LEN_MASK** bits requests the maximum supported vector length.

In addition, the other bits of *arg2* must be set to one of the following combinations of flags:

0
Perform the change immediately. At the next *execve*(2) in the thread, the vector length will be reset to the value configured in */proc/sys/abi/sve_default_vector_length*.

PR_SVE_VL_INHERIT
Perform the change immediately. Subsequent *execve*(2) calls will preserve the new vector length.

PR_SVE_SET_VL_ONEXEC
Defer the change, so that it is performed at the next *execve*(2) in the thread. Further *execve*(2) calls will reset the vector length to the value configured in */proc/sys/abi/sve_default_vector_length*.

PR_SVE_SET_VL_ONEXEC | PR_SVE_VL_INHERIT
Defer the change, so that it is performed at the next *execve*(2) in the thread. Further *execve*(2) calls will preserve the new vector length.

In all cases, any previously pending deferred change is canceled.

The call fails with error **EINVAL** if SVE is not supported on the platform, if *arg2* is unrecognized or invalid, or the value in the bits of *arg2* corresponding to **PR_SVE_VL_LEN_MASK** is outside the range **SVE_VL_MIN**..**SVE_VL_MAX** or is not a multiple of 16.

On success, a nonnegative value is returned that describes the *selected* configuration. If **PR_SVE_SET_VL_ONEXEC** was included in *arg2*, then the configuration described by the return value will take effect at the next *execve*(2). Otherwise, the configuration is already in effect when the **PR_SVE_SET_VL** call returns. In either case, the value is encoded in the same way as the return value of **PR_SVE_GET_VL**. Note that there is no explicit flag in the return value corresponding to **PR_SVE_SET_VL_ONEXEC**.

The configuration (including any pending deferred change) is inherited across *fork*(2) and *clone*(2).

For more information, see the kernel source file *Documentation/arm64/sve.rst* (or *Documentation/arm64/sve.txt* before Linux 5.3).

Warning: Because the compiler or run-time environment may be using SVE, using this call without the **PR_SVE_SET_VL_ONEXEC** flag may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you really know what you are doing.

PR_SVE_GET_VL (since Linux 4.15, only on arm64)
Get the thread's current SVE vector length configuration.

Arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

Provided that the kernel and platform support SVE, this operation always succeeds, returning a nonnegative value that describes the *current* configuration. The bits corresponding to **PR_SVE_VL_LEN_MASK** contain the currently configured vector length in bytes. The bit corresponding to **PR_SVE_VL_INHERIT** indicates whether the vector length will be inherited across *execve*(2).

Note that there is no way to determine whether there is a pending vector length change that has not yet taken effect.

For more information, see the kernel source file *Documentation/arm64/sve.rst* (or *Documentation/arm64/sve.txt* before Linux 5.3).

PR_SET_SYSCALL_USER_DISPATCH (since Linux 5.11, x86 only)
Configure the Syscall User Dispatch mechanism for the calling thread. This mechanism allows an application to selectively intercept system calls so that they can be handled within the application itself. Interception takes the form of a thread-directed **SIGSYS** signal that is delivered to the thread when it makes a system call. If intercepted, the system call is not executed by the kernel.

To enable this mechanism, *arg2* should be set to **PR_SYS_DISPATCH_ON**. Once enabled, further system calls will be selectively intercepted, depending on a control variable provided by user space. In this case, *arg3* and *arg4* respectively identify the *offset* and *length* of a single contiguous memory region in the process address space from where system calls are always allowed to be executed, regardless of the control variable. (Typically, this area would include the area of memory containing the C library.)

arg5 points to a char-sized variable that is a fast switch to allow/block system call execution without the overhead of doing another system call to re-configure Syscall User Dispatch. This control variable can either be set to **SYSCALL_DISPATCH_FILTER_BLOCK** to block system calls from executing or to **SYSCALL_DISPATCH_FILTER_ALLOW** to

temporarily allow them to be executed. This value is checked by the kernel on every system call entry, and any unexpected value will raise an uncatchable **SIGSYS** at that time, killing the application.

When a system call is intercepted, the kernel sends a thread-directed **SIGSYS** signal to the triggering thread. Various fields will be set in the *siginfo_t* structure (see *sigaction*(2)) associated with the signal:

- *si_signo* will contain **SIGSYS**.
- *si_call_addr* will show the address of the system call instruction.
- *si_syscall* and *si_arch* will indicate which system call was attempted.
- *si_code* will contain **SYS_USER_DISPATCH**.
- *si_errno* will be set to 0.

The program counter will be as though the system call happened (i.e., the program counter will not point to the system call instruction).

When the signal handler returns to the kernel, the system call completes immediately and returns to the calling thread, without actually being executed. If necessary (i.e., when emulating the system call on user space.), the signal handler should set the system call return value to a sane value, by modifying the register context stored in the *ucontext* argument of the signal handler. See *sigaction*(2), *sigreturn*(2), and *getcontext*(3) for more information.

If *arg2* is set to **PR_SYS_DISPATCH_OFF**, Syscall User Dispatch is disabled for that thread. the remaining arguments must be set to 0.

The setting is not preserved across *fork*(2), *clone*(2), or *execve*(2).

For more information, see the kernel source file *Documentation/admin−guide/syscall−user−dispatch.rst*

PR_SET_TAGGED_ADDR_CTRL (since Linux 5.4, only on arm64)
Controls support for passing tagged user-space addresses to the kernel (i.e., addresses where bits 56—63 are not all zero).

The level of support is selected by *arg2*, which can be one of the following:

0 Addresses that are passed for the purpose of being dereferenced by the kernel must be untagged.

PR_TAGGED_ADDR_ENABLE
Addresses that are passed for the purpose of being dereferenced by the kernel may be tagged, with the exceptions summarized below.

The remaining arguments *arg3*, *arg4*, and *arg5* must all be zero.

On success, the mode specified in *arg2* is set for the calling thread and the return value is 0. If the arguments are invalid, the mode specified in *arg2* is unrecognized, or if this feature is unsupported by the kernel or disabled via */proc/sys/abi/tagged_addr_disabled*, the call fails with the error **EINVAL**.

In particular, if **prctl**(**PR_SET_TAGGED_ADDR_CTRL**, 0, 0, 0, 0) fails with **EINVAL**, then all addresses passed to the kernel must be untagged.

Irrespective of which mode is set, addresses passed to certain interfaces must always be untagged:

- *brk*(2), *mmap*(2), *shmat*(2), *shmdt*(2), and the *new_address* argument of *mremap*(2).

  (Prior to Linux 5.6 these accepted tagged addresses, but the behaviour may not be what you expect. Don't rely on it.)
- 'polymorphic' interfaces that accept pointers to arbitrary types cast to a *void* * or other generic type, specifically **prctl**(), *ioctl*(2), and in general *setsockopt*(2) (only certain specific *setsockopt*(2) options allow tagged addresses).

This list of exclusions may shrink when moving from one kernel version to a later kernel version. While the kernel may make some guarantees for backwards compatibility reasons, for

the purposes of new software the effect of passing tagged addresses to these interfaces is unspecified.

The mode set by this call is inherited across *fork*(2) and *clone*(2). The mode is reset by *execve*(2) to 0 (i.e., tagged addresses not permitted in the user/kernel ABI).

For more information, see the kernel source file *Documentation/arm64/tagged−address−abi.rst*.

Warning: This call is primarily intended for use by the run-time environment. A successful **PR_SET_TAGGED_ADDR_CTRL** call elsewhere may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you know what you are doing.

PR_GET_TAGGED_ADDR_CTRL (since Linux 5.4, only on arm64)
Returns the current tagged address mode for the calling thread.

Arguments *arg2*, *arg3*, *arg4*, and *arg5* must all be zero.

If the arguments are invalid or this feature is disabled or unsupported by the kernel, the call fails with **EINVAL**. In particular, if **prctl**(**PR_GET_TAGGED_ADDR_CTRL**, 0, 0, 0, 0) fails with **EINVAL**, then this feature is definitely either unsupported, or disabled via */proc/sys/abi/tagged_addr_disabled*. In this case, all addresses passed to the kernel must be untagged.

Otherwise, the call returns a nonnegative value describing the current tagged address mode, encoded in the same way as the *arg2* argument of **PR_SET_TAGGED_ADDR_CTRL**.

For more information, see the kernel source file *Documentation/arm64/tagged−address−abi.rst*.

PR_TASK_PERF_EVENTS_DISABLE (since Linux 2.6.31)
Disable all performance counters attached to the calling process, regardless of whether the counters were created by this process or another process. Performance counters created by the calling process for other processes are unaffected. For more information on performance counters, see the Linux kernel source file *tools/perf/design.txt*.

Originally called **PR_TASK_PERF_COUNTERS_DISABLE**; renamed (retaining the same numerical value) in Linux 2.6.32.

PR_TASK_PERF_EVENTS_ENABLE (since Linux 2.6.31)
The converse of **PR_TASK_PERF_EVENTS_DISABLE**; enable performance counters attached to the calling process.

Originally called **PR_TASK_PERF_COUNTERS_ENABLE**; renamed in Linux 2.6.32.

PR_SET_THP_DISABLE (since Linux 3.15)
Set the state of the "THP disable" flag for the calling thread. If *arg2* has a nonzero value, the flag is set, otherwise it is cleared. Setting this flag provides a method for disabling transparent huge pages for jobs where the code cannot be modified, and using a malloc hook with *madvise*(2) is not an option (i.e., statically allocated data). The setting of the "THP disable" flag is inherited by a child created via *fork*(2) and is preserved across *execve*(2).

PR_GET_THP_DISABLE (since Linux 3.15)
Return (as the function result) the current setting of the "THP disable" flag for the calling thread: either 1, if the flag is set, or 0, if it is not.

PR_GET_TID_ADDRESS (since Linux 3.5)
Return the *clear_child_tid* address set by *set_tid_address*(2) and the *clone*(2) **CLONE_CHILD_CLEARTID** flag, in the location pointed to by *(int **) arg2*. This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled. Note that since the **prctl**() system call does not have a compat implementation for the AMD64 x32 and MIPS n32 ABIs, and the kernel writes out a pointer using the kernel's pointer size, this operation expects a user-space buffer of 8 (not 4) bytes on these ABIs.

PR_SET_TIMERSLACK (since Linux 2.6.28)
Each thread has two associated timer slack values: a "default" value, and a "current" value. This operation sets the "current" timer slack value for the calling thread. *arg2* is an unsigned

long value, then maximum "current" value is ULONG_MAX and the minimum "current" value is 1. If the nanosecond value supplied in *arg2* is greater than zero, then the "current" value is set to this value. If *arg2* is equal to zero, the "current" timer slack is reset to the thread's "default" timer slack value.

The "current" timer slack is used by the kernel to group timer expirations for the calling thread that are close to one another; as a consequence, timer expirations for the thread may be up to the specified number of nanoseconds late (but will never expire early). Grouping timer expirations can help reduce system power consumption by minimizing CPU wake-ups.

The timer expirations affected by timer slack are those set by *select*(2), *pselect*(2), *poll*(2), *ppoll*(2), *epoll_wait*(2), *epoll_pwait*(2), *clock_nanosleep*(2), *nanosleep*(2), and *futex*(2) (and thus the library functions implemented via futexes, including *pthread_cond_timedwait*(3), *pthread_mutex_timedlock*(3), *pthread_rwlock_timedrdlock*(3), *pthread_rwlock_timedwrlock*(3), and *sem_timedwait*(3)).

Timer slack is not applied to threads that are scheduled under a real-time scheduling policy (see *sched_setscheduler*(2)).

When a new thread is created, the two timer slack values are made the same as the "current" value of the creating thread. Thereafter, a thread can adjust its "current" timer slack value via **PR_SET_TIMERSLACK**. The "default" value can't be changed. The timer slack values of *init* (PID 1), the ancestor of all processes, are 50,000 nanoseconds (50 microseconds). The timer slack value is inherited by a child created via *fork*(2), and is preserved across *execve*(2).

Since Linux 4.6, the "current" timer slack value of any process can be examined and changed via the file */proc/*pid*/timerslack_ns*. See *proc*(5).

PR_GET_TIMERSLACK (since Linux 2.6.28)
Return (as the function result) the "current" timer slack value of the calling thread.

PR_SET_TIMING (since Linux 2.6.0)
Set whether to use (normal, traditional) statistical process timing or accurate timestamp-based process timing, by passing **PR_TIMING_STATISTICAL** or **PR_TIMING_TIMESTAMP** to *arg2*. **PR_TIMING_TIMESTAMP** is not currently implemented (attempting to set this mode will yield the error **EINVAL**).

PR_GET_TIMING (since Linux 2.6.0)
Return (as the function result) which process timing method is currently in use.

PR_SET_TSC (since Linux 2.6.26, x86 only)
Set the state of the flag determining whether the timestamp counter can be read by the process. Pass **PR_TSC_ENABLE** to *arg2* to allow it to be read, or **PR_TSC_SIGSEGV** to generate a **SIGSEGV** when the process tries to read the timestamp counter.

PR_GET_TSC (since Linux 2.6.26, x86 only)
Return the state of the flag determining whether the timestamp counter can be read, in the location pointed to by *(int *) arg2*.

PR_SET_UNALIGN
(Only on: ia64, since Linux 2.3.48; parisc, since Linux 2.6.15; PowerPC, since Linux 2.6.18; Alpha, since Linux 2.6.22; sh, since Linux 2.6.34; tile, since Linux 3.12) Set unaligned access control bits to *arg2*. Pass **PR_UNALIGN_NOPRINT** to silently fix up unaligned user accesses, or **PR_UNALIGN_SIGBUS** to generate **SIGBUS** on unaligned user access. Alpha also supports an additional flag with the value of 4 and no corresponding named constant, which instructs kernel to not fix up unaligned accesses (it is analogous to providing the **UAC_NOFIX** flag in **SSI_NVPAIRS** operation of the **setsysinfo**() system call on Tru64).

PR_GET_UNALIGN
(See **PR_SET_UNALIGN** for information on versions and architectures.) Return unaligned access control bits, in the location pointed to by *(unsigned int *) arg2*.

PR_GET_AUXV (since Linux 6.4)
Get the auxiliary vector (auxv) into the buffer pointed to by *(void *) arg2*, whose length is given by *arg3*. If the buffer is not long enough for the full auxiliary vector, the copy will be truncated. Return (as the function result) the full length of the auxiliary vector. *arg4* and *arg5*

must be 0.

PR_SET_MDWE (since Linux 6.3)
Set the calling process' Memory-Deny-Write-Execute protection mask. Once protection bits are set, they can not be changed. *arg2* must be a bit mask of:

PR_MDWE_REFUSE_EXEC_GAIN
New memory mapping protections can't be writable and executable. Non-executable mappings can't become executable.

PR_MDWE_NO_INHERIT (since Linux 6.6)
Do not propagate MDWE protection to child processes on *fork*(2). Setting this bit requires setting **PR_MDWE_REFUSE_EXEC_GAIN** too.

PR_GET_MDWE (since Linux 6.3)
Return (as the function result) the Memory-Deny-Write-Execute protection mask of the calling process. (See **PR_SET_MDWE** for information on the protection mask bits.)

RETURN VALUE

On success, **PR_CAP_AMBIENT**+**PR_CAP_AMBIENT_IS_SET**, **PR_CAPBSET_READ**, **PR_GET_DUMPABLE**, **PR_GET_FP_MODE**, **PR_GET_IO_FLUSHER**, **PR_GET_KEEPCAPS**, **PR_MCE_KILL_GET**, **PR_GET_NO_NEW_PRIVS**, **PR_GET_SECUREBITS**, **PR_GET_SPECULATION_CTRL**, **PR_SVE_GET_VL**, **PR_SVE_SET_VL**, **PR_GET_TAGGED_ADDR_CTRL**, **PR_GET_THP_DISABLE**, **PR_GET_TIMING**, **PR_GET_TIMERSLACK**, **PR_GET_AUXV**, and (if it returns) **PR_GET_SECCOMP** return the nonnegative values described above. All other *op* values return 0 on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
op is **PR_SET_SECCOMP** and *arg2* is **SECCOMP_MODE_FILTER**, but the process does not have the **CAP_SYS_ADMIN** capability or has not set the *no_new_privs* attribute (see the discussion of **PR_SET_NO_NEW_PRIVS** above).

EACCES
op is **PR_SET_MM**, and *arg3* is **PR_SET_MM_EXE_FILE**, the file is not executable.

EBADF
op is **PR_SET_MM**, *arg3* is **PR_SET_MM_EXE_FILE**, and the file descriptor passed in *arg4* is not valid.

EBUSY
op is **PR_SET_MM**, *arg3* is **PR_SET_MM_EXE_FILE**, and this the second attempt to change the */proc/*pid*/exe* symbolic link, which is prohibited.

EFAULT
arg2 is an invalid address.

EFAULT
op is **PR_SET_SECCOMP**, *arg2* is **SECCOMP_MODE_FILTER**, the system was built with **CONFIG_SECCOMP_FILTER**, and *arg3* is an invalid address.

EFAULT
op is **PR_SET_SYSCALL_USER_DISPATCH** and *arg5* has an invalid address.

EINVAL
The value of *op* is not recognized, or not supported on this system.

EINVAL
op is **PR_MCE_KILL** or **PR_MCE_KILL_GET** or **PR_SET_MM**, and unused **prctl**() arguments were not specified as zero.

EINVAL
arg2 is not valid value for this *op*.

EINVAL
op is **PR_SET_SECCOMP** or **PR_GET_SECCOMP**, and the kernel was not configured with **CONFIG_SECCOMP**.

EINVAL
op is **PR_SET_SECCOMP**, *arg2* is **SECCOMP_MODE_FILTER**, and the kernel was not configured with **CONFIG_SECCOMP_FILTER**.

EINVAL
op is **PR_SET_MM**, and one of the following is true

- *arg4* or *arg5* is nonzero;
- *arg3* is greater than **TASK_SIZE** (the limit on the size of the user address space for this architecture);
- *arg2* is **PR_SET_MM_START_CODE**, **PR_SET_MM_END_CODE**, **PR_SET_MM_START_DATA**, **PR_SET_MM_END_DATA**, or **PR_SET_MM_START_STACK**, and the permissions of the corresponding memory area are not as required;
- *arg2* is **PR_SET_MM_START_BRK** or **PR_SET_MM_BRK**, and *arg3* is less than or equal to the end of the data segment or specifies a value that would cause the **RLIMIT_DATA** resource limit to be exceeded.

EINVAL
op is **PR_SET_PTRACER** and *arg2* is not 0, **PR_SET_PTRACER_ANY**, or the PID of an existing process.

EINVAL
op is **PR_SET_PDEATHSIG** and *arg2* is not a valid signal number.

EINVAL
op is **PR_SET_DUMPABLE** and *arg2* is neither **SUID_DUMP_DISABLE** nor **SUID_DUMP_USER**.

EINVAL
op is **PR_SET_TIMING** and *arg2* is not **PR_TIMING_STATISTICAL**.

EINVAL
op is **PR_SET_NO_NEW_PRIVS** and *arg2* is not equal to 1 or *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
op is **PR_GET_NO_NEW_PRIVS** and *arg2*, *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
op is **PR_SET_THP_DISABLE** and *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
op is **PR_GET_THP_DISABLE** and *arg2*, *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
op is **PR_CAP_AMBIENT** and an unused argument (*arg4*, *arg5*, or, in the case of **PR_CAP_AMBIENT_CLEAR_ALL**, *arg3*) is nonzero; or *arg2* has an invalid value; or *arg2* is **PR_CAP_AMBIENT_LOWER**, **PR_CAP_AMBIENT_RAISE**, or **PR_CAP_AMBIENT_IS_SET** and *arg3* does not specify a valid capability.

EINVAL
op was **PR_GET_SPECULATION_CTRL** or **PR_SET_SPECULATION_CTRL** and unused arguments to **prctl**() are not 0.

EINVAL
op is **PR_PAC_RESET_KEYS** and the arguments are invalid or unsupported. See the description of **PR_PAC_RESET_KEYS** above for details.

EINVAL
op is **PR_SVE_SET_VL** and the arguments are invalid or unsupported, or SVE is not available on this platform. See the description of **PR_SVE_SET_VL** above for details.

EINVAL
op is **PR_SVE_GET_VL** and SVE is not available on this platform.

EINVAL
op is **PR_SET_SYSCALL_USER_DISPATCH** and one of the following is true:

- *arg2* is **PR_SYS_DISPATCH_OFF** and the remaining arguments are not 0;
- *arg2* is **PR_SYS_DISPATCH_ON** and the memory range specified is outside the address space of the process.
- *arg2* is invalid.

EINVAL
op is **PR_SET_TAGGED_ADDR_CTRL** and the arguments are invalid or unsupported. See the description of **PR_SET_TAGGED_ADDR_CTRL** above for details.

EINVAL
op is **PR_GET_TAGGED_ADDR_CTRL** and the arguments are invalid or unsupported. See the description of **PR_GET_TAGGED_ADDR_CTRL** above for details.

ENODEV
op was **PR_SET_SPECULATION_CTRL** the kernel or CPU does not support the requested speculation misfeature.

ENXIO
op was **PR_MPX_ENABLE_MANAGEMENT** or **PR_MPX_DISABLE_MANAGEMENT** and the kernel or the CPU does not support MPX management. Check that the kernel and processor have MPX support.

ENXIO
op was **PR_SET_SPECULATION_CTRL** implies that the control of the selected speculation misfeature is not possible. See **PR_GET_SPECULATION_CTRL** for the bit fields to determine which option is available.

EOPNOTSUPP
op is **PR_SET_FP_MODE** and *arg2* has an invalid or unsupported value.

EPERM
op is **PR_SET_SECUREBITS**, and the caller does not have the **CAP_SETPCAP** capability, or tried to unset a "locked" flag, or tried to set a flag whose corresponding locked flag was set (see *capabilities*(7)).

EPERM
op is **PR_SET_SPECULATION_CTRL** wherein the speculation was disabled with **PR_SPEC_FORCE_DISABLE** and caller tried to enable it again.

EPERM
op is **PR_SET_KEEPCAPS**, and the caller's **SECBIT_KEEP_CAPS_LOCKED** flag is set (see *capabilities*(7)).

EPERM
op is **PR_CAPBSET_DROP**, and the caller does not have the **CAP_SETPCAP** capability.

EPERM
op is **PR_SET_MM**, and the caller does not have the **CAP_SYS_RESOURCE** capability.

EPERM
op is **PR_CAP_AMBIENT** and *arg2* is **PR_CAP_AMBIENT_RAISE**, but either the capability specified in *arg3* is not present in the process's permitted and inheritable capability sets, or the **PR_CAP_AMBIENT_LOWER** securebit has been set.

ERANGE
op was **PR_SET_SPECULATION_CTRL** and *arg3* is not **PR_SPEC_ENABLE**, **PR_SPEC_DISABLE**, **PR_SPEC_FORCE_DISABLE**, nor **PR_SPEC_DISABLE_NOEXEC**.

VERSIONS

IRIX has a **prctl**() system call (also introduced in Linux 2.1.44 as irix_prctl on the MIPS architecture), with prototype

```
ptrdiff_t prctl(int op, int arg2, int arg3);
```

and operations to get the maximum number of processes per user, get the maximum number of processors the calling process can use, find out whether a specified process is currently blocked, get or set the maximum stack size, and so on.

STANDARDS

Linux.

HISTORY

Linux 2.1.57, glibc 2.0.6

SEE ALSO

signal(2), *core*(5)

NAME

pread, pwrite − read from or write to a file descriptor at a given offset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

ssize_t pread(int fd, void buf[.count], size_t count,
              off_t offset);
ssize_t pwrite(int fd, const void buf[.count], size_t count,
               off_t offset);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pread(), **pwrite**():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION

pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*. The file offset is not changed.

pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset*. The file offset is not changed.

The file referenced by *fd* must be capable of seeking.

RETURN VALUE

On success, **pread**() returns the number of bytes read (a return of zero indicates end of file) and **pwrite**() returns the number of bytes written.

Note that it is not an error for a successful call to transfer fewer bytes than requested (see *read*(2) and *write*(2)).

On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

pread() can fail and set *errno* to any error specified for *read*(2) or *lseek*(2). **pwrite**() can fail and set *errno* to any error specified for *write*(2) or *lseek*(2).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Added in Linux 2.1.60; the entries in the i386 system call table were added in Linux 2.1.69. C library support (including emulation using *lseek*(2) on older kernels without the system calls) was added in glibc 2.1.

C library/kernel differences

On Linux, the underlying system calls were renamed in Linux 2.6: **pread**() became **pread64**(), and **pwrite**() became **pwrite64**(). The system call numbers remained the same. The glibc **pread**() and **pwrite**() wrapper functions transparently deal with the change.

On some 32-bit architectures, the calling signature for these system calls differ, for the reasons described in *syscall*(2).

NOTES

The **pread**() and **pwrite**() system calls are especially useful in multithreaded applications. They allow multiple threads to perform I/O on the same file descriptor without being affected by changes to the file offset by other threads.

BUGS

POSIX requires that opening a file with the **O_APPEND** flag should have no effect on the location at which **pwrite**() writes data. However, on Linux, if a file is opened with **O_APPEND**, **pwrite**() appends data to the end of the file, regardless of the value of *offset*.

SEE ALSO

lseek(2), *read*(2), *readv*(2), *write*(2)

NAME

process_madvise − give advice about use of memory to a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

ssize_t process_madvise(int *pidfd***, const struct iovec** *iovec***[.***n***],**
size_t *n***, int** *advice***, unsigned int** *flags***);**

DESCRIPTION

The **process_madvise**() system call is used to give advice or directions to the kernel about the address ranges of another process or of the calling process. It provides the advice for the address ranges described by *iovec* and *n*. The goal of such advice is to improve system or application performance.

The *pidfd* argument is a PID file descriptor (see *pidfd_open*(2)) that specifies the process to which the advice is to be applied.

The pointer *iovec* points to an array of *iovec* structures, described in *iovec*(3type).

n specifies the number of elements in the array of *iovec* structures. This value must be less than or equal to **IOV_MAX** (defined in <*limits.h*> or accessible via the call *sysconf(_SC_IOV_MAX)*).

The *advice* argument is one of the following values:

MADV_COLD
See *madvise*(2).

MADV_COLLAPSE
See *madvise*(2).

MADV_PAGEOUT
See *madvise*(2).

MADV_WILLNEED
See *madvise*(2).

The *flags* argument is reserved for future use; currently, this argument must be specified as 0.

The *n* and *iovec* arguments are checked before applying any advice. If *n* is too big, or *iovec* is invalid, then an error will be returned immediately and no advice will be applied.

The advice might be applied to only a part of *iovec* if one of its elements points to an invalid memory region in the remote process. No further elements will be processed beyond that point. (See the discussion regarding partial advice in RETURN VALUE.)

Starting in Linux 5.12, permission to apply advice to another process is governed by ptrace access mode **PTRACE_MODE_READ_FSCREDS** check (see *ptrace*(2)); in addition, because of the performance implications of applying the advice, the caller must have the **CAP_SYS_NICE** capability (see *capabilities*(7)).

RETURN VALUE

On success, **process_madvise**() returns the number of bytes advised. This return value may be less than the total number of requested bytes, if an error occurred after some *iovec* elements were already processed. The caller should check the return value to determine whether a partial advice occurred.

On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
pidfd is not a valid PID file descriptor.

EFAULT
The memory described by *iovec* is outside the accessible address space of the process referred to by *pidfd*.

EINVAL
flags is not 0.

EINVAL
The sum of the *iov_len* values of *iovec* overflows a *ssize_t* value.

EINVAL
n is too large.

ENOMEM
Could not allocate memory for internal copies of the *iovec* structures.

EPERM
The caller does not have permission to access the address space of the process *pidfd*.

ESRCH
The target process does not exist (i.e., it has terminated and been waited on).

See *madvise*(2) for *advice*-specific errors.

STANDARDS

Linux.

HISTORY

Linux 5.10. glibc 2.36.

Support for this system call is optional, depending on the setting of the **CONFIG_ADVISE_SYSCALLS** configuration option.

When this system call first appeared in Linux 5.10, permission to apply advice to another process was entirely governed by ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check (see *ptrace*(2)). This requirement was relaxed in Linux 5.12 so that the caller didn't require full control over the target process.

SEE ALSO

madvise(2), *pidfd_open*(2), *process_vm_readv*(2), *process_vm_write*(2)

NAME

process_vm_readv, process_vm_writev − transfer data between process address spaces

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/uio.h>

ssize_t process_vm_readv(pid_t pid,
                         const struct iovec *local_iov,
                         unsigned long liovcnt,
                         const struct iovec *remote_iov,
                         unsigned long riovcnt,
                         unsigned long flags);
ssize_t process_vm_writev(pid_t pid,
                         const struct iovec *local_iov,
                         unsigned long liovcnt,
                         const struct iovec *remote_iov,
                         unsigned long riovcnt,
                         unsigned long flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

process_vm_readv(), **process_vm_writev**():
_GNU_SOURCE

DESCRIPTION

These system calls transfer data between the address space of the calling process ("the local process") and the process identified by *pid* ("the remote process"). The data moves directly between the address spaces of the two processes, without passing through kernel space.

The **process_vm_readv**() system call transfers data from the remote process to the local process. The data to be transferred is identified by *remote_iov* and *riovcnt*: *remote_iov* is a pointer to an array describing address ranges in the process *pid*, and *riovcnt* specifies the number of elements in *remote_iov*. The data is transferred to the locations specified by *local_iov* and *liovcnt*: *local_iov* is a pointer to an array describing address ranges in the calling process, and *liovcnt* specifies the number of elements in *local_iov*.

The **process_vm_writev**() system call is the converse of **process_vm_readv**()—it transfers data from the local process to the remote process. Other than the direction of the transfer, the arguments *liovcnt*, *local_iov*, *riovcnt*, and *remote_iov* have the same meaning as for **process_vm_readv**().

The *local_iov* and *remote_iov* arguments point to an array of *iovec* structures, described in *iovec*(3type).

Buffers are processed in array order. This means that **process_vm_readv**() completely fills *local_iov[0]* before proceeding to *local_iov[1]*, and so on. Likewise, *remote_iov[0]* is completely read before proceeding to *remote_iov[1]*, and so on.

Similarly, **process_vm_writev**() writes out the entire contents of *local_iov[0]* before proceeding to *local_iov[1]*, and it completely fills *remote_iov[0]* before proceeding to *remote_iov[1]*.

The lengths of *remote_iov[i].iov_len* and *local_iov[i].iov_len* do not have to be the same. Thus, it is possible to split a single local buffer into multiple remote buffers, or vice versa.

The *flags* argument is currently unused and must be set to 0.

The values specified in the *liovcnt* and *riovcnt* arguments must be less than or equal to **IOV_MAX** (defined in <*limits.h*> or accessible via the call *sysconf(_SC_IOV_MAX)*).

The count arguments and *local_iov* are checked before doing any transfers. If the counts are too big, or *local_iov* is invalid, or the addresses refer to regions that are inaccessible to the local process, none of the vectors will be processed and an error will be returned immediately.

Note, however, that these system calls do not check the memory regions in the remote process until just before doing the read/write. Consequently, a partial read/write (see RETURN VALUE) may result if one of the *remote_iov* elements points to an invalid memory region in the remote process. No further

reads/writes will be attempted beyond that point. Keep this in mind when attempting to read data of unknown length (such as C strings that are null-terminated) from a remote process, by avoiding spanning memory pages (typically 4 KiB) in a single remote *iovec* element. (Instead, split the remote read into two *remote_iov* elements and have them merge back into a single write *local_iov* entry. The first read entry goes up to the page boundary, while the second starts on the next page boundary.)

Permission to read from or write to another process is governed by a ptrace access mode **PTRACE_MODE_ATTACH_REALCREDS** check; see *ptrace*(2).

RETURN VALUE

On success, **process_vm_readv**() returns the number of bytes read and **process_vm_writev**() returns the number of bytes written. This return value may be less than the total number of requested bytes, if a partial read/write occurred. (Partial transfers apply at the granularity of *iovec* elements. These system calls won't perform a partial transfer that splits a single *iovec* element.) The caller should check the return value to determine whether a partial read/write occurred.

On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
The memory described by *local_iov* is outside the caller's accessible address space.

EFAULT
The memory described by *remote_iov* is outside the accessible address space of the process *pid*.

EINVAL
The sum of the *iov_len* values of either *local_iov* or *remote_iov* overflows a *ssize_t* value.

EINVAL
flags is not 0.

EINVAL
liovcnt or *riovcnt* is too large.

ENOMEM
Could not allocate memory for internal copies of the *iovec* structures.

EPERM
The caller does not have permission to access the address space of the process *pid*.

ESRCH
No process with ID *pid* exists.

STANDARDS

Linux.

HISTORY

Linux 3.2, glibc 2.15.

NOTES

The data transfers performed by **process_vm_readv**() and **process_vm_writev**() are not guaranteed to be atomic in any way.

These system calls were designed to permit fast message passing by allowing messages to be exchanged with a single copy operation (rather than the double copy that would be required when using, for example, shared memory or pipes).

EXAMPLES

The following code sample demonstrates the use of **process_vm_readv**(). It reads 20 bytes at the address 0x10000 from the process with PID 10 and writes the first 10 bytes into *buf1* and the second 10 bytes into *buf2*.

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <sys/types.h>
#include <sys/uio.h>

int
```

```
main(void)
{
    char          buf1[10];
    char          buf2[10];
    pid_t         pid = 10;    /* PID of remote process */
    ssize_t       nread;
    struct iovec  local[2];
    struct iovec  remote[1];

    local[0].iov_base = buf1;
    local[0].iov_len = 10;
    local[1].iov_base = buf2;
    local[1].iov_len = 10;
    remote[0].iov_base = (void *) 0x10000;
    remote[0].iov_len = 20;

    nread = process_vm_readv(pid, local, 2, remote, 1, 0);
    if (nread != 20)
        exit(EXIT_FAILURE);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

readv(2), *writev*(2)

NAME

ptrace − process trace

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/ptrace.h>

long ptrace(enum __ptrace_request *op***, pid_t** *pid***,**
void ****addr*, void ****data***);**

DESCRIPTION

The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

```
ptrace(PTRACE_foo, pid, ...)
```

where *pid* is the thread ID of the corresponding Linux thread.

(Note that in this page, a "multithreaded process" means a thread group consisting of threads created using the *clone*(2) **CLONE_THREAD** flag.)

A process can initiate a trace by calling *fork*(2) and having the resulting child do a **PTRACE_TRACEME**, followed (typically) by an *execve*(2). Alternatively, one process may commence tracing another process using **PTRACE_ATTACH** or **PTRACE_SEIZE**.

While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is **SIGKILL**, which has its usual effect.) The tracer will be notified at its next call to *waitpid*(2) (or one of the related "wait" system calls); that call will return a *status* value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace operations to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

If the **PTRACE_O_TRACEEXEC** option is not in effect, all successful calls to *execve*(2) by the traced process will cause it to be sent a **SIGTRAP** signal, giving the parent a chance to gain control before the new program begins execution.

When the tracer is finished tracing, it can cause the tracee to continue executing in a normal, untraced mode via **PTRACE_DETACH**.

The value of *op* determines the operation to be performed:

PTRACE_TRACEME

Indicate that this process is to be traced by its parent. A process probably shouldn't make this operation if its parent isn't expecting to trace it. (*pid*, *addr*, and *data* are ignored.)

The **PTRACE_TRACEME** operation is used only by the tracee; the remaining operations are used only by the tracer. In the following operations, *pid* specifies the thread ID of the tracee to be acted on. For operations other than **PTRACE_ATTACH**, **PTRACE_SEIZE**, **PTRACE_INTERRUPT**, and **PTRACE_KILL**, the tracee must be stopped.

PTRACE_PEEKTEXT
PTRACE_PEEKDATA

Read a word at the address *addr* in the tracee's memory, returning the word as the result of the **ptrace**() call. Linux does not have separate text and data address spaces, so these two operations are currently equivalent. (*data* is ignored; but see NOTES.)

PTRACE_PEEKUSER

Read a word at offset *addr* in the tracee's USER area, which holds the registers and other information about the process (see <*sys/user.h*>). The word is returned as the result of the **ptrace**() call. Typically, the offset must be word-aligned, though this might vary by

architecture. See NOTES. (*data* is ignored; but see NOTES.)

PTRACE_POKETEXT
PTRACE_POKEDATA
Copy the word *data* to the address *addr* in the tracee's memory. As for **PTRACE_PEEKTEXT** and **PTRACE_PEEKDATA**, these two operations are currently equivalent.

PTRACE_POKEUSER
Copy the word *data* to offset *addr* in the tracee's USER area. As for **PTRACE_PEEKUSER**, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

PTRACE_GETREGS
PTRACE_GETFPREGS
Copy the tracee's general-purpose or floating-point registers, respectively, to the address *data* in the tracer. See <*sys/user.h*> for information on the format of this data. (*addr* is ignored.) Note that SPARC systems have the meaning of *data* and *addr* reversed; that is, *data* is ignored and the registers are copied to the address *addr*. **PTRACE_GETREGS** and **PTRACE_GETFPREGS** are not present on all architectures.

PTRACE_GETREGSET (since Linux 2.6.34)
Read the tracee's registers. *addr* specifies, in an architecture-dependent way, the type of registers to be read. **NT_PRSTATUS** (with numerical value 1) usually results in reading of general-purpose registers. If the CPU has, for example, floating-point and/or vector registers, they can be retrieved by setting *addr* to the corresponding **NT_foo** constant. *data* points to a **struct iovec**, which describes the destination buffer's location and length. On return, the kernel modifies **iov.len** to indicate the actual number of bytes returned.

PTRACE_SETREGS
PTRACE_SETFPREGS
Modify the tracee's general-purpose or floating-point registers, respectively, from the address *data* in the tracer. As for **PTRACE_POKEUSER**, some general-purpose register modifications may be disallowed. (*addr* is ignored.) Note that SPARC systems have the meaning of *data* and *addr* reversed; that is, *data* is ignored and the registers are copied from the address *addr*. **PTRACE_SETREGS** and **PTRACE_SETFPREGS** are not present on all architectures.

PTRACE_SETREGSET (since Linux 2.6.34)
Modify the tracee's registers. The meaning of *addr* and *data* is analogous to **PTRACE_GETREGSET**.

PTRACE_GETSIGINFO (since Linux 2.3.99-pre6)
Retrieve information about the signal that caused the stop. Copy a *siginfo_t* structure (see *sigaction*(2)) from the tracee to the address *data* in the tracer. (*addr* is ignored.)

PTRACE_SETSIGINFO (since Linux 2.3.99-pre6)
Set signal information: copy a *siginfo_t* structure from the address *data* in the tracer to the tracee. This will affect only signals that would normally be delivered to the tracee and were caught by the tracer. It may be difficult to tell these normal signals from synthetic signals generated by **ptrace**() itself. (*addr* is ignored.)

PTRACE_PEEKSIGINFO (since Linux 3.10)
Retrieve *siginfo_t* structures without removing signals from a queue. *addr* points to a *ptrace_peeksiginfo_args* structure that specifies the ordinal position from which copying of signals should start, and the number of signals to copy. *siginfo_t* structures are copied into the buffer pointed to by *data*. The return value contains the number of copied signals (zero indicates that there is no signal corresponding to the specified ordinal position). Within the returned *siginfo* structures, the *si_code* field includes information (**__SI_CHLD**, **__SI_FAULT**, etc.) that are not otherwise exposed to user space.

```
struct ptrace_peeksiginfo_args {
    u64 off;    /* Ordinal position in queue at which
                   to start copying signals */
    u32 flags;  /* PTRACE_PEEKSIGINFO_SHARED or 0 */
    s32 nr;     /* Number of signals to copy */
```

```
};
```

Currently, there is only one flag, **PTRACE_PEEKSIGINFO_SHARED**, for dumping signals from the process-wide signal queue. If this flag is not set, signals are read from the per-thread queue of the specified thread.

PTRACE_GETSIGMASK (since Linux 3.11)
Place a copy of the mask of blocked signals (see *sigprocmask*(2)) in the buffer pointed to by *data*, which should be a pointer to a buffer of type *sigset_t*. The *addr* argument contains the size of the buffer pointed to by *data* (i.e., *sizeof(sigset_t)*).

PTRACE_SETSIGMASK (since Linux 3.11)
Change the mask of blocked signals (see *sigprocmask*(2)) to the value specified in the buffer pointed to by *data*, which should be a pointer to a buffer of type *sigset_t*. The *addr* argument contains the size of the buffer pointed to by *data* (i.e., *sizeof(sigset_t)*).

PTRACE_SETOPTIONS (since Linux 2.4.6; see BUGS for caveats)
Set ptrace options from *data*. (*addr* is ignored.) *data* is interpreted as a bit mask of options, which are specified by the following flags:

PTRACE_O_EXITKILL (since Linux 3.8)
Send a **SIGKILL** signal to the tracee if the tracer exits. This option is useful for ptrace jailers that want to ensure that tracees can never escape the tracer's control.

PTRACE_O_TRACECLONE (since Linux 2.5.46)
Stop the tracee at the next *clone*(2) and automatically start tracing the newly cloned process, which will start with a **SIGSTOP**, or **PTRACE_EVENT_STOP** if **PTRACE_SEIZE** was used. A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_CLONE<<8))

The PID of the new process can be retrieved with **PTRACE_GETEVENTMSG**.

This option may not catch *clone*(2) calls in all cases. If the tracee calls *clone*(2) with the **CLONE_VFORK** flag, **PTRACE_EVENT_VFORK** will be delivered instead if **PTRACE_O_TRACEVFORK** is set; otherwise if the tracee calls *clone*(2) with the exit signal set to **SIGCHLD**, **PTRACE_EVENT_FORK** will be delivered if **PTRACE_O_TRACEFORK** is set.

PTRACE_O_TRACEEXEC (since Linux 2.5.46)
Stop the tracee at the next *execve*(2). A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_EXEC<<8))

If the execing thread is not a thread group leader, the thread ID is reset to thread group leader's ID before this stop. Since Linux 3.0, the former thread ID can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACEEXIT (since Linux 2.5.60)
Stop the tracee at exit. A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_EXIT<<8))

The tracee's exit status can be retrieved with **PTRACE_GETEVENTMSG**.

The tracee is stopped early during process exit, when registers are still available, allowing the tracer to see where the exit occurred, whereas the normal exit notification is done after the process is finished exiting. Even though context is available, the tracer cannot prevent the exit from happening at this point.

PTRACE_O_TRACEFORK (since Linux 2.5.46)
Stop the tracee at the next *fork*(2) and automatically start tracing the newly forked process, which will start with a **SIGSTOP**, or **PTRACE_EVENT_STOP** if **PTRACE_SEIZE** was used. A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_FORK<<8))

The PID of the new process can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACESYSGOOD (since Linux 2.4.6)
When delivering system call traps, set bit 7 in the signal number (i.e., deliver *SIGTRAP|0x80*). This makes it easy for the tracer to distinguish normal traps from those caused by a system call.

PTRACE_O_TRACEVFORK (since Linux 2.5.46)
Stop the tracee at the next *vfork*(2) and automatically start tracing the newly vforked process, which will start with a **SIGSTOP**, or **PTRACE_EVENT_STOP** if **PTRACE_SEIZE** was used. A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK<<8))

The PID of the new process can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACEVFORKDONE (since Linux 2.5.60)
Stop the tracee at the completion of the next *vfork*(2). A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK_DONE<<8))

The PID of the new process can (since Linux 2.6.18) be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACESECCOMP (since Linux 3.5)
Stop the tracee when a *seccomp*(2) **SECCOMP_RET_TRACE** rule is triggered. A *waitpid*(2) by the tracer will return a *status* value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_SECCOMP<<8))

While this triggers a **PTRACE_EVENT** stop, it is similar to a syscall-enter-stop. For details, see the note on **PTRACE_EVENT_SECCOMP** below. The seccomp event message data (from the **SECCOMP_RET_DATA** portion of the seccomp filter rule) can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_SUSPEND_SECCOMP (since Linux 4.3)
Suspend the tracee's seccomp protections. This applies regardless of mode, and can be used when the tracee has not yet installed seccomp filters. That is, a valid use case is to suspend a tracee's seccomp protections before they are installed by the tracee, let the tracee install the filters, and then clear this flag when the filters should be resumed. Setting this option requires that the tracer have the **CAP_SYS_ADMIN** capability, not have any seccomp protections installed, and not have **PTRACE_O_SUSPEND_SECCOMP** set on itself.

PTRACE_GETEVENTMSG (since Linux 2.5.46)
Retrieve a message (as an *unsigned long*) about the ptrace event that just happened, placing it at the address *data* in the tracer. For **PTRACE_EVENT_EXIT**, this is the tracee's exit status. For **PTRACE_EVENT_FORK**, **PTRACE_EVENT_VFORK**, **PTRACE_EVENT_VFORK_DONE**, and **PTRACE_EVENT_CLONE**, this is the PID of the new process. For **PTRACE_EVENT_SECCOMP**, this is the *seccomp*(2) filter's **SECCOMP_RET_DATA** associated with the triggered rule. (*addr* is ignored.)

PTRACE_CONT
Restart the stopped tracee process. If *data* is nonzero, it is interpreted as the number of a signal to be delivered to the tracee; otherwise, no signal is delivered. Thus, for example, the tracer can control whether a signal sent to the tracee is delivered or not. (*addr* is ignored.)

PTRACE_SYSCALL
PTRACE_SINGLESTEP
Restart the stopped tracee as for **PTRACE_CONT**, but arrange for the tracee to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The tracee will also, as usual, be stopped upon receipt of a signal.) From the tracer's perspective, the tracee will appear to have been stopped by receipt of a **SIGTRAP**. So, for **PTRACE_SYSCALL**, for example, the idea is to inspect the arguments to the system call at

the first stop, then do another **PTRACE_SYSCALL** and inspect the return value of the system call at the second stop. The *data* argument is treated as for **PTRACE_CONT**. (*addr* is ignored.)

PTRACE_SET_SYSCALL (since Linux 2.6.16)
When in syscall-enter-stop, change the number of the system call that is about to be executed to the number specified in the *data* argument. The *addr* argument is ignored. This operation is currently supported only on arm (and arm64, though only for backwards compatibility), but most other architectures have other means of accomplishing this (usually by changing the register that the userland code passed the system call number in).

PTRACE_SYSEMU
PTRACE_SYSEMU_SINGLESTEP (since Linux 2.6.14)
For **PTRACE_SYSEMU**, continue and stop on entry to the next system call, which will not be executed. See the documentation on syscall-stops below. For **PTRACE_SYSEMU_SINGLESTEP**, do the same but also singlestep if not a system call. This call is used by programs like User Mode Linux that want to emulate all the tracee's system calls. The *data* argument is treated as for **PTRACE_CONT**. The *addr* argument is ignored. These operations are currently supported only on x86.

PTRACE_LISTEN (since Linux 3.4)
Restart the stopped tracee, but prevent it from executing. The resulting state of the tracee is similar to a process which has been stopped by a **SIGSTOP** (or other stopping signal). See the "group-stop" subsection for additional information. **PTRACE_LISTEN** works only on tracees attached by **PTRACE_SEIZE**.

PTRACE_KILL
Send the tracee a **SIGKILL** to terminate it. (*addr* and *data* are ignored.)

This operation is deprecated; do not use it! Instead, send a **SIGKILL** directly using *kill*(2) or *tgkill*(2). The problem with **PTRACE_KILL** is that it requires the tracee to be in signal-delivery-stop, otherwise it may not work (i.e., may complete successfully but won't kill the tracee). By contrast, sending a **SIGKILL** directly has no such limitation.

PTRACE_INTERRUPT (since Linux 3.4)
Stop a tracee. If the tracee is running or sleeping in kernel space and **PTRACE_SYSCALL** is in effect, the system call is interrupted and syscall-exit-stop is reported. (The interrupted system call is restarted when the tracee is restarted.) If the tracee was already stopped by a signal and **PTRACE_LISTEN** was sent to it, the tracee stops with **PTRACE_EVENT_STOP** and *WSTOPSIG(status)* returns the stop signal. If any other ptrace-stop is generated at the same time (for example, if a signal is sent to the tracee), this ptrace-stop happens. If none of the above applies (for example, if the tracee is running in user space), it stops with **PTRACE_EVENT_STOP** with *WSTOPSIG(status)* == **SIGTRAP**. **PTRACE_INTERRUPT** only works on tracees attached by **PTRACE_SEIZE**.

PTRACE_ATTACH
Attach to the process specified in *pid*, making it a tracee of the calling process. The tracee is sent a **SIGSTOP**, but will not necessarily have stopped by the completion of this call; use *waitpid*(2) to wait for the tracee to stop. See the "Attaching and detaching" subsection for additional information. (*addr* and *data* are ignored.)

Permission to perform a **PTRACE_ATTACH** is governed by a ptrace access mode **PTRACE_MODE_ATTACH_REALCREDS** check; see below.

PTRACE_SEIZE (since Linux 3.4)
Attach to the process specified in *pid*, making it a tracee of the calling process. Unlike **PTRACE_ATTACH**, **PTRACE_SEIZE** does not stop the process. Group-stops are reported as **PTRACE_EVENT_STOP** and *WSTOPSIG(status)* returns the stop signal. Automatically attached children stop with **PTRACE_EVENT_STOP** and *WSTOPSIG(status)* returns **SIGTRAP** instead of having **SIGSTOP** signal delivered to them. *execve*(2) does not deliver an extra **SIGTRAP**. Only a **PTRACE_SEIZE**d process can accept **PTRACE_INTERRUPT** and **PTRACE_LISTEN** commands. The "seized" behavior just described is inherited by children that are automatically attached using **PTRACE_O_TRACEFORK**, **PTRACE_O_TRACEVFORK**, and **PTRACE_O_TRACECLONE**. *addr* must be zero.

data contains a bit mask of ptrace options to activate immediately.

Permission to perform a **PTRACE_SEIZE** is governed by a ptrace access mode **PTRACE_MODE_ATTACH_REALCREDS** check; see below.

PTRACE_SECCOMP_GET_FILTER (since Linux 4.4)
This operation allows the tracer to dump the tracee's classic BPF filters.

addr is an integer specifying the index of the filter to be dumped. The most recently installed filter has the index 0. If *addr* is greater than the number of installed filters, the operation fails with the error **ENOENT**.

data is either a pointer to a *struct sock_filter* array that is large enough to store the BPF program, or NULL if the program is not to be stored.

Upon success, the return value is the number of instructions in the BPF program. If *data* was NULL, then this return value can be used to correctly size the *struct sock_filter* array passed in a subsequent call.

This operation fails with the error **EACCES** if the caller does not have the **CAP_SYS_ADMIN** capability or if the caller is in strict or filter seccomp mode. If the filter referred to by *addr* is not a classic BPF filter, the operation fails with the error **EMEDIUMTYPE**.

This operation is available if the kernel was configured with both the **CONFIG_SECCOMP_FILTER** and the **CONFIG_CHECKPOINT_RESTORE** options.

PTRACE_DETACH
Restart the stopped tracee as for **PTRACE_CONT**, but first detach from it. Under Linux, a tracee can be detached in this way regardless of which method was used to initiate tracing. (*addr* is ignored.)

PTRACE_GET_THREAD_AREA (since Linux 2.6.0)
This operation performs a similar task to *get_thread_area*(2). It reads the TLS entry in the GDT whose index is given in *addr*, placing a copy of the entry into the *struct user_desc* pointed to by *data*. (By contrast with *get_thread_area*(2), the *entry_number* of the *struct user_desc* is ignored.)

PTRACE_SET_THREAD_AREA (since Linux 2.6.0)
This operation performs a similar task to *set_thread_area*(2). It sets the TLS entry in the GDT whose index is given in *addr*, assigning it the data supplied in the *struct user_desc* pointed to by *data*. (By contrast with *set_thread_area*(2), the *entry_number* of the *struct user_desc* is ignored; in other words, this ptrace operation can't be used to allocate a free TLS entry.)

PTRACE_GET_SYSCALL_INFO (since Linux 5.3)
Retrieve information about the system call that caused the stop. The information is placed into the buffer pointed by the *data* argument, which should be a pointer to a buffer of type *struct ptrace_syscall_info*. The *addr* argument contains the size of the buffer pointed to by the *data* argument (i.e., *sizeof(struct ptrace_syscall_info)*). The return value contains the number of bytes available to be written by the kernel. If the size of the data to be written by the kernel exceeds the size specified by the *addr* argument, the output data is truncated.

The *ptrace_syscall_info* structure contains the following fields:

```
struct ptrace_syscall_info {
    __u8 op;        /* Type of system call stop */
    __u32 arch;     /* AUDIT_ARCH_* value; see seccomp(2) */
    __u64 instruction_pointer; /* CPU instruction pointer */
    __u64 stack_pointer;    /* CPU stack pointer */
    union {
        struct {    /* op == PTRACE_SYSCALL_INFO_ENTRY */
            __u64 nr;       /* System call number */
            __u64 args[6];  /* System call arguments */
        } entry;
        struct {    /* op == PTRACE_SYSCALL_INFO_EXIT */
            __s64 rval;     /* System call return value */
            __u8 is_error;  /* System call error flag;
```

```
                                    Boolean: does rval contain
                                    an error value (-ERRCODE) or
                                    a nonerror return value? */
                } exit;
                struct {    /* op == PTRACE_SYSCALL_INFO_SECCOMP */
                    __u64 nr;       /* System call number */
                    __u64 args[6];  /* System call arguments */
                    __u32 ret_data; /* SECCOMP_RET_DATA portion
                                       of SECCOMP_RET_TRACE
                                       return value */
                } seccomp;
            };
        };
```

The *op*, *arch*, *instruction_pointer*, and *stack_pointer* fields are defined for all kinds of ptrace system call stops. The rest of the structure is a union; one should read only those fields that are meaningful for the kind of system call stop specified by the *op* field.

The *op* field has one of the following values (defined in <*linux/ptrace.h*>) indicating what type of stop occurred and which part of the union is filled:

PTRACE_SYSCALL_INFO_ENTRY
: The *entry* component of the union contains information relating to a system call entry stop.

PTRACE_SYSCALL_INFO_EXIT
: The *exit* component of the union contains information relating to a system call exit stop.

PTRACE_SYSCALL_INFO_SECCOMP
: The *seccomp* component of the union contains information relating to a **PTRACE_EVENT_SECCOMP** stop.

PTRACE_SYSCALL_INFO_NONE
: No component of the union contains relevant information.

In case of system call entry or exit stops, the data returned by **PTRACE_GET_SYSCALL_INFO** is limited to type **PTRACE_SYSCALL_INFO_NONE** unless **PTRACE_O_TRACESYSGOOD** option is set before the corresponding system call stop has occurred.

Death under ptrace

When a (possibly multithreaded) process receives a killing signal (one whose disposition is set to **SIG_DFL** and whose default action is to kill the process), all threads exit. Tracees report their death to their tracer(s). Notification of this event is delivered via *waitpid*(2).

Note that the killing signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will death from the signal happen on *all* tracees within a multithreaded process. (The term "signal-delivery-stop" is explained below.)

SIGKILL does not generate signal-delivery-stop and therefore the tracer can't suppress it. **SIGKILL** kills even within system calls (syscall-exit-stop is not generated prior to death by **SIGKILL**). The net effect is that **SIGKILL** always kills the process (all its threads), even if some threads of the process are ptraced.

When the tracee calls *_exit*(2), it reports its death to its tracer. Other threads are not affected.

When any thread executes *exit_group*(2), every tracee in its thread group reports its death to its tracer.

If the **PTRACE_O_TRACEEXIT** option is on, **PTRACE_EVENT_EXIT** will happen before actual death. This applies to exits via *exit*(2), *exit_group*(2), and signal deaths (except **SIGKILL**, depending on the kernel version; see BUGS below), and when threads are torn down on *execve*(2) in a multithreaded process.

The tracer cannot assume that the ptrace-stopped tracee exists. There are many scenarios when the tracee may die while stopped (such as **SIGKILL**). Therefore, the tracer must be prepared to handle an

ESRCH error on any ptrace operation. Unfortunately, the same error is returned if the tracee exists but is not ptrace-stopped (for commands which require a stopped tracee), or if it is not traced by the process which issued the ptrace call. The tracer needs to keep track of the stopped/running state of the tracee, and interpret **ESRCH** as "tracee died unexpectedly" only if it knows that the tracee has been observed to enter ptrace-stop. Note that there is no guarantee that *waitpid(WNOHANG)* will reliably report the tracee's death status if a ptrace operation returned **ESRCH**. *waitpid(WNOHANG)* may return 0 instead. In other words, the tracee may be "not yet fully dead", but already refusing ptrace operations.

The tracer can't assume that the tracee *always* ends its life by reporting *WIFEXITED(status)* or *WIFSIGNALED(status)*; there are cases where this does not occur. For example, if a thread other than thread group leader does an *execve*(2), it disappears; its PID will never be seen again, and any subsequent ptrace stops will be reported under the thread group leader's PID.

Stopped states

A tracee can be in two states: running or stopped. For the purposes of ptrace, a tracee which is blocked in a system call (such as *read*(2), *pause*(2), etc.) is nevertheless considered to be running, even if the tracee is blocked for a long time. The state of the tracee after **PTRACE_LISTEN** is somewhat of a gray area: it is not in any ptrace-stop (ptrace commands won't work on it, and it will deliver *waitpid*(2) notifications), but it also may be considered "stopped" because it is not executing instructions (is not scheduled), and if it was in group-stop before **PTRACE_LISTEN**, it will not respond to signals until **SIGCONT** is received.

There are many kinds of states when the tracee is stopped, and in ptrace discussions they are often conflated. Therefore, it is important to use precise terms.

In this manual page, any stopped state in which the tracee is ready to accept ptrace commands from the tracer is called *ptrace-stop*. Ptrace-stops can be further subdivided into *signal-delivery-stop*, *group-stop*, *syscall-stop*, *PTRACE_EVENT stops*, and so on. These stopped states are described in detail below.

When the running tracee enters ptrace-stop, it notifies its tracer using *waitpid*(2) (or one of the other "wait" system calls). Most of this manual page assumes that the tracer waits with:

```
pid = waitpid(pid_or_minus_1, &status, __WALL);
```

Ptrace-stopped tracees are reported as returns with *pid* greater than 0 and *WIFSTOPPED(status)* true.

The **__WALL** flag does not include the **WSTOPPED** and **WEXITED** flags, but implies their functionality.

Setting the **WCONTINUED** flag when calling *waitpid*(2) is not recommended: the "continued" state is per-process and consuming it can confuse the real parent of the tracee.

Use of the **WNOHANG** flag may cause *waitpid*(2) to return 0 ("no wait results available yet") even if the tracer knows there should be a notification. Example:

```
errno = 0;
ptrace(PTRACE_CONT, pid, 0L, 0L);
if (errno == ESRCH) {
    /* tracee is dead */
    r = waitpid(tracee, &status, __WALL | WNOHANG);
    /* r can still be 0 here! */
}
```

The following kinds of ptrace-stops exist: signal-delivery-stops, group-stops, **PTRACE_EVENT** stops, syscall-stops. They all are reported by *waitpid*(2) with *WIFSTOPPED(status)* true. They may be differentiated by examining the value *status>>8*, and if there is ambiguity in that value, by querying **PTRACE_GETSIGINFO**. (Note: the *WSTOPSIG(status)* macro can't be used to perform this examination, because it returns the value *(status>>8) & 0xff*.)

Signal-delivery-stop

When a (possibly multithreaded) process receives any signal except **SIGKILL**, the kernel selects an arbitrary thread which handles the signal. (If the signal is generated with *tgkill*(2), the target thread can be explicitly selected by the caller.) If the selected thread is traced, it enters signal-delivery-stop. At this point, the signal is not yet delivered to the process, and can be suppressed by the tracer. If the

tracer doesn't suppress the signal, it passes the signal to the tracee in the next ptrace restart operation. This second step of signal delivery is called *signal injection* in this manual page. Note that if the signal is blocked, signal-delivery-stop doesn't happen until the signal is unblocked, with the usual exception that **SIGSTOP** can't be blocked.

Signal-delivery-stop is observed by the tracer as *waitpid*(2) returning with *WIFSTOPPED(status)* true, with the signal returned by *WSTOPSIG(status)*. If the signal is **SIGTRAP**, this may be a different kind of ptrace-stop; see the "Syscall-stops" and "execve" sections below for details. If *WSTOPSIG(status)* returns a stopping signal, this may be a group-stop; see below.

Signal injection and suppression

After signal-delivery-stop is observed by the tracer, the tracer should restart the tracee with the call

```
ptrace(PTRACE_restart, pid, 0, sig)
```

where **PTRACE_restart** is one of the restarting ptrace operations. If *sig* is 0, then a signal is not delivered. Otherwise, the signal *sig* is delivered. This operation is called *signal injection* in this manual page, to distinguish it from signal-delivery-stop.

The *sig* value may be different from the *WSTOPSIG(status)* value: the tracer can cause a different signal to be injected.

Note that a suppressed signal still causes system calls to return prematurely. In this case, system calls will be restarted: the tracer will observe the tracee to reexecute the interrupted system call (or *restart_syscall*(2) system call for a few system calls which use a different mechanism for restarting) if the tracer uses **PTRACE_SYSCALL**. Even system calls (such as *poll*(2)) which are not restartable after signal are restarted after signal is suppressed; however, kernel bugs exist which cause some system calls to fail with **EINTR** even though no observable signal is injected to the tracee.

Restarting ptrace commands issued in ptrace-stops other than signal-delivery-stop are not guaranteed to inject a signal, even if *sig* is nonzero. No error is reported; a nonzero *sig* may simply be ignored. Ptrace users should not try to "create a new signal" this way: use *tgkill*(2) instead.

The fact that signal injection operations may be ignored when restarting the tracee after ptrace stops that are not signal-delivery-stops is a cause of confusion among ptrace users. One typical scenario is that the tracer observes group-stop, mistakes it for signal-delivery-stop, restarts the tracee with

```
ptrace(PTRACE_restart, pid, 0, stopsig)
```

with the intention of injecting *stopsig*, but *stopsig* gets ignored and the tracee continues to run.

The **SIGCONT** signal has a side effect of waking up (all threads of) a group-stopped process. This side effect happens before signal-delivery-stop. The tracer can't suppress this side effect (it can only suppress signal injection, which only causes the **SIGCONT** handler to not be executed in the tracee, if such a handler is installed). In fact, waking up from group-stop may be followed by signal-delivery-stop for signal(s) *other than* **SIGCONT**, if they were pending when **SIGCONT** was delivered. In other words, **SIGCONT** may be not the first signal observed by the tracee after it was sent.

Stopping signals cause (all threads of) a process to enter group-stop. This side effect happens after signal injection, and therefore can be suppressed by the tracer.

In Linux 2.4 and earlier, the **SIGSTOP** signal can't be injected.

PTRACE_GETSIGINFO can be used to retrieve a *siginfo_t* structure which corresponds to the delivered signal. **PTRACE_SETSIGINFO** may be used to modify it. If **PTRACE_SETSIGINFO** has been used to alter *siginfo_t*, the *si_signo* field and the *sig* parameter in the restarting command must match, otherwise the result is undefined.

Group-stop

When a (possibly multithreaded) process receives a stopping signal, all threads stop. If some threads are traced, they enter a group-stop. Note that the stopping signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will group-stop be initiated on *all* tracees within the multithreaded process. As usual, every tracee reports its group-stop separately to the corresponding tracer.

Group-stop is observed by the tracer as *waitpid*(2) returning with *WIFSTOPPED(status)* true, with the stopping signal available via *WSTOPSIG(status)*. The same result is returned by some other classes of ptrace-stops, therefore the recommended practice is to perform the call

```
ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo)
```

The call can be avoided if the signal is not **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**; only these four signals are stopping signals. If the tracer sees something else, it can't be a group-stop. Otherwise, the tracer needs to call **PTRACE_GETSIGINFO**. If **PTRACE_GETSIGINFO** fails with **EINVAL**, then it is definitely a group-stop. (Other failure codes are possible, such as **ESRCH** ("no such process") if a **SIGKILL** killed the tracee.)

If tracee was attached using **PTRACE_SEIZE**, group-stop is indicated by **PTRACE_EVENT_STOP**: *status>>16 == PTRACE_EVENT_STOP*. This allows detection of group-stops without requiring an extra **PTRACE_GETSIGINFO** call.

As of Linux 2.6.38, after the tracer sees the tracee ptrace-stop and until it restarts or kills it, the tracee will not run, and will not send notifications (except **SIGKILL** death) to the tracer, even if the tracer enters into another *waitpid*(2) call.

The kernel behavior described in the previous paragraph causes a problem with transparent handling of stopping signals. If the tracer restarts the tracee after group-stop, the stopping signal is effectively ignored—the tracee doesn't remain stopped, it runs. If the tracer doesn't restart the tracee before entering into the next *waitpid*(2), future **SIGCONT** signals will not be reported to the tracer; this would cause the **SIGCONT** signals to have no effect on the tracee.

Since Linux 3.4, there is a method to overcome this problem: instead of **PTRACE_CONT**, a **PTRACE_LISTEN** command can be used to restart a tracee in a way where it does not execute, but waits for a new event which it can report via *waitpid*(2) (such as when it is restarted by a **SIGCONT**).

PTRACE_EVENT stops

If the tracer sets **PTRACE_O_TRACE_*** options, the tracee will enter ptrace-stops called **PTRACE_EVENT** stops.

PTRACE_EVENT stops are observed by the tracer as *waitpid*(2) returning with *WIFSTOPPED(status)*, and *WSTOPSIG(status)* returns **SIGTRAP** (or for **PTRACE_EVENT_STOP**, returns the stopping signal if tracee is in a group-stop). An additional bit is set in the higher byte of the status word: the value *status>>8* will be

```
((PTRACE_EVENT_foo<<8) | SIGTRAP).
```

The following events exist:

PTRACE_EVENT_VFORK
: Stop before return from *vfork*(2) or *clone*(2) with the **CLONE_VFORK** flag. When the tracee is continued after this stop, it will wait for child to exit/exec before continuing its execution (in other words, the usual behavior on *vfork*(2)).

PTRACE_EVENT_FORK
: Stop before return from *fork*(2) or *clone*(2) with the exit signal set to **SIGCHLD**.

PTRACE_EVENT_CLONE
: Stop before return from *clone*(2).

PTRACE_EVENT_VFORK_DONE
: Stop before return from *vfork*(2) or *clone*(2) with the **CLONE_VFORK** flag, but after the child unblocked this tracee by exiting or execing.

For all four stops described above, the stop occurs in the parent (i.e., the tracee), not in the newly created thread. **PTRACE_GETEVENTMSG** can be used to retrieve the new thread's ID.

PTRACE_EVENT_EXEC
: Stop before return from *execve*(2). Since Linux 3.0, **PTRACE_GETEVENTMSG** returns the former thread ID.

PTRACE_EVENT_EXIT
: Stop before exit (including death from *exit_group*(2)), signal death, or exit caused by *execve*(2) in a multithreaded process. **PTRACE_GETEVENTMSG** returns the exit status. Registers can be examined (unlike when "real" exit happens). The tracee is still alive; it needs to be **PTRACE_CONT**ed or **PTRACE_DETACH**ed to finish exiting.

PTRACE_EVENT_STOP
Stop induced by **PTRACE_INTERRUPT** command, or group-stop, or initial ptrace-stop when a new child is attached (only if attached using **PTRACE_SEIZE**).

PTRACE_EVENT_SECCOMP
Stop triggered by a *seccomp*(2) rule on tracee syscall entry when **PTRACE_O_TRACESECCOMP** has been set by the tracer. The seccomp event message data (from the **SECCOMP_RET_DATA** portion of the seccomp filter rule) can be retrieved with **PTRACE_GETEVENTMSG**. The semantics of this stop are described in detail in a separate section below.

PTRACE_GETSIGINFO on **PTRACE_EVENT** stops returns **SIGTRAP** in *si_signo*, with *si_code* set to *(event<<8) | SIGTRAP*.

Syscall-stops
If the tracee was restarted by **PTRACE_SYSCALL** or **PTRACE_SYSEMU**, the tracee enters syscall-enter-stop just prior to entering any system call (which will not be executed if the restart was using **PTRACE_SYSEMU**, regardless of any change made to registers at this point or how the tracee is restarted after this stop). No matter which method caused the syscall-entry-stop, if the tracer restarts the tracee with **PTRACE_SYSCALL**, the tracee enters syscall-exit-stop when the system call is finished, or if it is interrupted by a signal. (That is, signal-delivery-stop never happens between syscall-enter-stop and syscall-exit-stop; it happens *after* syscall-exit-stop.). If the tracee is continued using any other method (including **PTRACE_SYSEMU**), no syscall-exit-stop occurs. Note that all mentions **PTRACE_SYSEMU** apply equally to **PTRACE_SYSEMU_SINGLESTEP**.

However, even if the tracee was continued using **PTRACE_SYSCALL**, it is not guaranteed that the next stop will be a syscall-exit-stop. Other possibilities are that the tracee may stop in a **PTRACE_EVENT** stop (including seccomp stops), exit (if it entered *_exit*(2) or *exit_group*(2)), be killed by **SIGKILL**, or die silently (if it is a thread group leader, the *execve*(2) happened in another thread, and that thread is not traced by the same tracer; this situation is discussed later).

Syscall-enter-stop and syscall-exit-stop are observed by the tracer as *waitpid*(2) returning with *WIFSTOPPED(status)* true, and *WSTOPSIG(status)* giving **SIGTRAP**. If the **PTRACE_O_TRACESYSGOOD** option was set by the tracer, then *WSTOPSIG(status)* will give the value *(SIGTRAP | 0x80)*.

Syscall-stops can be distinguished from signal-delivery-stop with **SIGTRAP** by querying **PTRACE_GETSIGINFO** for the following cases:

si_code <= 0
SIGTRAP was delivered as a result of a user-space action, for example, a system call (**tgkill**(2), *kill*(2), *sigqueue*(3), etc.), expiration of a POSIX timer, change of state on a POSIX message queue, or completion of an asynchronous I/O operation.

si_code == SI_KERNEL (0x80)
SIGTRAP was sent by the kernel.

si_code == SIGTRAP or *si_code* == (SIGTRAP|0x80)
This is a syscall-stop.

However, syscall-stops happen very often (twice per system call), and performing **PTRACE_GETSIGINFO** for every syscall-stop may be somewhat expensive.

Some architectures allow the cases to be distinguished by examining registers. For example, on x86, *rax* == −**ENOSYS** in syscall-enter-stop. Since **SIGTRAP** (like any other signal) always happens *after* syscall-exit-stop, and at this point *rax* almost never contains −**ENOSYS**, the **SIGTRAP** looks like "syscall-stop which is not syscall-enter-stop"; in other words, it looks like a "stray syscall-exit-stop" and can be detected this way. But such detection is fragile and is best avoided.

Using the **PTRACE_O_TRACESYSGOOD** option is the recommended method to distinguish syscall-stops from other kinds of ptrace-stops, since it is reliable and does not incur a performance penalty.

Syscall-enter-stop and syscall-exit-stop are indistinguishable from each other by the tracer. The tracer needs to keep track of the sequence of ptrace-stops in order to not misinterpret syscall-enter-stop as syscall-exit-stop or vice versa. In general, a syscall-enter-stop is always followed by syscall-exit-stop, **PTRACE_EVENT** stop, or the tracee’s death; no other kinds of ptrace-stop can occur in between.

However, note that seccomp stops (see below) can cause syscall-exit-stops, without preceding syscall-entry-stops. If seccomp is in use, care needs to be taken not to misinterpret such stops as syscall-entry-stops.

If after syscall-enter-stop, the tracer uses a restarting command other than **PTRACE_SYSCALL**, syscall-exit-stop is not generated.

PTRACE_GETSIGINFO on syscall-stops returns **SIGTRAP** in *si_signo*, with *si_code* set to **SIGTRAP** or *(SIGTRAP|0x80)*.

PTRACE_EVENT_SECCOMP stops (Linux 3.5 to Linux 4.7)

The behavior of **PTRACE_EVENT_SECCOMP** stops and their interaction with other kinds of ptrace stops has changed between kernel versions. This documents the behavior from their introduction until Linux 4.7 (inclusive). The behavior in later kernel versions is documented in the next section.

A **PTRACE_EVENT_SECCOMP** stop occurs whenever a **SECCOMP_RET_TRACE** rule is triggered. This is independent of which methods was used to restart the system call. Notably, seccomp still runs even if the tracee was restarted using **PTRACE_SYSEMU** and this system call is unconditionally skipped.

Restarts from this stop will behave as if the stop had occurred right before the system call in question. In particular, both **PTRACE_SYSCALL** and **PTRACE_SYSEMU** will normally cause a subsequent syscall-entry-stop. However, if after the **PTRACE_EVENT_SECCOMP** the system call number is negative, both the syscall-entry-stop and the system call itself will be skipped. This means that if the system call number is negative after a **PTRACE_EVENT_SECCOMP** and the tracee is restarted using **PTRACE_SYSCALL**, the next observed stop will be a syscall-exit-stop, rather than the syscall-entry-stop that might have been expected.

PTRACE_EVENT_SECCOMP stops (since Linux 4.8)

Starting with Linux 4.8, the **PTRACE_EVENT_SECCOMP** stop was reordered to occur between syscall-entry-stop and syscall-exit-stop. Note that seccomp no longer runs (and no **PTRACE_EVENT_SECCOMP** will be reported) if the system call is skipped due to **PTRACE_SYSEMU**.

Functionally, a **PTRACE_EVENT_SECCOMP** stop functions comparably to a syscall-entry-stop (i.e., continuations using **PTRACE_SYSCALL** will cause syscall-exit-stops, the system call number may be changed and any other modified registers are visible to the to-be-executed system call as well). Note that there may be, but need not have been a preceding syscall-entry-stop.

After a **PTRACE_EVENT_SECCOMP** stop, seccomp will be rerun, with a **SECCOMP_RET_TRACE** rule now functioning the same as a **SECCOMP_RET_ALLOW**. Specifically, this means that if registers are not modified during the **PTRACE_EVENT_SECCOMP** stop, the system call will then be allowed.

PTRACE_SINGLESTEP stops

[Details of these kinds of stops are yet to be documented.]

Informational and restarting ptrace commands

Most ptrace commands (all except **PTRACE_ATTACH**, **PTRACE_SEIZE**, **PTRACE_TRACEME**, **PTRACE_INTERRUPT**, and **PTRACE_KILL**) require the tracee to be in a ptrace-stop, otherwise they fail with **ESRCH**.

When the tracee is in ptrace-stop, the tracer can read and write data to the tracee using informational commands. These commands leave the tracee in ptrace-stopped state:

```
ptrace(PTRACE_PEEKTEXT/PEEKDATA/PEEKUSER, pid, addr, 0);
ptrace(PTRACE_POKETEXT/POKEDATA/POKEUSER, pid, addr, long_val);
ptrace(PTRACE_GETREGS/GETFPREGS, pid, 0, &struct);
ptrace(PTRACE_SETREGS/SETFPREGS, pid, 0, &struct);
ptrace(PTRACE_GETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_SETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_SETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_GETEVENTMSG, pid, 0, &long_var);
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);
```

Note that some errors are not reported. For example, setting signal information (*siginfo*) may have no effect in some ptrace-stops, yet the call may succeed (return 0 and not set *errno*); querying **PTRACE_GETEVENTMSG** may succeed and return some random value if current ptrace-stop is not documented as returning a meaningful event message.

The call

```
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);
```

affects one tracee. The tracee's current flags are replaced. Flags are inherited by new tracees created and "auto-attached" via active **PTRACE_O_TRACEFORK**, **PTRACE_O_TRACEVFORK**, or **PTRACE_O_TRACECLONE** options.

Another group of commands makes the ptrace-stopped tracee run. They have the form:

```
ptrace(cmd, pid, 0, sig);
```

where *cmd* is **PTRACE_CONT**, **PTRACE_LISTEN**, **PTRACE_DETACH**, **PTRACE_SYSCALL**, **PTRACE_SINGLESTEP**, **PTRACE_SYSEMU**, or **PTRACE_SYSEMU_SINGLESTEP**. If the tracee is in signal-delivery-stop, *sig* is the signal to be injected (if it is nonzero). Otherwise, *sig* may be ignored. (When restarting a tracee from a ptrace-stop other than signal-delivery-stop, recommended practice is to always pass 0 in *sig*.)

Attaching and detaching

A thread can be attached to the tracer using the call

```
ptrace(PTRACE_ATTACH, pid, 0, 0);
```

or

```
ptrace(PTRACE_SEIZE, pid, 0, PTRACE_O_flags);
```

PTRACE_ATTACH sends **SIGSTOP** to this thread. If the tracer wants this **SIGSTOP** to have no effect, it needs to suppress it. Note that if other signals are concurrently sent to this thread during attach, the tracer may see the tracee enter signal-delivery-stop with other signal(s) first! The usual practice is to reinject these signals until **SIGSTOP** is seen, then suppress **SIGSTOP** injection. The design bug here is that a ptrace attach and a concurrently delivered **SIGSTOP** may race and the concurrent **SIGSTOP** may be lost.

Since attaching sends **SIGSTOP** and the tracer usually suppresses it, this may cause a stray **EINTR** return from the currently executing system call in the tracee, as described in the "Signal injection and suppression" section.

Since Linux 3.4, **PTRACE_SEIZE** can be used instead of **PTRACE_ATTACH**. **PTRACE_SEIZE** does not stop the attached process. If you need to stop it after attach (or at any other time) without sending it any signals, use **PTRACE_INTERRUPT** command.

The operation

```
ptrace(PTRACE_TRACEME, 0, 0, 0);
```

turns the calling thread into a tracee. The thread continues to run (doesn't enter ptrace-stop). A common practice is to follow the **PTRACE_TRACEME** with

```
raise(SIGSTOP);
```

and allow the parent (which is our tracer now) to observe our signal-delivery-stop.

If the **PTRACE_O_TRACEFORK**, **PTRACE_O_TRACEVFORK**, or **PTRACE_O_TRACECLONE** options are in effect, then children created by, respectively, *vfork*(2) or *clone*(2) with the **CLONE_VFORK** flag, *fork*(2) or *clone*(2) with the exit signal set to **SIGCHLD**, and other kinds of *clone*(2), are automatically attached to the same tracer which traced their parent. **SIGSTOP** is delivered to the children, causing them to enter signal-delivery-stop after they exit the system call which created them.

Detaching of the tracee is performed by:

```
ptrace(PTRACE_DETACH, pid, 0, sig);
```

PTRACE_DETACH is a restarting operation; therefore it requires the tracee to be in ptrace-stop. If the tracee is in signal-delivery-stop, a signal can be injected. Otherwise, the *sig* parameter may be silently ignored.

If the tracee is running when the tracer wants to detach it, the usual solution is to send **SIGSTOP** (using *tgkill*(2), to make sure it goes to the correct thread), wait for the tracee to stop in signal-delivery-stop for **SIGSTOP** and then detach it (suppressing **SIGSTOP** injection). A design bug is that this can race with concurrent **SIGSTOP**s. Another complication is that the tracee may enter other ptrace-stops and needs to be restarted and waited for again, until **SIGSTOP** is seen. Yet another complication is to be sure that the tracee is not already ptrace-stopped, because no signal delivery happens while it is—not even **SIGSTOP**.

If the tracer dies, all tracees are automatically detached and restarted, unless they were in group-stop. Handling of restart from group-stop is currently buggy, but the "as planned" behavior is to leave tracee stopped and waiting for **SIGCONT**. If the tracee is restarted from signal-delivery-stop, the pending signal is injected.

execve(2) under ptrace

When one thread in a multithreaded process calls *execve*(2), the kernel destroys all other threads in the process, and resets the thread ID of the execing thread to the thread group ID (process ID). (Or, to put things another way, when a multithreaded process does an *execve*(2), at completion of the call, it appears as though the *execve*(2) occurred in the thread group leader, regardless of which thread did the *execve*(2).) This resetting of the thread ID looks very confusing to tracers:

- All other threads stop in **PTRACE_EVENT_EXIT** stop, if the **PTRACE_O_TRACEEXIT** option was turned on. Then all other threads except the thread group leader report death as if they exited via *_exit*(2) with exit code 0.
- The execing tracee changes its thread ID while it is in the *execve*(2). (Remember, under ptrace, the "pid" returned from *waitpid*(2), or fed into ptrace calls, is the tracee's thread ID.) That is, the tracee's thread ID is reset to be the same as its process ID, which is the same as the thread group leader's thread ID.
- Then a **PTRACE_EVENT_EXEC** stop happens, if the **PTRACE_O_TRACEEXEC** option was turned on.
- If the thread group leader has reported its **PTRACE_EVENT_EXIT** stop by this time, it appears to the tracer that the dead thread leader "reappears from nowhere". (Note: the thread group leader does not report death via *WIFEXITED(status)* until there is at least one other live thread. This eliminates the possibility that the tracer will see it dying and then reappearing.) If the thread group leader was still alive, for the tracer this may look as if thread group leader returns from a different system call than it entered, or even "returned from a system call even though it was not in any system call". If the thread group leader was not traced (or was traced by a different tracer), then during *execve*(2) it will appear as if it has become a tracee of the tracer of the execing tracee.

All of the above effects are the artifacts of the thread ID change in the tracee.

The **PTRACE_O_TRACEEXEC** option is the recommended tool for dealing with this situation. First, it enables **PTRACE_EVENT_EXEC** stop, which occurs before *execve*(2) returns. In this stop, the tracer can use **PTRACE_GETEVENTMSG** to retrieve the tracee's former thread ID. (This feature was introduced in Linux 3.0.) Second, the **PTRACE_O_TRACEEXEC** option disables legacy **SIGTRAP** generation on *execve*(2).

When the tracer receives **PTRACE_EVENT_EXEC** stop notification, it is guaranteed that except this tracee and the thread group leader, no other threads from the process are alive.

On receiving the **PTRACE_EVENT_EXEC** stop notification, the tracer should clean up all its internal data structures describing the threads of this process, and retain only one data structure—one which describes the single still running tracee, with

```
thread ID == thread group ID == process ID.
```

Example: two threads call *execve*(2) at the same time:

*** we get syscall-enter-stop in thread 1: **
PID1 execve("/bin/foo", "foo" <unfinished ...>
*** we issue PTRACE_SYSCALL for thread 1 **
*** we get syscall-enter-stop in thread 2: **
PID2 execve("/bin/bar", "bar" <unfinished ...>
*** we issue PTRACE_SYSCALL for thread 2 **

```
*** we get PTRACE_EVENT_EXEC for PID0, we issue PTRACE_SYSCALL **
*** we get syscall-exit-stop for PID0: **
PID0 <... execve resumed> )             = 0
```

If the **PTRACE_O_TRACEEXEC** option is *not* in effect for the execing tracee, and if the tracee was **PTRACE_ATTACH**ed rather that **PTRACE_SEIZE**d, the kernel delivers an extra **SIGTRAP** to the tracee after *execve*(2) returns. This is an ordinary signal (similar to one which can be generated by *kill −TRAP*), not a special kind of ptrace-stop. Employing **PTRACE_GETSIGINFO** for this signal returns *si_code* set to 0 (*SI_USER*). This signal may be blocked by signal mask, and thus may be delivered (much) later.

Usually, the tracer (for example, *strace*(1)) would not want to show this extra post-execve **SIGTRAP** signal to the user, and would suppress its delivery to the tracee (if **SIGTRAP** is set to **SIG_DFL**, it is a killing signal). However, determining *which* **SIGTRAP** to suppress is not easy. Setting the **PTRACE_O_TRACEEXEC** option or using **PTRACE_SEIZE** and thus suppressing this extra **SIGTRAP** is the recommended approach.

Real parent

The ptrace API (ab)uses the standard UNIX parent/child signaling over *waitpid*(2). This used to cause the real parent of the process to stop receiving several kinds of *waitpid*(2) notifications when the child process is traced by some other process.

Many of these bugs have been fixed, but as of Linux 2.6.38 several still exist; see BUGS below.

As of Linux 2.6.38, the following is believed to work correctly:

- exit/death by signal is reported first to the tracer, then, when the tracer consumes the *waitpid*(2) result, to the real parent (to the real parent only when the whole multithreaded process exits). If the tracer and the real parent are the same process, the report is sent only once.

RETURN VALUE

On success, the **PTRACE_PEEK*** operations return the requested data (but see NOTES), the **PTRACE_SECCOMP_GET_FILTER** operation returns the number of instructions in the BPF program, the **PTRACE_GET_SYSCALL_INFO** operation returns the number of bytes available to be written by the kernel, and other operations return zero.

On error, all operations return −1, and *errno* is set to indicate the error. Since the value returned by a successful **PTRACE_PEEK*** operation may be −1, the caller must clear *errno* before the call, and then check it afterward to determine whether or not an error occurred.

ERRORS

EBUSY
(i386 only) There was an error with allocating or freeing a debug register.

EFAULT
There was an attempt to read from or write to an invalid area in the tracer's or the tracee's memory, probably because the area wasn't mapped or accessible. Unfortunately, under Linux, different variations of this fault will return **EIO** or **EFAULT** more or less arbitrarily.

EINVAL
An attempt was made to set an invalid option.

EIO *op* is invalid, or an attempt was made to read from or write to an invalid area in the tracer's or the tracee's memory, or there was a word-alignment violation, or an invalid signal was specified during a restart operation.

EPERM
The specified process cannot be traced. This could be because the tracer has insufficient privileges (the required capability is **CAP_SYS_PTRACE**); unprivileged processes cannot trace processes that they cannot send signals to or those running set-user-ID/set-group-ID programs, for obvious reasons. Alternatively, the process may already be being traced, or (before Linux 2.6.26) be *init*(1) (PID 1).

ESRCH
The specified process does not exist, or is not currently being traced by the caller, or is not stopped (for operations that require a stopped tracee).

STANDARDS

None.

HISTORY

SVr4, 4.3BSD.

Before Linux 2.6.26, *init*(1), the process with PID 1, may not be traced.

NOTES

Although arguments to **ptrace**() are interpreted according to the prototype given, glibc currently declares **ptrace**() as a variadic function with only the *op* argument fixed. It is recommended to always supply four arguments, even if the requested operation does not use them, setting unused/ignored arguments to *0L* or *(void *) 0*.

A tracees parent continues to be the tracer even if that tracer calls *execve*(2).

The layout of the contents of memory and the USER area are quite operating-system- and architecture-specific. The offset supplied, and the data returned, might not entirely match with the definition of *struct user*.

The size of a "word" is determined by the operating-system variant (e.g., for 32-bit Linux it is 32 bits).

This page documents the way the **ptrace**() call works currently in Linux. Its behavior differs significantly on other flavors of UNIX. In any case, use of **ptrace**() is highly specific to the operating system and architecture.

Ptrace access mode checking

Various parts of the kernel-user-space API (not just **ptrace**() operations), require so-called "ptrace access mode" checks, whose outcome determines whether an operation is permitted (or, in a few cases, causes a "read" operation to return sanitized data). These checks are performed in cases where one process can inspect sensitive information about, or in some cases modify the state of, another process. The checks are based on factors such as the credentials and capabilities of the two processes, whether or not the "target" process is dumpable, and the results of checks performed by any enabled Linux Security Module (LSM)—for example, SELinux, Yama, or Smack—and by the commoncap LSM (which is always invoked).

Prior to Linux 2.6.27, all access checks were of a single type. Since Linux 2.6.27, two access mode levels are distinguished:

PTRACE_MODE_READ
: For "read" operations or other operations that are less dangerous, such as: *get_robust_list*(2); *kcmp*(2); reading */proc/*pid*/auxv*, */proc/*pid*/environ*, or */proc/*pid*/stat*; or *readlink*(2) of a */proc/*pid*/ns/** file.

PTRACE_MODE_ATTACH
: For "write" operations, or other operations that are more dangerous, such as: ptrace attaching (**PTRACE_ATTACH**) to another process or calling *process_vm_writev*(2). (**PTRACE_MODE_ATTACH** was effectively the default before Linux 2.6.27.)

Since Linux 4.5, the above access mode checks are combined (ORed) with one of the following modifiers:

PTRACE_MODE_FSCREDS
: Use the caller's filesystem UID and GID (see *credentials*(7)) or effective capabilities for LSM checks.

PTRACE_MODE_REALCREDS
: Use the caller's real UID and GID or permitted capabilities for LSM checks. This was effectively the default before Linux 4.5.

Because combining one of the credential modifiers with one of the aforementioned access modes is typical, some macros are defined in the kernel sources for the combinations:

PTRACE_MODE_READ_FSCREDS
: Defined as **PTRACE_MODE_READ | PTRACE_MODE_FSCREDS**.

PTRACE_MODE_READ_REALCREDS
: Defined as **PTRACE_MODE_READ | PTRACE_MODE_REALCREDS**.

PTRACE_MODE_ATTACH_FSCREDS
Defined as **PTRACE_MODE_ATTACH | PTRACE_MODE_FSCREDS**.

PTRACE_MODE_ATTACH_REALCREDS
Defined as **PTRACE_MODE_ATTACH | PTRACE_MODE_REALCREDS**.

One further modifier can be ORed with the access mode:

PTRACE_MODE_NOAUDIT (since Linux 3.3)
Don't audit this access mode check. This modifier is employed for ptrace access mode checks (such as checks when reading */proc/*pid*/stat*) that merely cause the output to be filtered or sanitized, rather than causing an error to be returned to the caller. In these cases, accessing the file is not a security violation and there is no reason to generate a security audit record. This modifier suppresses the generation of such an audit record for the particular access check.

Note that all of the **PTRACE_MODE_*** constants described in this subsection are kernel-internal, and not visible to user space. The constant names are mentioned here in order to label the various kinds of ptrace access mode checks that are performed for various system calls and accesses to various pseudo-files (e.g., under */proc*). These names are used in other manual pages to provide a simple shorthand for labeling the different kernel checks.

The algorithm employed for ptrace access mode checking determines whether the calling process is allowed to perform the corresponding action on the target process. (In the case of opening */proc/*pid files, the "calling process" is the one opening the file, and the process with the corresponding PID is the "target process".) The algorithm is as follows:

(1) If the calling thread and the target thread are in the same thread group, access is always allowed.

(2) If the access mode specifies **PTRACE_MODE_FSCREDS**, then, for the check in the next step, employ the caller's filesystem UID and GID. (As noted in *credentials*(7), the filesystem UID and GID almost always have the same values as the corresponding effective IDs.)

Otherwise, the access mode specifies **PTRACE_MODE_REALCREDS**, so use the caller's real UID and GID for the checks in the next step. (Most APIs that check the caller's UID and GID use the effective IDs. For historical reasons, the **PTRACE_MODE_REALCREDS** check uses the real IDs instead.)

(3) Deny access if *neither* of the following is true:

- The real, effective, and saved-set user IDs of the target match the caller's user ID, *and* the real, effective, and saved-set group IDs of the target match the caller's group ID.
- The caller has the **CAP_SYS_PTRACE** capability in the user namespace of the target.

(4) Deny access if the target process "dumpable" attribute has a value other than 1 (**SUID_DUMP_USER**; see the discussion of **PR_SET_DUMPABLE** in *prctl*(2)), and the caller does not have the **CAP_SYS_PTRACE** capability in the user namespace of the target process.

(5) The kernel LSM *security_ptrace_access_check*() interface is invoked to see if ptrace access is permitted. The results depend on the LSM(s). The implementation of this interface in the commoncap LSM performs the following steps:

(5.1) If the access mode includes **PTRACE_MODE_FSCREDS**, then use the caller's *effective* capability set in the following check; otherwise (the access mode specifies **PTRACE_MODE_REALCREDS**, so) use the caller's *permitted* capability set.

(5.2) Deny access if *neither* of the following is true:

- The caller and the target process are in the same user namespace, and the caller's capabilities are a superset of the target process's *permitted* capabilities.
- The caller has the **CAP_SYS_PTRACE** capability in the target process's user namespace.

Note that the commoncap LSM does not distinguish between **PTRACE_MODE_READ** and **PTRACE_MODE_ATTACH**.

(6) If access has not been denied by any of the preceding steps, then access is allowed.

/proc/sys/kernel/yama/ptrace_scope

On systems with the Yama Linux Security Module (LSM) installed (i.e., the kernel was configured with **CONFIG_SECURITY_YAMA**), the */proc/sys/kernel/yama/ptrace_scope* file (available since Linux 3.4) can be used to restrict the ability to trace a process with **ptrace**() (and thus also the ability to use tools such as *strace*(1) and *gdb*(1)). The goal of such restrictions is to prevent attack escalation whereby a compromised process can ptrace-attach to other sensitive processes (e.g., a GPG agent or an SSH session) owned by the user in order to gain additional credentials that may exist in memory and thus expand the scope of the attack.

More precisely, the Yama LSM limits two types of operations:

- Any operation that performs a ptrace access mode **PTRACE_MODE_ATTACH** check—for example, **ptrace**() **PTRACE_ATTACH**. (See the "Ptrace access mode checking" discussion above.)
- **ptrace**() **PTRACE_TRACEME**.

A process that has the **CAP_SYS_PTRACE** capability can update the */proc/sys/kernel/yama/ptrace_scope* file with one of the following values:

0 ("classic ptrace permissions")
: No additional restrictions on operations that perform **PTRACE_MODE_ATTACH** checks (beyond those imposed by the commoncap and other LSMs).

  The use of **PTRACE_TRACEME** is unchanged.

1 ("restricted ptrace") [default value]
: When performing an operation that requires a **PTRACE_MODE_ATTACH** check, the calling process must either have the **CAP_SYS_PTRACE** capability in the user namespace of the target process or it must have a predefined relationship with the target process. By default, the predefined relationship is that the target process must be a descendant of the caller.

  A target process can employ the *prctl*(2) **PR_SET_PTRACER** operation to declare an additional PID that is allowed to perform **PTRACE_MODE_ATTACH** operations on the target. See the kernel source file *Documentation/admin−guide/LSM/Yama.rst* (or *Documentation/security/Yama.txt* before Linux 4.13) for further details.

  The use of **PTRACE_TRACEME** is unchanged.

2 ("admin-only attach")
: Only processes with the **CAP_SYS_PTRACE** capability in the user namespace of the target process may perform **PTRACE_MODE_ATTACH** operations or trace children that employ **PTRACE_TRACEME**.

3 ("no attach")
: No process may perform **PTRACE_MODE_ATTACH** operations or trace children that employ **PTRACE_TRACEME**.

  Once this value has been written to the file, it cannot be changed.

With respect to values 1 and 2, note that creating a new user namespace effectively removes the protection offered by Yama. This is because a process in the parent user namespace whose effective UID matches the UID of the creator of a child namespace has all capabilities (including **CAP_SYS_PTRACE**) when performing operations within the child user namespace (and further-removed descendants of that namespace). Consequently, when a process tries to use user namespaces to sandbox itself, it inadvertently weakens the protections offered by the Yama LSM.

C library/kernel differences

At the system call level, the **PTRACE_PEEKTEXT**, **PTRACE_PEEKDATA**, and **PTRACE_PEEKUSER** operations have a different API: they store the result at the address specified by the *data* parameter, and the return value is the error flag. The glibc wrapper function provides the API given in DESCRIPTION above, with the result being returned via the function return value.

BUGS

On hosts with Linux 2.6 kernel headers, **PTRACE_SETOPTIONS** is declared with a different value than the one for Linux 2.4. This leads to applications compiled with Linux 2.6 kernel headers failing when run on Linux 2.4. This can be worked around by redefining **PTRACE_SETOPTIONS** to **PTRACE_OLDSETOPTIONS**, if that is defined.

Group-stop notifications are sent to the tracer, but not to real parent. Last confirmed on 2.6.38.6.

If a thread group leader is traced and exits by calling *_exit*(2), a **PTRACE_EVENT_EXIT** stop will happen for it (if requested), but the subsequent **WIFEXITED** notification will not be delivered until all other threads exit. As explained above, if one of other threads calls *execve*(2), the death of the thread group leader will *never* be reported. If the execed thread is not traced by this tracer, the tracer will never know that *execve*(2) happened. One possible workaround is to **PTRACE_DETACH** the thread group leader instead of restarting it in this case. Last confirmed on 2.6.38.6.

A **SIGKILL** signal may still cause a **PTRACE_EVENT_EXIT** stop before actual signal death. This may be changed in the future; **SIGKILL** is meant to always immediately kill tasks even under ptrace. Last confirmed on Linux 3.13.

Some system calls return with **EINTR** if a signal was sent to a tracee, but delivery was suppressed by the tracer. (This is very typical operation: it is usually done by debuggers on every attach, in order to not introduce a bogus **SIGSTOP**). As of Linux 3.2.9, the following system calls are affected (this list is likely incomplete): *epoll_wait*(2), and *read*(2) from an *inotify*(7) file descriptor. The usual symptom of this bug is that when you attach to a quiescent process with the command

```
strace -p <process-ID>
```

then, instead of the usual and expected one-line output such as

```
restart_syscall(<... resuming interrupted call ...>_
```

or

```
select(6, [5], NULL, [5], NULL_
```

('_' denotes the cursor position), you observe more than one line. For example:

```
clock_gettime(CLOCK_MONOTONIC, {15370, 690928118}) = 0
epoll_wait(4,_
```

What is not visible here is that the process was blocked in *epoll_wait*(2) before *strace*(1) has attached to it. Attaching caused *epoll_wait*(2) to return to user space with the error **EINTR**. In this particular case, the program reacted to **EINTR** by checking the current time, and then executing *epoll_wait*(2) again. (Programs which do not expect such "stray" **EINTR** errors may behave in an unintended way upon an *strace*(1) attach.)

Contrary to the normal rules, the glibc wrapper for **ptrace**() can set *errno* to zero.

SEE ALSO

gdb(1), *ltrace*(1), *strace*(1), *clone*(2), *execve*(2), *fork*(2), *gettid*(2), *prctl*(2), *seccomp*(2), *sigaction*(2), *tgkill*(2), *vfork*(2), *waitpid*(2), *exec*(3), *capabilities*(7), *signal*(7)

NAME

query_module − query the kernel for various bits pertaining to modules

SYNOPSIS

#include <linux/module.h>

[[deprecated]] int query_module(const char ****name*, int** *which***,**
void *buf* **[.**bufsize**], size_t** *bufsize***,**
size_t ****ret*);**

DESCRIPTION

Note: This system call is present only before Linux 2.6.

query_module() requests information from the kernel about loadable modules. The returned information is placed in the buffer pointed to by *buf*. The caller must specify the size of *buf* in *bufsize*. The precise nature and format of the returned information depend on the operation specified by *which*. Some operations require *name* to identify a currently loaded module, some allow *name* to be NULL, indicating the kernel proper.

The following values can be specified for *which*:

0 Returns success, if the kernel supports **query_module**(). Used to probe for availability of the system call.

QM_MODULES
Returns the names of all loaded modules. The returned buffer consists of a sequence of null-terminated strings; *ret* is set to the number of modules.

QM_DEPS
Returns the names of all modules used by the indicated module. The returned buffer consists of a sequence of null-terminated strings; *ret* is set to the number of modules.

QM_REFS
Returns the names of all modules using the indicated module. This is the inverse of **QM_DEPS**. The returned buffer consists of a sequence of null-terminated strings; *ret* is set to the number of modules.

QM_SYMBOLS
Returns the symbols and values exported by the kernel or the indicated module. The returned buffer is an array of structures of the following form

```
struct module_symbol {
    unsigned long value;
    unsigned long name;
};
```

followed by null-terminated strings. The value of *name* is the character offset of the string relative to the start of *buf*; *ret* is set to the number of symbols.

QM_INFO
Returns miscellaneous information about the indicated module. The output buffer format is:

```
struct module_info {
    unsigned long address;
    unsigned long size;
    unsigned long flags;
};
```

where *address* is the kernel address at which the module resides, *size* is the size of the module in bytes, and *flags* is a mask of **MOD_RUNNING**, **MOD_AUTOCLEAN**, and so on, that indicates the current status of the module (see the Linux kernel source file *include/linux/module.h*). *ret* is set to the size of the *module_info* structure.

RETURN VALUE

On success, zero is returned. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
At least one of *name*, *buf*, or *ret* was outside the program's accessible address space.

EINVAL
Invalid *which*; or *name* is NULL (indicating "the kernel"), but this is not permitted with the specified value of *which*.

ENOENT
No module by that *name* exists.

ENOSPC
The buffer size provided was too small. *ret* is set to the minimum size needed.

ENOSYS
query_module() is not supported in this version of the kernel (e.g., Linux 2.6 or later).

STANDARDS
Linux.

VERSIONS
Removed in Linux 2.6.

Some of the information that was formerly available via **query_module**() can be obtained from */proc/modules*, */proc/kallsyms*, and the files under the directory */sys/module*.

The **query_module**() system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc does export an ABI for this system call. Therefore, in order to employ this system call, it is sufficient to manually declare the interface in your code; alternatively, you can invoke the system call using *syscall*(2).

SEE ALSO
create_module(2), *delete_module*(2), *get_kernel_syms*(2), *init_module*(2), *lsmod*(8), *modinfo*(8)

NAME

quotactl − manipulate disk quotas

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/quota.h>
#include <xfs/xqm.h> /* Definition of **Q_X*** and **XFS_QUOTA_*** constants
(or **<linux/dqblk_xfs.h>**; see NOTES) */

int quotactl(int *op***, const char *_Nullable** *special***, int** *id***,**
caddr_t *addr***);**

DESCRIPTION

The quota system can be used to set per-user, per-group, and per-project limits on the amount of disk space used on a filesystem. For each user and/or group, a soft limit and a hard limit can be set for each filesystem. The hard limit can't be exceeded. The soft limit can be exceeded, but warnings will ensue. Moreover, the user can't exceed the soft limit for more than grace period duration (one week by default) at a time; after this, the soft limit counts as a hard limit.

The **quotactl**() call manipulates disk quotas. The *op* argument indicates an operation to be applied to the user or group ID specified in *id*. To initialize the *op* argument, use the *QCMD(subop, type)* macro. The *type* value is either **USRQUOTA**, for user quotas, **GRPQUOTA**, for group quotas, or (since Linux 4.1) **PRJQUOTA**, for project quotas. The *subop* value is described below.

The *special* argument is a pointer to a null-terminated string containing the pathname of the (mounted) block special device for the filesystem being manipulated.

The *addr* argument is the address of an optional, operation-specific, data structure that is copied in or out of the system. The interpretation of *addr* is given with each operation below.

The *subop* value is one of the following operations:

Q_QUOTAON
Turn on quotas for a filesystem. The *id* argument is the identification number of the quota format to be used. Currently, there are three supported quota formats:

QFMT_VFS_OLD
The original quota format.

QFMT_VFS_V0
The standard VFS v0 quota format, which can handle 32-bit UIDs and GIDs and quota limits up to 2^42 bytes and 2^32 inodes.

QFMT_VFS_V1
A quota format that can handle 32-bit UIDs and GIDs and quota limits of 2^63 − 1 bytes and 2^63 − 1 inodes.

The *addr* argument points to the pathname of a file containing the quotas for the filesystem. The quota file must exist; it is normally created with the *quotacheck*(8) program

Quota information can be also stored in hidden system inodes for ext4, XFS, and other filesystems if the filesystem is configured so. In this case, there are no visible quota files and there is no need to use *quotacheck*(8)Quota information is always kept consistent by the filesystem and the **Q_QUOTAON** operation serves only to enable enforcement of quota limits. The presence of hidden system inodes with quota information is indicated by the **DQF_SYS_FILE** flag in the *dqi_flags* field returned by the **Q_GETINFO** operation.

This operation requires privilege (**CAP_SYS_ADMIN**).

Q_QUOTAOFF
Turn off quotas for a filesystem. The *addr* and *id* arguments are ignored. This operation requires privilege (**CAP_SYS_ADMIN**).

Q_GETQUOTA
Get disk quota limits and current usage for user or group *id*. The *addr* argument is a pointer to a *dqblk* structure defined in <*sys/quota.h*> as follows:

```
/* uint64_t is an unsigned 64-bit integer;
   uint32_t is an unsigned 32-bit integer */

struct dqblk {      /* Definition since Linux 2.4.22 */
    uint64_t dqb_bhardlimit;  /* Absolute limit on disk
                                 quota blocks alloc */
    uint64_t dqb_bsoftlimit;  /* Preferred limit on
                                 disk quota blocks */
    uint64_t dqb_curspace;    /* Current occupied space
                                 (in bytes) */
    uint64_t dqb_ihardlimit;  /* Maximum number of
                                 allocated inodes */
    uint64_t dqb_isoftlimit;  /* Preferred inode limit */
    uint64_t dqb_curinodes;   /* Current number of
                                 allocated inodes */
    uint64_t dqb_btime;       /* Time limit for excessive
                                 disk use */
    uint64_t dqb_itime;       /* Time limit for excessive
                                 files */
    uint32_t dqb_valid;       /* Bit mask of QIF_*
                                 constants */
};

/* Flags in dqb_valid that indicate which fields in
   dqblk structure are valid. */

#define QIF_BLIMITS   1
#define QIF_SPACE     2
#define QIF_ILIMITS   4
#define QIF_INODES    8
#define QIF_BTIME     16
#define QIF_ITIME     32
#define QIF_LIMITS    (QIF_BLIMITS | QIF_ILIMITS)
#define QIF_USAGE     (QIF_SPACE | QIF_INODES)
#define QIF_TIMES     (QIF_BTIME | QIF_ITIME)
#define QIF_ALL       (QIF_LIMITS | QIF_USAGE | QIF_TIMES)
```

The *dqb_valid* field is a bit mask that is set to indicate the entries in the *dqblk* structure that are valid. Currently, the kernel fills in all entries of the *dqblk* structure and marks them as valid in the *dqb_valid* field. Unprivileged users may retrieve only their own quotas; a privileged user (**CAP_SYS_ADMIN**) can retrieve the quotas of any user.

Q_GETNEXTQUOTA (since Linux 4.6)

This operation is the same as **Q_GETQUOTA**, but it returns quota information for the next ID greater than or equal to *id* that has a quota set.

The *addr* argument is a pointer to a *nextdqblk* structure whose fields are as for the *dqblk*, except for the addition of a *dqb_id* field that is used to return the ID for which quota information is being returned:

```
struct nextdqblk {
    uint64_t dqb_bhardlimit;
    uint64_t dqb_bsoftlimit;
    uint64_t dqb_curspace;
    uint64_t dqb_ihardlimit;
    uint64_t dqb_isoftlimit;
    uint64_t dqb_curinodes;
    uint64_t dqb_btime;
    uint64_t dqb_itime;
    uint32_t dqb_valid;
    uint32_t dqb_id;
```

```
};
```

Q_SETQUOTA
Set quota information for user or group *id*, using the information supplied in the *dqblk* structure pointed to by *addr*. The *dqb_valid* field of the *dqblk* structure indicates which entries in the structure have been set by the caller. This operation supersedes the **Q_SETQLIM** and **Q_SETUSE** operations in the previous quota interfaces. This operation requires privilege (**CAP_SYS_ADMIN**).

Q_GETINFO (since Linux 2.4.22)
Get information (like grace times) about quotafile. The *addr* argument should be a pointer to a *dqinfo* structure. This structure is defined in <*sys/quota.h*> as follows:

```
/* uint64_t is an unsigned 64-bit integer;
   uint32_t is an unsigned 32-bit integer */

struct dqinfo {         /* Defined since Linux 2.4.22 */
    uint64_t dqi_bgrace;  /* Time before block soft limit
                             becomes hard limit */
    uint64_t dqi_igrace;  /* Time before inode soft limit
                             becomes hard limit */
    uint32_t dqi_flags;   /* Flags for quotafile
                             (DQF_*) */
    uint32_t dqi_valid;
};

/* Bits for dqi_flags */

/* Quota format QFMT_VFS_OLD */

#define DQF_ROOT_SQUASH (1 << 0) /* Root squash enabled */
             /* Before Linux v4.0, this had been defined
                privately as V1_DQF_RSQUASH */

/* Quota format QFMT_VFS_V0 / QFMT_VFS_V1 */

#define DQF_SYS_FILE    (1 << 16)   /* Quota stored in
                                       a system file */

/* Flags in dqi_valid that indicate which fields in
   dqinfo structure are valid. */

#define IIF_BGRACE  1
#define IIF_IGRACE  2
#define IIF_FLAGS   4
#define IIF_ALL     (IIF_BGRACE | IIF_IGRACE | IIF_FLAGS)
```

The *dqi_valid* field in the *dqinfo* structure indicates the entries in the structure that are valid. Currently, the kernel fills in all entries of the *dqinfo* structure and marks them all as valid in the *dqi_valid* field. The *id* argument is ignored.

Q_SETINFO (since Linux 2.4.22)
Set information about quotafile. The *addr* argument should be a pointer to a *dqinfo* structure. The *dqi_valid* field of the *dqinfo* structure indicates the entries in the structure that have been set by the caller. This operation supersedes the **Q_SETGRACE** and **Q_SETFLAGS** operations in the previous quota interfaces. The *id* argument is ignored. This operation requires privilege (**CAP_SYS_ADMIN**).

Q_GETFMT (since Linux 2.4.22)
Get quota format used on the specified filesystem. The *addr* argument should be a pointer to a 4-byte buffer where the format number will be stored.

Q_SYNC
Update the on-disk copy of quota usages for a filesystem. If *special* is NULL, then all filesystems with active quotas are sync'ed. The *addr* and *id* arguments are ignored.

Q_GETSTATS (supported up to Linux 2.4.21)
Get statistics and other generic information about the quota subsystem. The *addr* argument should be a pointer to a *dqstats* structure in which data should be stored. This structure is defined in <*sys/quota.h*>. The *special* and *id* arguments are ignored.

This operation is obsolete and was removed in Linux 2.4.22. Files in */proc/sys/fs/quota/* carry the information instead.

For XFS filesystems making use of the XFS Quota Manager (XQM), the above operations are bypassed and the following operations are used:

Q_XQUOTAON
Turn on quotas for an XFS filesystem. XFS provides the ability to turn on/off quota limit enforcement with quota accounting. Therefore, XFS expects *addr* to be a pointer to an *unsigned int* that contains a bitwise combination of the following flags (defined in <*xfs/xqm.h*>):

```
XFS_QUOTA_UDQ_ACCT  /* User quota accounting */
XFS_QUOTA_UDQ_ENFD  /* User quota limits enforcement */
XFS_QUOTA_GDQ_ACCT  /* Group quota accounting */
XFS_QUOTA_GDQ_ENFD  /* Group quota limits enforcement */
XFS_QUOTA_PDQ_ACCT  /* Project quota accounting */
XFS_QUOTA_PDQ_ENFD  /* Project quota limits enforcement */
```

This operation requires privilege (**CAP_SYS_ADMIN**). The *id* argument is ignored.

Q_XQUOTAOFF
Turn off quotas for an XFS filesystem. As with **Q_QUOTAON**, XFS filesystems expect a pointer to an *unsigned int* that specifies whether quota accounting and/or limit enforcement need to be turned off (using the same flags as for **Q_XQUOTAON** operation). This operation requires privilege (**CAP_SYS_ADMIN**). The *id* argument is ignored.

Q_XGETQUOTA
Get disk quota limits and current usage for user *id*. The *addr* argument is a pointer to an *fs_disk_quota* structure, which is defined in <*xfs/xqm.h*> as follows:

```
/* All the blk units are in BBs (Basic Blocks) of
   512 bytes. */

#define FS_DQUOT_VERSION  1  /* fs_disk_quota.d_version */

#define XFS_USER_QUOTA    (1<<0)  /* User quota type */
#define XFS_PROJ_QUOTA    (1<<1)  /* Project quota type */
#define XFS_GROUP_QUOTA   (1<<2)  /* Group quota type */

struct fs_disk_quota {
    int8_t   d_version;   /* Version of this structure */
    int8_t   d_flags;     /* XFS_{USER,PROJ,GROUP}_QUOTA */
    uint16_t d_fieldmask; /* Field specifier */
    uint32_t d_id;        /* User, project, or group ID */
    uint64_t d_blk_hardlimit; /* Absolute limit on
                                 disk blocks */
    uint64_t d_blk_softlimit; /* Preferred limit on
                                 disk blocks */
    uint64_t d_ino_hardlimit; /* Maximum # allocated
                                 inodes */
    uint64_t d_ino_softlimit; /* Preferred inode limit */
    uint64_t d_bcount;    /* # disk blocks owned by
                             the user */
    uint64_t d_icount;    /* # inodes owned by the user */
    int32_t  d_itimer;    /* Zero if within inode limits */
```

```
                            /* If not, we refuse service */
    int32_t  d_btimer;      /* Similar to above; for
                               disk blocks */
    uint16_t d_iwarns;      /* # warnings issued with
                               respect to # of inodes */
    uint16_t d_bwarns;      /* # warnings issued with
                               respect to disk blocks */
    int32_t  d_padding2;    /* Padding - for future use */
    uint64_t d_rtb_hardlimit; /* Absolute limit on realtime
                                 (RT) disk blocks */
    uint64_t d_rtb_softlimit; /* Preferred limit on RT
                                 disk blocks */
    uint64_t d_rtbcount;    /* # realtime blocks owned */
    int32_t  d_rtbtimer;    /* Similar to above; for RT
                               disk blocks */
    uint16_t d_rtbwarns;    /* # warnings issued with
                               respect to RT disk blocks */
    int16_t  d_padding3;    /* Padding - for future use */
    char     d_padding4[8];   /* Yet more padding */
};
```

Unprivileged users may retrieve only their own quotas; a privileged user (**CAP_SYS_ADMIN**) may retrieve the quotas of any user.

Q_XGETNEXTQUOTA (since Linux 4.6)
This operation is the same as **Q_XGETQUOTA**, but it returns (in the *fs_disk_quota* structure pointed by *addr*) quota information for the next ID greater than or equal to *id* that has a quota set. Note that since *fs_disk_quota* already has *q_id* field, no separate structure type is needed (in contrast with **Q_GETQUOTA** and **Q_GETNEXTQUOTA** operations)

Q_XSETQLIM
Set disk quota limits for user *id*. The *addr* argument is a pointer to an *fs_disk_quota* structure. This operation requires privilege (**CAP_SYS_ADMIN**).

Q_XGETQSTAT
Returns XFS filesystem-specific quota information in the *fs_quota_stat* structure pointed by *addr*. This is useful for finding out how much space is used to store quota information, and also to get the quota on/off status of a given local XFS filesystem. The *fs_quota_stat* structure itself is defined as follows:

```
#define FS_QSTAT_VERSION 1  /* fs_quota_stat.qs_version */

struct fs_qfilestat {
    uint64_t qfs_ino;       /* Inode number */
    uint64_t qfs_nblks;     /* Number of BBs
                               512-byte-blocks */
    uint32_t qfs_nextents;  /* Number of extents */
};

struct fs_quota_stat {
    int8_t   qs_version; /* Version number for
                            future changes */
    uint16_t qs_flags; /* XFS_QUOTA_{U,P,G}DQ_{ACCT,ENFD} */
    int8_t   qs_pad;   /* Unused */
    struct fs_qfilestat qs_uquota;  /* User quota storage
                                       information */
    struct fs_qfilestat qs_gquota;  /* Group quota storage
                                       information */
    uint32_t qs_incoredqs;   /* Number of dquots in core */
    int32_t  qs_btimelimit;  /* Limit for blocks timer */
    int32_t  qs_itimelimit;  /* Limit for inodes timer */
    int32_t  qs_rtbtimelimit;/* Limit for RT
```

```
                                    blocks timer */
    uint16_t qs_bwarnlimit;  /* Limit for # of warnings */
    uint16_t qs_iwarnlimit;  /* Limit for # of warnings */
};
```

The *id* argument is ignored.

Q_XGETQSTATV

Returns XFS filesystem-specific quota information in the *fs_quota_statv* pointed to by *addr*. This version of the operation uses a structure with proper versioning support, along with appropriate layout (all fields are naturally aligned) and padding to avoiding special compat handling; it also provides the ability to get statistics regarding the project quota file. The *fs_quota_statv* structure itself is defined as follows:

```
#define FS_QSTATV_VERSION1 1 /* fs_quota_statv.qs_version */

struct fs_qfilestatv {
    uint64_t qfs_ino;       /* Inode number */
    uint64_t qfs_nblks;     /* Number of BBs
                               512-byte-blocks */
    uint32_t qfs_nextents;  /* Number of extents */
    uint32_t qfs_pad;       /* Pad for 8-byte alignment */
};

struct fs_quota_statv {
    int8_t   qs_version;    /* Version for future
                               changes */
    uint8_t  qs_pad1;       /* Pad for 16-bit alignment */
    uint16_t qs_flags;      /* XFS_QUOTA_.* flags */
    uint32_t qs_incoredqs;  /* Number of dquots incore */
    struct fs_qfilestatv qs_uquota;  /* User quota
                                        information */
    struct fs_qfilestatv qs_gquota;  /* Group quota
                                        information */
    struct fs_qfilestatv qs_pquota;  /* Project quota
                                        information */
    int32_t  qs_btimelimit;   /* Limit for blocks timer */
    int32_t  qs_itimelimit;   /* Limit for inodes timer */
    int32_t  qs_rtbtimelimit; /* Limit for RT blocks
                                 timer */
    uint16_t qs_bwarnlimit;   /* Limit for # of warnings */
    uint16_t qs_iwarnlimit;   /* Limit for # of warnings */
    uint64_t qs_pad2[8];      /* For future proofing */
};
```

The *qs_version* field of the structure should be filled with the version of the structure supported by the callee (for now, only *FS_QSTAT_VERSION1* is supported). The kernel will fill the structure in accordance with version provided. The *id* argument is ignored.

Q_XQUOTARM (buggy until Linux 3.16)

Free the disk space taken by disk quotas. The *addr* argument should be a pointer to an *unsigned int* value containing flags (the same as in *d_flags* field of *fs_disk_quota* structure) which identify what types of quota should be removed. (Note that the quota type passed in the *op* argument is ignored, but should remain valid in order to pass preliminary quotactl syscall handler checks.)

Quotas must have already been turned off. The *id* argument is ignored.

Q_XQUOTASYNC (since Linux 2.6.15; no-op since Linux 3.4)

This operation was an XFS quota equivalent to **Q_SYNC**, but it is no-op since Linux 3.4, as *sync*(1) writes quota information to disk now (in addition to the other filesystem metadata that it writes out). The *special*, *id* and *addr* arguments are ignored.

RETURN VALUE

On success, **quotactl**() returns 0; on error −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: *op* is **Q_QUOTAON**, and the quota file pointed to by *addr* exists, but is not a regular file or is not on the filesystem pointed to by *special*.

EBUSY
: *op* is **Q_QUOTAON**, but another **Q_QUOTAON** had already been performed.

EFAULT
: *addr* or *special* is invalid.

EINVAL
: *op* or *type* is invalid.

EINVAL
: *op* is **Q_QUOTAON**, but the specified quota file is corrupted.

EINVAL (since Linux 5.5)
: *op* is **Q_XQUOTARM**, but *addr* does not point to valid quota types.

ENOENT
: The file specified by *special* or *addr* does not exist.

ENOSYS
: The kernel has not been compiled with the **CONFIG_QUOTA** option.

ENOTBLK
: *special* is not a block device.

EPERM
: The caller lacked the required privilege (**CAP_SYS_ADMIN**) for the specified operation.

ERANGE
: *op* is **Q_SETQUOTA**, but the specified limits are out of the range allowed by the quota format.

ESRCH
: No disk quota is found for the indicated user. Quotas have not been turned on for this filesystem.

ESRCH
: *op* is **Q_QUOTAON**, but the specified quota format was not found.

ESRCH
: *op* is **Q_GETNEXTQUOTA** or **Q_XGETNEXTQUOTA**, but there is no ID greater than or equal to *id* that has an active quota.

NOTES

Instead of <*xfs/xqm.h*> one can use <*linux/dqblk_xfs.h*>, taking into account that there are several naming discrepancies:

- Quota enabling flags (of format **XFS_QUOTA_[UGP]DQ_{ACCT,ENFD}**) are defined without a leading "X", as **FS_QUOTA_[UGP]DQ_{ACCT,ENFD}**.
- The same is true for **XFS_{USER,GROUP,PROJ}_QUOTA** quota type flags, which are defined as **FS_{USER,GROUP,PROJ}_QUOTA**.
- The *dqblk_xfs.h* header file defines its own **XQM_USRQUOTA**, **XQM_GRPQUOTA**, and **XQM_PRJQUOTA** constants for the available quota types, but their values are the same as for constants without the **XQM_** prefix.

SEE ALSO

quota(1), *getrlimit*(2), *quotacheck*(8), *quotaon*(8)

NAME

read − read from a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

ssize_t read(int *fd***, void** *buf***[.***count***], size_t** *count***);**

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. See also NOTES.

On error, −1 is returned, and *errno* is set to indicate the error. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS

EAGAIN
: The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. See *open*(2) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**
: The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF
: *fd* is not a valid file descriptor or is not open for reading.

EFAULT
: *buf* is outside your accessible address space.

EINTR
: The call was interrupted by a signal before any data was read; see *signal*(7).

EINVAL
: *fd* is attached to an object which is unsuitable for reading; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EINVAL
: *fd* was created via a call to *timerfd_create*(2) and the wrong size buffer was given to **read**(); see *timerfd_create*(2) for further information.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking **SIGTTIN** or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape. A further possible cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost*

locks section of *fcntl*(2) for further details.

EISDIR
fd refers to a directory.

Other errors may occur, depending on the object connected to *fd*.

STANDARDS
POSIX.1-2008.

HISTORY
SVr4, 4.3BSD, POSIX.1-2001.

NOTES
On Linux, **read**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

On NFS filesystems, reading small amounts of data will update the timestamp only the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave *st_atime* (last file access time) updates to the server, and client side reads satisfied from the client's cache will not cause *st_atime* updates on the server as there are no server-side reads. UNIX semantics can be obtained by disabling client-side attribute caching, but in most situations this will substantially increase server load and decrease performance.

BUGS
According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

> All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **read**() and *readv*(2). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, before Linux 3.14, this was not the case: if two processes that share an open file description (see *open*(2)) perform a **read**() (or *readv*(2)) at the same time, then the I/O operations were not atomic with respect to updating the file offset, with the result that the reads in the two processes might (incorrectly) overlap in the blocks of data that they obtained. This problem was fixed in Linux 3.14.

SEE ALSO
close(2), *fcntl*(2), *ioctl*(2), *lseek*(2), *open*(2), *pread*(2), *readdir*(2), *readlink*(2), *readv*(2), *select*(2), *write*(2), *fread*(3)

NAME

readahead − initiate file readahead into page cache

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>

ssize_t readahead(int fd, off_t offset, size_t count);
```

DESCRIPTION

readahead() initiates readahead on a file so that subsequent reads from that file will be satisfied from the cache, and not block on disk I/O (assuming the readahead was initiated early enough and that other activity on the system did not in the meantime flush pages from the cache).

The *fd* argument is a file descriptor identifying the file which is to be read. The *offset* argument specifies the starting point from which data is to be read and *count* specifies the number of bytes to be read. I/O is performed in whole pages, so that *offset* is effectively rounded down to a page boundary and bytes are read up to the next page boundary greater than or equal to *(offset+count)*. **readahead**() does not read beyond the end of the file. The file offset of the open file description referred to by the file descriptor *fd* is left unchanged.

RETURN VALUE

On success, **readahead**() returns 0; on failure, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF
: *fd* is not a valid file descriptor or is not open for reading.

EINVAL
: *fd* does not refer to a file type to which **readahead**() can be applied.

VERSIONS

On some 32-bit architectures, the calling signature for this system call differs, for the reasons described in *syscall*(2).

STANDARDS

Linux.

HISTORY

Linux 2.4.13, glibc 2.3.

NOTES

_FILE_OFFSET_BITS should be defined to be 64 in code that uses a pointer to **readahead**, if the code is intended to be portable to traditional 32-bit x86 and ARM platforms where **off_t**'s width defaults to 32 bits.

BUGS

readahead() attempts to schedule the reads in the background and return immediately. However, it may block while it reads the filesystem metadata needed to locate the requested blocks. This occurs frequently with ext[234] on large files using indirect blocks instead of extents, giving the appearance that the call blocks until the requested data has been read.

SEE ALSO

lseek(2), *madvise*(2), *mmap*(2), *posix_fadvise*(2), *read*(2)

NAME

readdir − read directory entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_readdir, unsigned int *fd***,**
struct old_linux_dirent ***dirp*, unsigned int** *count***);**

Note: There is no definition of **struct old_linux_dirent**; see NOTES.

DESCRIPTION

This is not the function you are interested in. Look at *readdir*(3) for the POSIX conforming C library interface. This page documents the bare kernel system call interface, which is superseded by *getdents*(2).

readdir() reads one *old_linux_dirent* structure from the directory referred to by the file descriptor *fd* into the buffer pointed to by *dirp*. The argument *count* is ignored; at most one *old_linux_dirent* structure is read.

The *old_linux_dirent* structure is declared (privately in Linux kernel file **fs/readdir.c**) as follows:

```
struct old_linux_dirent {
    unsigned long d_ino;     /* inode number */
    unsigned long d_offset;  /* offset to this old_linux_dirent */
    unsigned short d_namlen; /* length of this d_name */
    char  d_name[1];         /* filename (null-terminated) */
}
```

d_ino is an inode number. *d_offset* is the distance from the start of the directory to this *old_linux_dirent*. *d_reclen* is the size of *d_name*, not counting the terminating null byte ('\0'). *d_name* is a null-terminated filename.

RETURN VALUE

On success, 1 is returned. On end of directory, 0 is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
Invalid file descriptor *fd*.

EFAULT
Argument points outside the calling process's address space.

EINVAL
Result buffer is too small.

ENOENT
No such directory.

ENOTDIR
File descriptor does not refer to a directory.

VERSIONS

You will need to define the *old_linux_dirent* structure yourself. However, probably you should use *readdir*(3) instead.

This system call does not exist on x86-64.

STANDARDS

Linux.

SEE ALSO

getdents(2), *readdir*(3)

NAME

readlink, readlinkat − read value of a symbolic link

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

ssize_t readlink(const char *restrict pathname, char *restrict buf,
                 size_t bufsiz);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

ssize_t readlinkat(int dirfd, const char *restrict pathname,
                   char *restrict buf, size_t bufsiz);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

readlink():
  _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200112L
    || /* glibc <= 2.19: */ _BSD_SOURCE

readlinkat():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _ATFILE_SOURCE

DESCRIPTION

readlink() places the contents of the symbolic link *pathname* in the buffer *buf*, which has size *bufsiz*. **readlink**() does not append a terminating null byte to *buf*. It will (silently) truncate the contents (to a length of *bufsiz* characters), in case the buffer is too small to hold all of the contents.

readlinkat()

The **readlinkat**() system call operates in exactly the same way as **readlink**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **readlink**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *readlink*())

If *pathname* is absolute, then *dirfd* is ignored.

Since Linux 2.6.39, *pathname* can be an empty string, in which case the call operates on the symbolic link referred to by *dirfd* (which should have been obtained using *open*(2) with the **O_PATH** and **O_NOFOLLOW** flags).

See *openat*(2) for an explanation of the need for **readlinkat**().

RETURN VALUE

On success, these calls return the number of bytes placed in *buf*. (If the returned value equals *bufsiz*, then truncation may have occurred.) On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
: Search permission is denied for a component of the path prefix. (See also *path_resolution*(7).)

EBADF
: (**readlinkat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
: *buf* extends outside the process's allocated address space.

EINVAL
bufsiz is not positive.

EINVAL
The named file (i.e., the final filename component of *pathname*) is not a symbolic link.

EIO An I/O error occurred while reading from the filesystem.

ELOOP
Too many symbolic links were encountered in translating the pathname.

ENAMETOOLONG
A pathname, or a component of a pathname, was too long.

ENOENT
The named file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

ENOTDIR
(**readlinkat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

STANDARDS

POSIX.1-2008.

HISTORY

readlink()
4.4BSD (first appeared in 4.2BSD), POSIX.1-2001, POSIX.1-2008.

readlinkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

Up to and including glibc 2.4, the return type of **readlink**() was declared as *int*. Nowadays, the return type is declared as *ssize_t*, as (newly) required in POSIX.1-2001.

glibc
On older kernels where **readlinkat**() is unavailable, the glibc wrapper function falls back to the use of **readlink**(). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

NOTES

Using a statically sized buffer might not provide enough room for the symbolic link contents. The required size for the buffer can be obtained from the *stat.st_size* value returned by a call to *lstat*(2) on the link. However, the number of bytes written by **readlink**() and **readlinkat**() should be checked to make sure that the size of the symbolic link did not increase between the calls. Dynamically allocating the buffer for **readlink**() and **readlinkat**() also addresses a common portability problem when using **PATH_MAX** for the buffer size, as this constant is not guaranteed to be defined per POSIX if the system does not have such limit.

EXAMPLES

The following program allocates the buffer needed by **readlink**() dynamically from the information provided by *lstat*(2), falling back to a buffer of size **PATH_MAX** in cases where *lstat*(2) reports a size of zero.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
```

```
        char         *buf;
        ssize_t      nbytes, bufsiz;
        struct stat  sb;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        if (lstat(argv[1], &sb) == -1) {
            perror("lstat");
            exit(EXIT_FAILURE);
        }

        /* Add one to the link size, so that we can determine whether
           the buffer returned by readlink() was truncated. */

        bufsiz = sb.st_size + 1;

        /* Some magic symlinks under (for example) /proc and /sys
           report 'st_size' as zero. In that case, take PATH_MAX as
           a "good enough" estimate. */

        if (sb.st_size == 0)
            bufsiz = PATH_MAX;

        buf = malloc(bufsiz);
        if (buf == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        nbytes = readlink(argv[1], buf, bufsiz);
        if (nbytes == -1) {
            perror("readlink");
            exit(EXIT_FAILURE);
        }

        /* Print only 'nbytes' of 'buf', as it doesn't contain a terminating
           null byte ('\0'). */
        printf("'%s' points to '%.*s'\n", argv[1], (int) nbytes, buf);

        /* If the return value was equal to the buffer size, then
           the link target was larger than expected (perhaps because the
           target was changed between the call to lstat() and the call to
           readlink()). Warn the user that the returned target may have
           been truncated. */

        if (nbytes == bufsiz)
            printf("(Returned buffer may have been truncated)\n");

        free(buf);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

readlink(1), *lstat*(2), *stat*(2), *symlink*(2), *realpath*(3), *path_resolution*(7), *symlink*(7)

NAME

readv, writev, preadv, pwritev, preadv2, pwritev2 − read or write data into multiple buffers

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

ssize_t preadv(int fd, const struct iovec *iov, int iovcnt,
               off_t offset);
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt,
               off_t offset);

ssize_t preadv2(int fd, const struct iovec *iov, int iovcnt,
               off_t offset, int flags);
ssize_t pwritev2(int fd, const struct iovec *iov, int iovcnt,
               off_t offset, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

preadv(), **pwritev**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **readv**() system call reads *iovcnt* buffers from the file associated with the file descriptor *fd* into the buffers described by *iov* ("scatter input").

The **writev**() system call writes *iovcnt* buffers of data described by *iov* to the file associated with the file descriptor *fd* ("gather output").

The pointer *iov* points to an array of *iovec* structures, described in *iovec*(3type).

The **readv**() system call works just like *read*(2) except that multiple buffers are filled.

The **writev**() system call works just like *write*(2) except that multiple buffers are written out.

Buffers are processed in array order. This means that **readv**() completely fills *iov[0]* before proceeding to *iov[1]*, and so on. (If there is insufficient data, then not all buffers pointed to by *iov* may be filled.) Similarly, **writev**() writes out the entire contents of *iov[0]* before proceeding to *iov[1]*, and so on.

The data transfers performed by **readv**() and **writev**() are atomic: the data written by **writev**() is written as a single block that is not intermingled with output from writes in other processes; analogously, **readv**() is guaranteed to read a contiguous block of data from the file, regardless of read operations performed in other threads or processes that have file descriptors referring to the same open file description (see *open*(2)).

preadv() and pwritev()

The **preadv**() system call combines the functionality of **readv**() and *pread*(2). It performs the same task as **readv**(), but adds a fourth argument, *offset*, which specifies the file offset at which the input operation is to be performed.

The **pwritev**() system call combines the functionality of **writev**() and *pwrite*(2). It performs the same task as **writev**(), but adds a fourth argument, *offset*, which specifies the file offset at which the output operation is to be performed.

The file offset is not changed by these system calls. The file referred to by *fd* must be capable of seeking.

preadv2() and pwritev2()

These system calls are similar to **preadv**() and **pwritev**() calls, but add a fifth argument, *flags*, which modifies the behavior on a per-call basis.

Unlike **preadv**() and **pwritev**(), if the *offset* argument is −1, then the current file offset is used and

updated.

The *flags* argument contains a bitwise OR of zero or more of the following flags:

RWF_DSYNC (since Linux 4.7)
Provide a per-write equivalent of the **O_DSYNC** *open*(2) flag. This flag is meaningful only for **pwritev2**(), and its effect applies only to the data range written by the system call.

RWF_HIPRI (since Linux 4.6)
High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources. (Currently, this feature is usable only on a file descriptor opened using the **O_DIRECT** flag.)

RWF_SYNC (since Linux 4.7)
Provide a per-write equivalent of the **O_SYNC** *open*(2) flag. This flag is meaningful only for **pwritev2**(), and its effect applies only to the data range written by the system call.

RWF_NOWAIT (since Linux 4.14)
Do not wait for data which is not immediately available. If this flag is specified, the **preadv2**() system call will return instantly if it would have to read data from the backing storage or wait for a lock. If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return −1 and set *errno* to **EAGAIN** (but see **BUGS**). Currently, this flag is meaningful only for **preadv2**().

RWF_APPEND (since Linux 4.16)
Provide a per-write equivalent of the **O_APPEND** *open*(2) flag. This flag is meaningful only for **pwritev2**(), and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is −1, the current file offset is updated.

RETURN VALUE

On success, **readv**(), **preadv**(), and **preadv2**() return the number of bytes read; **writev**(), **pwritev**(), and **pwritev2**() return the number of bytes written.

Note that it is not an error for a successful call to transfer fewer bytes than requested (see *read*(2) and *write*(2)).

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The errors are as given for *read*(2) and *write*(2). Furthermore, **preadv**(), **preadv2**(), **pwritev**(), and **pwritev2**() can also fail for the same reasons as *lseek*(2). Additionally, the following errors are defined:

EINVAL
The sum of the *iov_len* values overflows an *ssize_t* value.

EINVAL
The vector count, *iovcnt*, is less than zero or greater than the permitted maximum.

EOPNOTSUPP
An unknown flag is specified in *flags*.

VERSIONS

C library/kernel differences

The raw **preadv**() and **pwritev**() system calls have call signatures that differ slightly from that of the corresponding GNU C library wrapper functions shown in the SYNOPSIS. The final argument, *offset*, is unpacked by the wrapper functions into two arguments in the system calls:

unsigned long *pos_l*, **unsigned long** *pos*

These arguments contain, respectively, the low order and high order 32 bits of *offset*.

STANDARDS

readv()
writev()
POSIX.1-2008.

preadv()
pwritev()
BSD.

preadv2()
pwritev2()
Linux.

HISTORY

readv()
writev()
POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

preadv(), **pwritev**(): Linux 2.6.30, glibc 2.10.

preadv2(), **pwritev2**(): Linux 4.6, glibc 2.26.

Historical C library/kernel differences

To deal with the fact that **IOV_MAX** was so low on early versions of Linux, the glibc wrapper functions for **readv**() and **writev**() did some extra work if they detected that the underlying kernel system call failed because this limit was exceeded. In the case of **readv**(), the wrapper function allocated a temporary buffer large enough for all of the items specified by *iov*, passed that buffer in a call to *read*(2), copied data from the buffer to the locations specified by the *iov_base* fields of the elements of *iov*, and then freed the buffer. The wrapper function for **writev**() performed the analogous task using a temporary buffer and a call to *write*(2).

The need for this extra effort in the glibc wrapper functions went away with Linux 2.2 and later. However, glibc continued to provide this behavior until glibc 2.10. Starting with glibc 2.9, the wrapper functions provide this behavior only if the library detects that the system is running a Linux kernel older than Linux 2.6.18 (an arbitrarily selected kernel version). And since glibc 2.20 (which requires a minimum of Linux 2.6.32), the glibc wrapper functions always just directly invoke the system calls.

NOTES

POSIX.1 allows an implementation to place a limit on the number of items that can be passed in *iov*. An implementation can advertise its limit by defining **IOV_MAX** in *<limits.h>* or at run time via the return value from *sysconf(_SC_IOV_MAX)*. On modern Linux systems, the limit is 1024. Back in Linux 2.0 days, this limit was 16.

BUGS

Linux 5.9 and Linux 5.10 have a bug where **preadv2**() with the **RWF_NOWAIT** flag may return 0 even when not at end of file.

EXAMPLES

The following code sample demonstrates the use of **writev**():

```
char          *str0 = "hello ";
char          *str1 = "world\n";
ssize_t       nwritten;
struct iovec  iov[2];

iov[0].iov_base = str0;
iov[0].iov_len = strlen(str0);
iov[1].iov_base = str1;
iov[1].iov_len = strlen(str1);

nwritten = writev(STDOUT_FILENO, iov, 2);
```

SEE ALSO

pread(2), *read*(2), *write*(2)

NAME

reboot − reboot or enable/disable Ctrl-Alt-Del

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
/* Since Linux 2.1.30 there are symbolic names LINUX_REBOOT_*
   for the constants and a fourth argument to the call: */

#include <linux/reboot.h>  /* Definition of LINUX_REBOOT_* constants */
#include <sys/syscall.h>   /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_reboot, int magic, int magic2, int op, void *arg);

/* Under glibc and most alternative libc's (including uclibc, dietlibc,
   musl and a few others), some of the constants involved have gotten
   symbolic names RB_*, and the library call is a 1-argument
   wrapper around the system call: */

#include <sys/reboot.h>    /* Definition of RB_* constants */
#include <unistd.h>

int reboot(int op);
```

DESCRIPTION

The **reboot**() call reboots the system, or enables/disables the reboot keystroke (abbreviated CAD, since the default is Ctrl-Alt-Delete; it can be changed using *loadkeys*(1)).

This system call fails (with the error **EINVAL**) unless *magic* equals **LINUX_REBOOT_MAGIC1** (that is, 0xfee1dead) and *magic2* equals **LINUX_REBOOT_MAGIC2** (that is, 0x28121969). However, since Linux 2.1.17 also **LINUX_REBOOT_MAGIC2A** (that is, 0x05121996) and since Linux 2.1.97 also **LINUX_REBOOT_MAGIC2B** (that is, 0x16041998) and since Linux 2.5.71 also **LINUX_REBOOT_MAGIC2C** (that is, 0x20112000) are permitted as values for *magic2*. (The hexadecimal values of these constants are meaningful.)

The *op* argument can have the following values:

LINUX_REBOOT_CMD_CAD_OFF
(**RB_DISABLE_CAD**, 0). CAD is disabled. This means that the CAD keystroke will cause a **SIGINT** signal to be sent to init (process 1), whereupon this process may decide upon a proper action (maybe: kill all processes, sync, reboot).

LINUX_REBOOT_CMD_CAD_ON
(**RB_ENABLE_CAD**, 0x89abcdef). CAD is enabled. This means that the CAD keystroke will immediately cause the action associated with **LINUX_REBOOT_CMD_RESTART**.

LINUX_REBOOT_CMD_HALT
(**RB_HALT_SYSTEM**, 0xcdef0123; since Linux 1.1.76). The message "System halted." is printed, and the system is halted. Control is given to the ROM monitor, if there is one. If not preceded by a *sync*(2), data will be lost.

LINUX_REBOOT_CMD_KEXEC
(**RB_KEXEC**, 0x45584543, since Linux 2.6.13). Execute a kernel that has been loaded earlier with *kexec_load*(2). This option is available only if the kernel was configured with **CONFIG_KEXEC**.

LINUX_REBOOT_CMD_POWER_OFF
(**RB_POWER_OFF**, 0x4321fedc; since Linux 2.1.30). The message "Power down." is printed, the system is stopped, and all power is removed from the system, if possible. If not preceded by a *sync*(2), data will be lost.

LINUX_REBOOT_CMD_RESTART
(**RB_AUTOBOOT**, 0x1234567). The message "Restarting system." is printed, and a default restart is performed immediately. If not preceded by a *sync*(2), data will be lost.

LINUX_REBOOT_CMD_RESTART2
(0xa1b2c3d4; since Linux 2.1.30). The message "Restarting system with command '%s'" is printed, and a restart (using the command string given in *arg*) is performed immediately. If not preceded by a *sync*(2), data will be lost.

LINUX_REBOOT_CMD_SW_SUSPEND
(**RB_SW_SUSPEND**, 0xd000fce1; since Linux 2.5.18). The system is suspended (hibernated) to disk. This option is available only if the kernel was configured with **CONFIG_HIBERNATION**.

Only the superuser may call **reboot**().

The precise effect of the above actions depends on the architecture. For the i386 architecture, the additional argument does not do anything at present (2.1.122), but the type of reboot can be determined by kernel command-line arguments ("reboot=...") to be either warm or cold, and either hard or through the BIOS.

Behavior inside PID namespaces

Since Linux 3.4, if **reboot**() is called from a PID namespace other than the initial PID namespace with one of the *op* values listed below, it performs a "reboot" of that namespace: the "init" process of the PID namespace is immediately terminated, with the effects described in *pid_namespaces*(7).

The values that can be supplied in *op* when calling **reboot**() in this case are as follows:

LINUX_REBOOT_CMD_RESTART
LINUX_REBOOT_CMD_RESTART2
The "init" process is terminated, and *wait*(2) in the parent process reports that the child was killed with a **SIGHUP** signal.

LINUX_REBOOT_CMD_POWER_OFF
LINUX_REBOOT_CMD_HALT
The "init" process is terminated, and *wait*(2) in the parent process reports that the child was killed with a **SIGINT** signal.

For the other *op* values, **reboot**() returns −1 and *errno* is set to **EINVAL**.

RETURN VALUE

For the values of *op* that stop or restart the system, a successful call to **reboot**() does not return. For the other *op* values, zero is returned on success. In all cases, −1 is returned on failure, and *errno* is set to indicate the error.

ERRORS

EFAULT
Problem with getting user-space data under **LINUX_REBOOT_CMD_RESTART2**.

EINVAL
Bad magic numbers or *op*.

EPERM
The calling process has insufficient privilege to call **reboot**(); the caller must have the **CAP_SYS_BOOT** inside its user namespace.

STANDARDS

Linux.

SEE ALSO

systemctl(1), *systemd*(1), *kexec_load*(2), *sync*(2), *bootparam*(7), *capabilities*(7), *ctrlaltdel*(8), *halt*(8), *shutdown*(8)

NAME

recv, recvfrom, recvmsg − receive a message from a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void buf[.len], size_t len,
             int flags);
ssize_t recvfrom(int sockfd, void buf[restrict .len], size_t len,
             int flags,
             struct sockaddr *_Nullable restrict src_addr,
             socklen_t *_Nullable restrict addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

DESCRIPTION

The **recv**(), **recvfrom**(), and **recvmsg**() calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.

The only difference between **recv**() and *read*(2) is the presence of *flags*. With a zero *flags* argument, **recv**() is generally equivalent to *read*(2) (but see NOTES). Also, the following call

```
recv(sockfd, buf, len, flags);
```

is equivalent to

```
recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

All three calls return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see *fcntl*(2)), in which case the value −1 is returned and *errno* is set to **EAGAIN** or **EWOULDBLOCK**. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

An application can use *select*(2), *poll*(2), or *epoll*(7) to determine when more data arrives on a socket.

The flags argument

The *flags* argument is formed by ORing one or more of the following values:

MSG_CMSG_CLOEXEC (**recvmsg**() only; since Linux 2.6.23)
Set the close-on-exec flag for the file descriptor received via a UNIX domain file descriptor using the **SCM_RIGHTS** operation (described in *unix*(7)). This flag is useful for the same reasons as the **O_CLOEXEC** flag of *open*(2).

MSG_DONTWAIT (since Linux 2.2)
Enables nonblocking operation; if the operation would block, the call fails with the error **EAGAIN** or **EWOULDBLOCK**. This provides similar behavior to setting the **O_NONBLOCK** flag (via the *fcntl*(2) **F_SETFL** operation), but differs in that **MSG_DONTWAIT** is a per-call option, whereas **O_NONBLOCK** is a setting on the open file description (see *open*(2)), which will affect all threads in the calling process as well as other processes that hold file descriptors referring to the same open file description.

MSG_ERRQUEUE (since Linux 2.2)
This flag specifies that queued errors should be received from the socket error queue. The error is passed in an ancillary message with a type dependent on the protocol (for IPv4 **IP_RECVERR**). The user should supply a buffer of sufficient size. See *cmsg*(3) and *ip*(7) for more information. The payload of the original packet that caused the error is passed as normal data via *msg_iovec*. The original destination address of the datagram that caused the error is supplied via *msg_name*.

The error is supplied in a *sock_extended_err* structure:

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err
{
    uint32_t ee_errno;   /* Error number */
    uint8_t  ee_origin;  /* Where the error originated */
    uint8_t  ee_type;    /* Type */
    uint8_t  ee_code;    /* Code */
    uint8_t  ee_pad;     /* Padding */
    uint32_t ee_info;    /* Additional information */
    uint32_t ee_data;    /* Other data */
    /* More data may follow */
};

struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

ee_errno contains the *errno* number of the queued error. *ee_origin* is the origin code of where the error originated. The other fields are protocol-specific. The macro **SO_EE_OFFENDER** returns a pointer to the address of the network object where the error originated from given a pointer to the ancillary message. If this address is not known, the *sa_family* member of the *sockaddr* contains **AF_UNSPEC** and the other fields of the *sockaddr* are undefined. The payload of the packet that caused the error is passed as normal data.

For local errors, no address is passed (this can be checked with the *cmsg_len* member of the *cmsghdr*). For error receives, the **MSG_ERRQUEUE** flag is set in the *msghdr*. After an error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

MSG_OOB
This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols.

MSG_PEEK
This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

MSG_TRUNC (since Linux 2.2)
For raw (**AF_PACKET**), Internet datagram (since Linux 2.4.27/2.6.8), netlink (since Linux 2.6.22), and UNIX datagram as well as sequenced-packet (since Linux 3.4) sockets: return the real length of the packet or datagram, even when it was longer than the passed buffer.

For use with Internet stream sockets, see *tcp*(7).

MSG_WAITALL (since Linux 2.2)
This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned. This flag has no effect for datagram sockets.

recvfrom()
recvfrom() places the received message into the buffer *buf*. The caller must specify the size of the buffer in *len*.

If *src_addr* is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by *src_addr*. In this case, *addrlen* is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with *src_addr*. Upon return, *addrlen* is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

If the caller is not interested in the source address, *src_addr* and *addrlen* should be specified as NULL.

recv()

The **recv**() call is normally used only on a *connected* socket (see *connect*(2)). It is equivalent to the call:

```
recvfrom(fd, buf, len, flags, NULL, 0);
```

recvmsg()

The **recvmsg**() call uses a *msghdr* structure to minimize the number of directly supplied arguments. This structure is defined as follows in <*sys/socket.h*>:

```
struct msghdr {
    void         *msg_name;       /* Optional address */
    socklen_t     msg_namelen;    /* Size of address */
    struct iovec *msg_iov;        /* Scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* Ancillary data, see below */
    size_t        msg_controllen; /* Ancillary data buffer len */
    int           msg_flags;      /* Flags on received message */
};
```

The *msg_name* field points to a caller-allocated buffer that is used to return the source address if the socket is unconnected. The caller should set *msg_namelen* to the size of this buffer before this call; upon return from a successful call, *msg_namelen* will contain the length of the returned address. If the application does not need to know the source address, *msg_name* can be specified as NULL.

The fields *msg_iov* and *msg_iovlen* describe scatter-gather locations, as discussed in *readv*(2).

The field *msg_control*, which has length *msg_controllen*, points to a buffer for other protocol control-related messages or miscellaneous ancillary data. When **recvmsg**() is called, *msg_controllen* should contain the length of the available buffer in *msg_control*; upon return from a successful call it will contain the length of the control message sequence.

The messages are of the form:

```
struct cmsghdr {
    size_t cmsg_len;    /* Data byte count, including header
                           (type is socklen_t in POSIX) */
    int    cmsg_level;  /* Originating protocol */
    int    cmsg_type;   /* Protocol-specific type */
/* followed by
    unsigned char cmsg_data[]; */
};
```

Ancillary data should be accessed only by the macros defined in *cmsg*(3).

As an example, Linux uses this ancillary data mechanism to pass extended errors, IP options, or file descriptors over UNIX domain sockets. For further information on the use of ancillary data in various socket domains, see *unix*(7) and *ip*(7).

The *msg_flags* field in the *msghdr* is set on return of **recvmsg**(). It can contain several flags:

MSG_EOR
: indicates end-of-record; the data returned completed a record (generally used with sockets of type **SOCK_SEQPACKET**).

MSG_TRUNC
: indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

MSG_CTRUNC
: indicates that some control data was discarded due to lack of space in the buffer for ancillary data.

MSG_OOB
: is returned to indicate that expedited or out-of-band data was received.

MSG_ERRQUEUE
indicates that no data was received but an extended error from the socket error queue.

MSG_CMSG_CLOEXEC (since Linux 2.6.23)
indicates that **MSG_CMSG_CLOEXEC** was specified in the *flags* argument of **recvmsg**().

RETURN VALUE

These calls return the number of bytes received, or −1 if an error occurred. In the event of an error, *errno* is set to indicate the error.

When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return).

Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0.

The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

ERRORS

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their manual pages.

EAGAIN or **EWOULDBLOCK**
The socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received. POSIX.1 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF
The argument *sockfd* is an invalid file descriptor.

ECONNREFUSED
A remote host refused to allow the network connection (typically because it is not running the requested service).

EFAULT
The receive buffer pointer(s) point outside the process's address space.

EINTR
The receive was interrupted by delivery of a signal before any data was available; see *signal*(7).

EINVAL
Invalid argument passed.

ENOMEM
Could not allocate memory for **recvmsg**().

ENOTCONN
The socket is associated with a connection-oriented protocol and has not been connected (see *connect*(2) and *accept*(2)).

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

VERSIONS

According to POSIX.1, the *msg_controllen* field of the *msghdr* structure should be typed as *socklen_t*, and the *msg_iovlen* field should be typed as *int*, but glibc currently types both as *size_t*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

POSIX.1 describes only the **MSG_OOB**, **MSG_PEEK**, and **MSG_WAITALL** flags.

NOTES

If a zero-length datagram is pending, *read*(2) and **recv**() with a *flags* argument of zero provide different behavior. In this circumstance, *read*(2) has no effect (the datagram remains pending), while **recv**()

consumes the pending datagram.

See *recvmmsg*(2) for information about a Linux-specific system call that can be used to receive multiple datagrams in a single call.

EXAMPLES

An example of the use of **recvfrom**() is shown in *getaddrinfo*(3).

SEE ALSO

fcntl(2), *getsockopt*(2), *read*(2), *recvmmsg*(2), *select*(2), *shutdown*(2), *socket*(2), *cmsg*(3), *sockatmark*(3), *ip*(7), *ipv6*(7), *socket*(7), *tcp*(7), *udp*(7), *unix*(7)

NAME

recvmmsg − receive multiple messages on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/socket.h>

int recvmmsg(int sockfd, struct mmsghdr *msgvec, unsigned int vlen,
             int flags, struct timespec *timeout);
```

DESCRIPTION

The **recvmmsg**() system call is an extension of *recvmsg*(2) that allows the caller to receive multiple messages from a socket using a single system call. (This has performance benefits for some applications.) A further extension over *recvmsg*(2) is support for a timeout on the receive operation.

The *sockfd* argument is the file descriptor of the socket to receive data from.

The *msgvec* argument is a pointer to an array of *mmsghdr* structures. The size of this array is specified in *vlen*.

The *mmsghdr* structure is defined in <*sys/socket.h*> as:

```
struct mmsghdr {
    struct msghdr msg_hdr;  /* Message header */
    unsigned int  msg_len;  /* Number of received bytes for header */
};
```

The *msg_hdr* field is a *msghdr* structure, as described in *recvmsg*(2). The *msg_len* field is the number of bytes returned for the message in the entry. This field has the same value as the return value of a single *recvmsg*(2) on the header.

The *flags* argument contains flags ORed together. The flags are the same as documented for *recvmsg*(2), with the following addition:

MSG_WAITFORONE (since Linux 2.6.34)
Turns on **MSG_DONTWAIT** after the first message has been received.

The *timeout* argument points to a *struct timespec* (see *clock_gettime*(2)) defining a timeout (seconds plus nanoseconds) for the receive operation (*but see BUGS!*). (This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.) If *timeout* is NULL, then the operation blocks indefinitely.

A blocking **recvmmsg**() call blocks until *vlen* messages have been received or until the timeout expires. A nonblocking call reads as many messages as are available (up to the limit specified by *vlen*) and returns immediately.

On return from **recvmmsg**(), successive elements of *msgvec* are updated to contain information about each received message: *msg_len* contains the size of the received message; the subfields of *msg_hdr* are updated as described in *recvmsg*(2). The return value of the call indicates the number of elements of *msgvec* that have been updated.

RETURN VALUE

On success, **recvmmsg**() returns the number of messages received in *msgvec*; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Errors are as for *recvmsg*(2). In addition, the following error can occur:

EINVAL
timeout is invalid.

See also BUGS.

STANDARDS

Linux.

HISTORY

Linux 2.6.33, glibc 2.12.

BUGS

The *timeout* argument does not work as intended. The timeout is checked only after the receipt of each datagram, so that if up to *vlen−1* datagrams are received before the timeout expires, but then no further datagrams are received, the call will block forever.

If an error occurs after at least one message has been received, the call succeeds, and returns the number of messages received. The error code is expected to be returned on a subsequent call to **recvmmsg**(). In the current implementation, however, the error code can be overwritten in the meantime by an unrelated network event on a socket, for example an incoming ICMP packet.

EXAMPLES

The following program uses **recvmmsg**() to receive multiple messages on a socket and stores them in multiple buffers. The call returns if all buffers are filled or if the timeout specified has expired.

The following snippet periodically generates UDP datagrams containing a random number:

```
$ while true; do echo $RANDOM > /dev/udp/127.0.0.1/1234;
      sleep 0.25; done
```

These datagrams are read by the example application, which can give the following output:

```
$ ./a.out
5 messages received
1 11782
2 11345
3 304
4 13514
5 28421
```

Program source

```
#define _GNU_SOURCE
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <time.h>

int
main(void)
{
#define VLEN 10
#define BUFSIZE 200
#define TIMEOUT 1
    int                 sockfd, retval;
    char                bufs[VLEN][BUFSIZE+1];
    struct iovec        iovecs[VLEN];
    struct mmsghdr      msgs[VLEN];
    struct timespec     timeout;
    struct sockaddr_in  addr;

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    addr.sin_family = AF_INET;
```

```
        addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
        addr.sin_port = htons(1234);
        if (bind(sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
            perror("bind()");
            exit(EXIT_FAILURE);
        }

        memset(msgs, 0, sizeof(msgs));
        for (size_t i = 0; i < VLEN; i++) {
            iovecs[i].iov_base         = bufs[i];
            iovecs[i].iov_len          = BUFSIZE;
            msgs[i].msg_hdr.msg_iov    = &iovecs[i];
            msgs[i].msg_hdr.msg_iovlen = 1;
        }

        timeout.tv_sec = TIMEOUT;
        timeout.tv_nsec = 0;

        retval = recvmmsg(sockfd, msgs, VLEN, 0, &timeout);
        if (retval == -1) {
            perror("recvmmsg()");
            exit(EXIT_FAILURE);
        }

        printf("%d messages received\n", retval);
        for (size_t i = 0; i < retval; i++) {
            bufs[i][msgs[i].msg_len] = 0;
            printf("%zu %s", i+1, bufs[i]);
        }
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

clock_gettime(2), *recvmsg*(2), *sendmmsg*(2), *sendmsg*(2), *socket*(2), *socket*(7)

NAME

remap_file_pages − create a nonlinear file mapping

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/mman.h>

[[deprecated]] int remap_file_pages(void addr[.size], size_t size,
                                    int prot, size_t pgoff, int flags);
```

DESCRIPTION

Note: this system call was marked as deprecated starting with Linux 3.16. In Linux 4.0, the implementation was replaced by a slower in-kernel emulation. Those few applications that use this system call should consider migrating to alternatives. This change was made because the kernel code for this system call was complex, and it is believed to be little used or perhaps even completely unused. While it had some use cases in database applications on 32-bit systems, those use cases don't exist on 64-bit systems.

The **remap_file_pages**() system call is used to create a nonlinear mapping, that is, a mapping in which the pages of the file are mapped into a nonsequential order in memory. The advantage of using **remap_file_pages**() over using repeated calls to *mmap*(2) is that the former approach does not require the kernel to create additional VMA (Virtual Memory Area) data structures.

To create a nonlinear mapping we perform the following steps:

1. Use *mmap*(2) to create a mapping (which is initially linear). This mapping must be created with the **MAP_SHARED** flag.
2. Use one or more calls to **remap_file_pages**() to rearrange the correspondence between the pages of the mapping and the pages of the file. It is possible to map the same page of a file into multiple locations within the mapped region.

The *pgoff* and *size* arguments specify the region of the file that is to be relocated within the mapping: *pgoff* is a file offset in units of the system page size; *size* is the length of the region in bytes.

The *addr* argument serves two purposes. First, it identifies the mapping whose pages we want to rearrange. Thus, *addr* must be an address that falls within a region previously mapped by a call to *mmap*(2). Second, *addr* specifies the address at which the file pages identified by *pgoff* and *size* will be placed.

The values specified in *addr* and *size* should be multiples of the system page size. If they are not, then the kernel rounds *both* values *down* to the nearest multiple of the page size.

The *prot* argument must be specified as 0.

The *flags* argument has the same meaning as for *mmap*(2), but all flags other than **MAP_NONBLOCK** are ignored.

RETURN VALUE

On success, **remap_file_pages**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: *addr* does not refer to a valid mapping created with the **MAP_SHARED** flag.

EINVAL
: *addr*, *size*, *prot*, or *pgoff* is invalid.

STANDARDS

Linux.

HISTORY

Linux 2.5.46, glibc 2.3.3.

NOTES

Since Linux 2.6.23, **remap_file_pages**() creates non-linear mappings only on in-memory filesystems such as *tmpfs*(5), hugetlbfs or ramfs. On filesystems with a backing store, **remap_file_pages**() is not much more efficient than using *mmap*(2) to adjust which parts of the file are mapped to which addresses.

SEE ALSO

getpagesize(2), *mmap*(2), *mmap2*(2), *mprotect*(2), *mremap*(2), *msync*(2)

NAME

removexattr, lremovexattr, fremovexattr − remove an extended attribute

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/xattr.h>

int removexattr(const char *path, const char *name);
int lremovexattr(const char *path, const char *name);
int fremovexattr(int fd, const char *name);
```

DESCRIPTION

Extended attributes are *name*:*value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the *stat*(2) data). A complete overview of extended attributes concepts can be found in *xattr*(7).

removexattr() removes the extended attribute identified by *name* and associated with the given *path* in the filesystem.

lremovexattr() is identical to **removexattr**(), except in the case of a symbolic link, where the extended attribute is removed from the link itself, not the file that it refers to.

fremovexattr() is identical to **removexattr**(), only the extended attribute is removed from the open file referred to by *fd* (as returned by *open*(2)) in place of *path*.

An extended attribute name is a null-terminated string. The *name* includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode.

RETURN VALUE

On success, zero is returned. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

ENODATA
: The named attribute does not exist.

ENOTSUP
: Extended attributes are not supported by the filesystem, or are disabled.

In addition, the errors documented in *stat*(2) can also occur.

STANDARDS

Linux.

HISTORY

Linux 2.4, glibc 2.3.

SEE ALSO

getfattr(1), *setfattr*(1), *getxattr*(2), *listxattr*(2), *open*(2), *setxattr*(2), *stat*(2), *symlink*(7), *xattr*(7)

NAME

rename, renameat, renameat2 − change the name or location of a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <stdio.h>

int renameat(int olddirfd, const char *oldpath,
             int newdirfd, const char *newpath);
int renameat2(int olddirfd, const char *oldpath,
              int newdirfd, const char *newpath, unsigned int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

renameat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

renameat2():
_GNU_SOURCE

DESCRIPTION

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using *link*(2)) are unaffected. Open file descriptors for *oldpath* are also unaffected.

Various restrictions determine whether or not the rename operation succeeds: see ERRORS below.

If *newpath* already exists, it will be atomically replaced, so that there is no point at which another process attempting to access *newpath* will find it missing. However, there will probably be a window in which both *oldpath* and *newpath* refer to the file being renamed.

If *oldpath* and *newpath* are existing hard links referring to the same file, then **rename**() does nothing, and returns a success status.

If *newpath* exists but the operation fails for some reason, **rename**() guarantees to leave an instance of *newpath* in place.

oldpath can specify a directory. In this case, *newpath* must either not exist, or it must specify an empty directory.

If *oldpath* refers to a symbolic link, the link is renamed; if *newpath* refers to a symbolic link, the link will be overwritten.

renameat()

The **renameat**() system call operates in exactly the same way as **rename**(), except for the differences described here.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by **rename**() for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value **AT_FDCWD**, then *oldpath* is interpreted relative to the current working directory of the calling process (like *rename*())

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor *newdirfd*.

See *openat*(2) for an explanation of the need for **renameat**().

renameat2()

renameat2() has an additional *flags* argument. A **renameat2**() call with a zero *flags* argument is equivalent to **renameat**().

The *flags* argument is a bit mask consisting of zero or more of the following flags:

RENAME_EXCHANGE

Atomically exchange *oldpath* and *newpath*. Both pathnames must exist but may be of different types (e.g., one could be a non-empty directory and the other a symbolic link).

RENAME_NOREPLACE

Don't overwrite *newpath* of the rename. Return an error if *newpath* already exists.

RENAME_NOREPLACE can't be employed together with **RENAME_EXCHANGE**.

RENAME_NOREPLACE requires support from the underlying filesystem. Support for various filesystems was added as follows:

- ext4 (Linux 3.15);
- btrfs, tmpfs, and cifs (Linux 3.17);
- xfs (Linux 4.0);
- Support for many other filesystems was added in Linux 4.9, including ext2, minix, reiserfs, jfs, vfat, and bpf.

RENAME_WHITEOUT (since Linux 3.18)

This operation makes sense only for overlay/union filesystem implementations.

Specifying **RENAME_WHITEOUT** creates a "whiteout" object at the source of the rename at the same time as performing the rename. The whole operation is atomic, so that if the rename succeeds then the whiteout will also have been created.

A "whiteout" is an object that has special meaning in union/overlay filesystem constructs. In these constructs, multiple layers exist and only the top one is ever modified. A whiteout on an upper layer will effectively hide a matching file in the lower layer, making it appear as if the file didn't exist.

When a file that exists on the lower layer is renamed, the file is first copied up (if not already on the upper layer) and then renamed on the upper, read-write layer. At the same time, the source file needs to be "whiteouted" (so that the version of the source file in the lower layer is rendered invisible). The whole operation needs to be done atomically.

When not part of a union/overlay, the whiteout appears as a character device with a {0,0} device number. (Note that other union/overlay implementations may employ different methods for storing whiteout entries; specifically, BSD union mount employs a separate inode type, **DT_WHT**, which, while supported by some filesystems available in Linux, such as CODA and XFS, is ignored by the kernel's whiteout support code, as of Linux 4.19, at least.)

RENAME_WHITEOUT requires the same privileges as creating a device node (i.e., the **CAP_MKNOD** capability).

RENAME_WHITEOUT can't be employed together with **RENAME_EXCHANGE**.

RENAME_WHITEOUT requires support from the underlying filesystem. Among the filesystems that support it are tmpfs (since Linux 3.18), ext4 (since Linux 3.18), XFS (since Linux 4.1), f2fs (since Linux 4.2), btrfs (since Linux 4.7), and ubifs (since Linux 4.9).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES

Write permission is denied for the directory containing *oldpath* or *newpath*, or, search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*, or *oldpath* is a directory and does not allow write permission (needed to update the .. entry). (See also *path_resolution*(7).)

EBUSY
The rename fails because *oldpath* or *newpath* is a directory that is in use by some process (perhaps as current working directory, or as root directory, or because it was open for reading) or is in use by the system (for example as a mount point), while the system considers this an error. (Note that there is no requirement to return **EBUSY** in such cases—there is nothing wrong with doing the rename anyway—but it is allowed to return **EBUSY** if the system cannot otherwise handle such situations.)

EDQUOT
The user's quota of disk blocks on the filesystem has been exhausted.

EFAULT
oldpath or *newpath* points outside your accessible address space.

EINVAL
The new pathname contained a path prefix of the old, or, more generally, an attempt was made to make a directory a subdirectory of itself.

EISDIR
newpath is an existing directory, but *oldpath* is not a directory.

ELOOP
Too many symbolic links were encountered in resolving *oldpath* or *newpath*.

EMLINK
oldpath already has the maximum number of links to it, or it was a directory and the directory containing *newpath* has the maximum number of links.

ENAMETOOLONG
oldpath or *newpath* was too long.

ENOENT
The link named by *oldpath* does not exist; or, a directory component in *newpath* does not exist; or, *oldpath* or *newpath* is an empty string.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing the file has no room for the new directory entry.

ENOTDIR
A component used as a directory in *oldpath* or *newpath* is not, in fact, a directory. Or, *oldpath* is a directory, and *newpath* exists but is not a directory.

ENOTEMPTY or **EEXIST**
newpath is a nonempty directory, that is, contains entries other than "." and "..".

EPERM or **EACCES**
The directory containing *oldpath* has the sticky bit (**S_ISVTX**) set and the process's effective user ID is neither the user ID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or *newpath* is an existing file and the directory containing it has the sticky bit set and the process's effective user ID is neither the user ID of the file to be replaced nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or the filesystem containing *oldpath* does not support renaming of the type requested.

EROFS
The file is on a read-only filesystem.

EXDEV
oldpath and *newpath* are not on the same mounted filesystem. (Linux permits a filesystem to be mounted at multiple points, but **rename**() does not work across different mount points, even if the same filesystem is mounted on both.)

The following additional errors can occur for **renameat**() and **renameat2**():

EBADF
oldpath (*newpath*) is relative but *olddirfd* (*newdirfd*) is not a valid file descriptor.

ENOTDIR
oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

The following additional errors can occur for **renameat2**():

EEXIST
flags contains **RENAME_NOREPLACE** and *newpath* already exists.

EINVAL
An invalid flag was specified in *flags*.

EINVAL
Both **RENAME_NOREPLACE** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL
Both **RENAME_WHITEOUT** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL
The filesystem does not support one of the flags in *flags*.

ENOENT
flags contains **RENAME_EXCHANGE** and *newpath* does not exist.

EPERM
RENAME_WHITEOUT was specified in *flags*, but the caller does not have the **CAP_MKNOD** capability.

STANDARDS

rename()
C11, POSIX.1-2008.

renameat()
POSIX.1-2008.

renameat2()
Linux.

HISTORY

rename()
4.3BSD, C89, POSIX.1-2001.

renameat()
Linux 2.6.16, glibc 2.4.

renameat2()
Linux 3.15, glibc 2.28.

glibc notes

On older kernels where **renameat**() is unavailable, the glibc wrapper function falls back to the use of **rename**(). When *oldpath* and *newpath* are relative pathnames, glibc constructs pathnames based on the symbolic links in */proc/self/fd* that correspond to the *olddirfd* and *newdirfd* arguments.

BUGS

On NFS filesystems, you can not assume that if the operation failed, the file was not renamed. If the server does the rename operation and then crashes, the retransmitted RPC which will be processed when the server is up again causes a failure. The application is expected to deal with this. See *link*(2) for a similar problem.

SEE ALSO

mv(1), *rename*(1), *chmod*(2), *link*(2), *symlink*(2), *unlink*(2), *path_resolution*(7), *symlink*(7)

NAME

request_key − request a key from the kernel's key management facility

LIBRARY

Linux Key Management Utilities (*libkeyutils*, *−lkeyutils*)

SYNOPSIS

#include <keyutils.h>

key_serial_t request_key(const char ****type*, const char ****description***,**
const char *_Nullable *callout_info***,**
key_serial_t *dest_keyring***);**

DESCRIPTION

request_key() attempts to find a key of the given *type* with a description (name) that matches the specified *description*. If such a key could not be found, then the key is optionally created. If the key is found or created, **request_key**() attaches it to the keyring whose ID is specified in *dest_keyring* and returns the key's serial number.

request_key() first recursively searches for a matching key in all of the keyrings attached to the calling process. The keyrings are searched in the order: thread-specific keyring, process-specific keyring, and then session keyring.

If **request_key**() is called from a program invoked by **request_key**() on behalf of some other process to generate a key, then the keyrings of that other process will be searched next, using that other process's user ID, group ID, supplementary group IDs, and security context to determine access.

The search of the keyring tree is breadth-first: the keys in each keyring searched are checked for a match before any child keyrings are recursed into. Only keys for which the caller has *search* permission be found, and only keyrings for which the caller has *search* permission may be searched.

If the key is not found and *callout* is NULL, then the call fails with the error **ENOKEY**.

If the key is not found and *callout* is not NULL, then the kernel attempts to invoke a user-space program to instantiate the key. The details are given below.

The *dest_keyring* serial number may be that of a valid keyring for which the caller has *write* permission, or it may be one of the following special keyring IDs:

KEY_SPEC_THREAD_KEYRING
This specifies the caller's thread-specific keyring (see *thread-keyring*(7)).

KEY_SPEC_PROCESS_KEYRING
This specifies the caller's process-specific keyring (see *process-keyring*(7)).

KEY_SPEC_SESSION_KEYRING
This specifies the caller's session-specific keyring (see *session-keyring*(7)).

KEY_SPEC_USER_KEYRING
This specifies the caller's UID-specific keyring (see *user-keyring*(7)).

KEY_SPEC_USER_SESSION_KEYRING
This specifies the caller's UID-session keyring (see *user-session-keyring*(7)).

When the *dest_keyring* is specified as 0 and no key construction has been performed, then no additional linking is done.

Otherwise, if *dest_keyring* is 0 and a new key is constructed, the new key will be linked to the "default" keyring. More precisely, when the kernel tries to determine to which keyring the newly constructed key should be linked, it tries the following keyrings, beginning with the keyring set via the *keyctl*(2) **KEYCTL_SET_REQKEY_KEYRING** operation and continuing in the order shown below until it finds the first keyring that exists:

- The requestor keyring (**KEY_REQKEY_DEFL_REQUESTOR_KEYRING**, since Linux 2.6.29).
- The thread-specific keyring (**KEY_REQKEY_DEFL_THREAD_KEYRING**; see *thread-keyring*(7)).

- The process-specific keyring (**KEY_REQKEY_DEFL_PROCESS_KEYRING**; see *process-keyring*(7)).
- The session-specific keyring (**KEY_REQKEY_DEFL_SESSION_KEYRING**; see *session-keyring*(7)).
- The session keyring for the process's user ID (**KEY_REQKEY_DEFL_USER_SESSION_KEYRING**; see *user-session-keyring*(7)). This keyring is expected to always exist.
- The UID-specific keyring (**KEY_REQKEY_DEFL_USER_KEYRING**; see *user-keyring*(7)). This keyring is also expected to always exist.

If the *keyctl*(2) **KEYCTL_SET_REQKEY_KEYRING** operation specifies **KEY_REQKEY_DEFL_DEFAULT** (or no **KEYCTL_SET_REQKEY_KEYRING** operation is performed), then the kernel looks for a keyring starting from the beginning of the list.

Requesting user-space instantiation of a key

If the kernel cannot find a key matching *type* and *description*, and *callout* is not NULL, then the kernel attempts to invoke a user-space program to instantiate a key with the given *type* and *description*. In this case, the following steps are performed:

(1) The kernel creates an uninstantiated key, U, with the requested *type* and *description*.

(2) The kernel creates an authorization key, V, that refers to the key U and records the facts that the caller of **request_key**() is:

(2.1) the context in which the key U should be instantiated and secured, and

(2.2) the context from which associated key requests may be satisfied.

The authorization key is constructed as follows:

- The key type is *".request_key_auth"*.
- The key's UID and GID are the same as the corresponding filesystem IDs of the requesting process.
- The key grants *view*, *read*, and *search* permissions to the key possessor as well as *view* permission for the key user.
- The description (name) of the key is the hexadecimal string representing the ID of the key that is to be instantiated in the requesting program.
- The payload of the key is taken from the data specified in *callout_info*.
- Internally, the kernel also records the PID of the process that called **request_key**().

(3) The kernel creates a process that executes a user-space service such as *request-key*(8) with a new session keyring that contains a link to the authorization key, V.

This program is supplied with the following command-line arguments:

[0] The string *"/sbin/request−key"*.

[1] The string *"create"* (indicating that a key is to be created).

[2] The ID of the key that is to be instantiated.

[3] The filesystem UID of the caller of **request_key**().

[4] The filesystem GID of the caller of **request_key**().

[5] The ID of the thread keyring of the caller of **request_key**(). This may be zero if that keyring hasn't been created.

[6] The ID of the process keyring of the caller of **request_key**(). This may be zero if that keyring hasn't been created.

[7] The ID of the session keyring of the caller of **request_key**().

Note: each of the command-line arguments that is a key ID is encoded in *decimal* (unlike the key IDs shown in */proc/keys*, which are shown as hexadecimal values).

(4) The program spawned in the previous step:

- Assumes the authority to instantiate the key U using the *keyctl*(2) **KEYCTL_ASSUME_AUTHORITY** operation (typically via the *keyctl_assume_authority*(3) function).
- Obtains the callout data from the payload of the authorization key V (using the *keyctl*(2) **KEYCTL_READ** operation (or, more commonly, the *keyctl_read*(3) function) with a key ID value of **KEY_SPEC_REQKEY_AUTH_KEY**).
- Instantiates the key (or execs another program that performs that task), specifying the payload and destination keyring. (The destination keyring that the requestor specified when calling **request_key**() can be accessed using the special key ID **KEY_SPEC_REQUESTOR_KEYRING**.) Instantiation is performed using the *keyctl*(2) **KEYCTL_INSTANTIATE** operation (or, more commonly, the *keyctl_instantiate*(3) function). At this point, the **request_key**() call completes, and the requesting program can continue execution.

If these steps are unsuccessful, then an **ENOKEY** error will be returned to the caller of **request_key**() and a temporary, negatively instantiated key will be installed in the keyring specified by *dest_keyring*. This will expire after a few seconds, but will cause subsequent calls to **request_key**() to fail until it does. The purpose of this negatively instantiated key is to prevent (possibly different) processes making repeated requests (that require expensive *request-key*(8) upcalls) for a key that can't (at the moment) be positively instantiated.

Once the key has been instantiated, the authorization key (**KEY_SPEC_REQKEY_AUTH_KEY**) is revoked, and the destination keyring (**KEY_SPEC_REQUESTOR_KEYRING**) is no longer accessible from the *request-key*(8) program.

If a key is created, then—regardless of whether it is a valid key or a negatively instantiated key—it will displace any other key with the same type and description from the keyring specified in *dest_keyring*.

RETURN VALUE

On success, **request_key**() returns the serial number of the key it found or caused to be created. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
The keyring wasn't available for modification by the user.

EDQUOT
The key quota for this user would be exceeded by creating this key or linking it to the keyring.

EFAULT
One of *type*, *description*, or *callout_info* points outside the process's accessible address space.

EINTR
The request was interrupted by a signal; see *signal*(7).

EINVAL
The size of the string (including the terminating null byte) specified in *type* or *description* exceeded the limit (32 bytes and 4096 bytes respectively).

EINVAL
The size of the string (including the terminating null byte) specified in *callout_info* exceeded the system page size.

EKEYEXPIRED
An expired key was found, but no replacement could be obtained.

EKEYREJECTED
The attempt to generate a new key was rejected.

EKEYREVOKED
A revoked key was found, but no replacement could be obtained.

ENOKEY
No matching key was found.

ENOMEM
Insufficient memory to create a key.

EPERM
The *type* argument started with a period ('.').

STANDARDS
Linux.

HISTORY
Linux 2.6.10.

The ability to instantiate keys upon request was added in Linux 2.6.13.

EXAMPLES
The program below demonstrates the use of **request_key**(). The *type*, *description*, and *callout_info* arguments for the system call are taken from the values supplied in the command-line arguments. The call specifies the session keyring as the target keyring.

In order to demonstrate this program, we first create a suitable entry in the file */etc/request−key.conf*.

```
$ sudo sh
# echo 'create user mtk:* *   /bin/keyctl instantiate %k %c %S' \
        > /etc/request-key.conf
# exit
```

This entry specifies that when a new "user" key with the prefix "mtk:" must be instantiated, that task should be performed via the *keyctl*(1) command's **instantiate** operation. The arguments supplied to the **instantiate** operation are: the ID of the uninstantiated key (*%k*); the callout data supplied to the **request_key**() call (*%c*); and the session keyring (*%S*) of the requestor (i.e., the caller of *request_key*())See *request-key.conf* (5) for details of these *%* specifiers.

Then we run the program and check the contents of */proc/keys* to verify that the requested key has been instantiated:

```
$ ./t_request_key user mtk:key1 "Payload data"
$ grep '2dddaf50' /proc/keys
2dddaf50 I--Q---  1 perm 3f010000  1000  1000 user  mtk:key1: 12
```

For another example of the use of this program, see *keyctl*(2).

Program source

```
/* t_request_key.c */

#include <keyutils.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    key_serial_t key;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s type description callout-data\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    key = request_key(argv[1], argv[2], argv[3],
                      KEY_SPEC_SESSION_KEYRING);
    if (key == -1) {
        perror("request_key");
        exit(EXIT_FAILURE);
```

```
        }

        printf("Key ID is %jx\n", (uintmax_t) key);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

keyctl(1), *add_key*(2), *keyctl*(2), *keyctl*(3), *capabilities*(7), *keyrings*(7), *keyutils*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7), *request-key*(8)

The kernel source files *Documentation/security/keys/core.rst* and *Documentation/keys/request−key.rst* (or, before Linux 4.13, in the files *Documentation/security/keys.txt* and *Documentation/security/keys−request−key.txt*).

NAME

restart_syscall − restart a system call after interruption by a stop signal

SYNOPSIS

long restart_syscall(void);

Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION

The **restart_syscall**() system call is used to restart certain system calls after a process that was stopped by a signal (e.g., **SIGSTOP** or **SIGTSTP**) is later resumed after receiving a **SIGCONT** signal. This system call is designed only for internal use by the kernel.

restart_syscall() is used for restarting only those system calls that, when restarted, should adjust their time-related parameters—namely *poll*(2) (since Linux 2.6.24), *nanosleep*(2) (since Linux 2.6), *clock_nanosleep*(2) (since Linux 2.6), and *futex*(2), when employed with the **FUTEX_WAIT** (since Linux 2.6.22) and **FUTEX_WAIT_BITSET** (since Linux 2.6.31) operations. **restart_syscall**() restarts the interrupted system call with a time argument that is suitably adjusted to account for the time that has already elapsed (including the time where the process was stopped by a signal). Without the **restart_syscall**() mechanism, restarting these system calls would not correctly deduct the already elapsed time when the process continued execution.

RETURN VALUE

The return value of **restart_syscall**() is the return value of whatever system call is being restarted.

ERRORS

errno is set as per the errors for whatever system call is being restarted by **restart_syscall**().

STANDARDS

Linux.

HISTORY

Linux 2.6.

NOTES

There is no glibc wrapper for this system call, because it is intended for use only by the kernel and should never be called by applications.

The kernel uses **restart_syscall**() to ensure that when a system call is restarted after a process has been stopped by a signal and then resumed by **SIGCONT**, then the time that the process spent in the stopped state is counted against the timeout interval specified in the original system call. In the case of system calls that take a timeout argument and automatically restart after a stop signal plus **SIGCONT**, but which do not have the **restart_syscall**() mechanism built in, then, after the process resumes execution, the time that the process spent in the stop state is *not* counted against the timeout value. Notable examples of system calls that suffer this problem are *ppoll*(2), *select*(2), and *pselect*(2).

From user space, the operation of **restart_syscall**() is largely invisible: to the process that made the system call that is restarted, it appears as though that system call executed and returned in the usual fashion.

SEE ALSO

sigaction(2), *sigreturn*(2), *signal*(7)

NAME

rmdir − delete a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int rmdir(const char * *pathname*);

DESCRIPTION

rmdir() deletes a directory, which must be empty.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: Write access to the directory containing *pathname* was not allowed, or one of the directories in the path prefix of *pathname* did not allow search permission. (See also *path_resolution*(7).)

EBUSY
: *pathname* is currently in use by the system or some process that prevents its removal. On Linux, this means *pathname* is currently used as a mount point or is the root directory of the calling process.

EFAULT
: *pathname* points outside your accessible address space.

EINVAL
: *pathname* has . as last component.

ELOOP
: Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
: *pathname* was too long.

ENOENT
: A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOMEM
: Insufficient kernel memory was available.

ENOTDIR
: *pathname*, or a component used as a directory in *pathname*, is not, in fact, a directory.

ENOTEMPTY
: *pathname* contains entries other than . and .. ; or, *pathname* has .. as its final component. POSIX.1 also allows **EEXIST** for this condition.

EPERM
: The directory containing *pathname* has the sticky bit (**S_ISVTX**) set and the process's effective user ID is neither the user ID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability).

EPERM
: The filesystem containing *pathname* does not support the removal of directories.

EROFS
: *pathname* refers to a directory on a read-only filesystem.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

BUGS

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of directories which are still being used.

SEE ALSO

rm(1), *rmdir*(1), *chdir*(2), *chmod*(2), *mkdir*(2), *rename*(2), *unlink*(2), *unlinkat*(2)

NAME

rt_sigqueueinfo, rt_tgsigqueueinfo − queue a signal and data

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/signal.h>     /* Definition of SI_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_rt_sigqueueinfo, pid_t tgid,
        int sig, siginfo_t *info);
int syscall(SYS_rt_tgsigqueueinfo, pid_t tgid, pid_t tid,
        int sig, siginfo_t *info);
```

Note: There are no glibc wrappers for these system calls; see NOTES.

DESCRIPTION

The **rt_sigqueueinfo**() and **rt_tgsigqueueinfo**() system calls are the low-level interfaces used to send a signal plus data to a process or thread. The receiver of the signal can obtain the accompanying data by establishing a signal handler with the *sigaction*(2) **SA_SIGINFO** flag.

These system calls are not intended for direct application use; they are provided to allow the implementation of *sigqueue*(3) and *pthread_sigqueue*(3).

The **rt_sigqueueinfo**() system call sends the signal *sig* to the thread group with the ID *tgid*. (The term "thread group" is synonymous with "process", and *tid* corresponds to the traditional UNIX process ID.) The signal will be delivered to an arbitrary member of the thread group (i.e., one of the threads that is not currently blocking the signal).

The *info* argument specifies the data to accompany the signal. This argument is a pointer to a structure of type *siginfo_t*, described in *sigaction*(2) (and defined by including <*sigaction.h*>). The caller should set the following fields in this structure:

si_code
: This should be one of the **SI_*** codes in the Linux kernel source file *include/asm−generic/siginfo.h*. If the signal is being sent to any process other than the caller itself, the following restrictions apply:

  - The code can't be a value greater than or equal to zero. In particular, it can't be **SI_USER**, which is used by the kernel to indicate a signal sent by *kill*(2), and nor can it be **SI_KERNEL**, which is used to indicate a signal generated by the kernel.
  - The code can't (since Linux 2.6.39) be **SI_TKILL**, which is used by the kernel to indicate a signal sent using *tgkill*(2).

si_pid This should be set to a process ID, typically the process ID of the sender.

si_uid This should be set to a user ID, typically the real user ID of the sender.

si_value
: This field contains the user data to accompany the signal. For more information, see the description of the last (*union sigval*) argument of *sigqueue*(3).

Internally, the kernel sets the *si_signo* field to the value specified in *sig*, so that the receiver of the signal can also obtain the signal number via that field.

The **rt_tgsigqueueinfo**() system call is like **rt_sigqueueinfo**(), but sends the signal and data to the single thread specified by the combination of *tgid*, a thread group ID, and *tid*, a thread in that thread group.

RETURN VALUE

On success, these system calls return 0. On error, they return −1 and *errno* is set to indicate the error.

ERRORS

EAGAIN
: The limit of signals which may be queued has been reached. (See *signal*(7) for further information.)

EINVAL
sig, *tgid*, or *tid* was invalid.

EPERM
The caller does not have permission to send the signal to the target. For the required permissions, see *kill*(2).

EPERM
tgid specifies a process other than the caller and *info->si_code* is invalid.

ESRCH
rt_sigqueueinfo(): No thread group matching *tgid* was found.

rt_tgsigqueinfo(): No thread matching *tgid* and *tid* was found.

STANDARDS
Linux.

HISTORY
rt_sigqueueinfo()
Linux 2.2.

rt_tgsigqueueinfo()
Linux 2.6.31.

NOTES
Since these system calls are not intended for application use, there are no glibc wrapper functions; use *syscall*(2) in the unlikely case that you want to call them directly.

As with *kill*(2), the null signal (0) can be used to check if the specified process or thread exists.

SEE ALSO
kill(2), *pidfd_send_signal*(2), *sigaction*(2), *sigprocmask*(2), *tgkill*(2), *pthread_sigqueue*(3), *sigqueue*(3), *signal*(7)

NAME

s390_guarded_storage − operations with z/Architecture guarded storage facility

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <asm/guarded_storage.h> /* Definition of **GS_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_s390_guarded_storage, int *command***,**
struct gs_cb ***gs_cb*);**

Note: glibc provides no wrapper for **s390_guarded_storage**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **s390_guarded_storage**() system call enables the use of the Guarded Storage Facility (a z/Architecture-specific feature) for user-space processes.

The guarded storage facility is a hardware feature that allows marking up to 64 memory regions (as of z14) as guarded; reading a pointer with a newly introduced "Load Guarded" (LGG) or "Load Logical and Shift Guarded" (LLGFSG) instructions will cause a range check on the loaded value and invoke a (previously set up) user-space handler if one of the guarded regions is affected.

The *command* argument indicates which function to perform. The following commands are supported:

GS_ENABLE
Enable the guarded storage facility for the calling task. The initial content of the guarded storage control block will be all zeros. After enablement, user-space code can use the "Load Guarded Storage Controls" (LGSC) instruction (or the **load_gs_cb**() function wrapper provided in the *asm/guarded_storage.h* header) to load an arbitrary control block. While a task is enabled, the kernel will save and restore the calling content of the guarded storage registers on context switch.

GS_DISABLE
Disables the use of the guarded storage facility for the calling task. The kernel will cease to save and restore the content of the guarded storage registers, the task-specific content of these registers is lost.

GS_SET_BC_CB
Set a broadcast guarded storage control block to the one provided in the *gs_cb* argument. This is called per thread and associates a specific guarded storage control block with the calling task. This control block will be used in the broadcast command **GS_BROADCAST**.

GS_CLEAR_BC_CB
Clears the broadcast guarded storage control block. The guarded storage control block will no longer have the association established by the **GS_SET_BC_CB** command.

GS_BROADCAST
Sends a broadcast to all thread siblings of the calling task. Every sibling that has established a broadcast guarded storage control block will load this control block and will be enabled for guarded storage. The broadcast guarded storage control block is consumed; a second broadcast without a refresh of the stored control block with **GS_SET_BC_CB** will not have any effect.

The *gs_cb* argument specifies the address of a guarded storage control block structure and is currently used only by the **GS_SET_BC_CB** command; all other aforementioned commands ignore this argument.

RETURN VALUE

On success, the return value of **s390_guarded_storage**() is 0.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
command was **GS_SET_BC_CB** and the copying of the guarded storage control block structure pointed by the *gs_cb* argument has failed.

EINVAL
The value provided in the *command* argument was not valid.

ENOMEM
command was one of **GS_ENABLE** or **GS_SET_BC_CB**, and the allocation of a new guarded storage control block has failed.

EOPNOTSUPP
The guarded storage facility is not supported by the hardware.

STANDARDS
Linux on s390.

HISTORY
Linux 4.12. System z14.

NOTES
The description of the guarded storage facility along with related instructions and Guarded Storage Control Block and Guarded Storage Event Parameter List structure layouts is available in "z/Architecture Principles of Operations" beginning from the twelfth edition.

The *gs_cb* structure has a field *gsepla* (Guarded Storage Event Parameter List Address), which is a user-space pointer to a Guarded Storage Event Parameter List structure (that contains the address of the aforementioned event handler in the *gseha* field), and its layout is available as a **gs_epl** structure type definition in the *asm/guarded_storage.h* header.

SEE ALSO
syscall(2)

NAME

s390_pci_mmio_write, s390_pci_mmio_read − transfer data to/from PCI MMIO memory page

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_s390_pci_mmio_write, unsigned long mmio_addr,
            const void user_buffer[.length], size_t length);
int syscall(SYS_s390_pci_mmio_read, unsigned long mmio_addr,
            void user_buffer[.length], size_t length);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

The **s390_pci_mmio_write**() system call writes *length* bytes of data from the user-space buffer *user_buffer* to the PCI MMIO memory location specified by *mmio_addr*. The **s390_pci_mmio_read**() system call reads *length* bytes of data from the PCI MMIO memory location specified by *mmio_addr* to the user-space buffer *user_buffer*.

These system calls must be used instead of the simple assignment or data-transfer operations that are used to access the PCI MMIO memory areas mapped to user space on the Linux System z platform. The address specified by *mmio_addr* must belong to a PCI MMIO memory page mapping in the caller's address space, and the data being written or read must not cross a page boundary. The *length* value cannot be greater than the system page size.

RETURN VALUE

On success, **s390_pci_mmio_write**() and **s390_pci_mmio_read**() return 0. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
: The address in *mmio_addr* is invalid.

EFAULT
: *user_buffer* does not point to a valid location in the caller's address space.

EINVAL
: Invalid *length* argument.

ENODEV
: PCI support is not enabled.

ENOMEM
: Insufficient memory.

STANDARDS

Linux on s390.

HISTORY

Linux 3.19. System z EC12.

SEE ALSO

syscall(2)

NAME

s390_runtime_instr − enable/disable s390 CPU run-time instrumentation

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <asm/runtime_instr.h> /* Definition of **S390_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_s390_runtime_instr, int *command***, int** *signum***);**

Note: glibc provides no wrapper for **s390_runtime_instr**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **s390_runtime_instr**() system call starts or stops CPU run-time instrumentation for the calling thread.

The *command* argument controls whether run-time instrumentation is started (**S390_RUNTIME_INSTR_START**, 1) or stopped (**S390_RUNTIME_INSTR_STOP**, 2) for the calling thread.

The *signum* argument specifies the number of a real-time signal. This argument was used to specify a signal number that should be delivered to the thread if the run-time instrumentation buffer was full or if the run-time-instrumentation-halted interrupt had occurred. This feature was never used, and in Linux 4.4 support for this feature was removed; thus, in current kernels, this argument is ignored.

RETURN VALUE

On success, **s390_runtime_instr**() returns 0 and enables the thread for run-time instrumentation by assigning the thread a default run-time instrumentation control block. The caller can then read and modify the control block and start the run-time instrumentation. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
The value specified in *command* is not a valid command.

EINVAL
The value specified in *signum* is not a real-time signal number. From Linux 4.4 onwards, the *signum* argument has no effect, so that an invalid signal number will not result in an error.

ENOMEM
Allocating memory for the run-time instrumentation control block failed.

EOPNOTSUPP
The run-time instrumentation facility is not available.

STANDARDS

Linux on s390.

HISTORY

Linux 3.7. System z EC12.

NOTES

The *asm/runtime_instr.h* header file is available since Linux 4.16.

Starting with Linux 4.4, support for signalling was removed, as was the check whether *signum* is a valid real-time signal. For backwards compatibility with older kernels, it is recommended to pass a valid real-time signal number in *signum* and install a handler for that signal.

SEE ALSO

syscall(2), *signal*(7)

NAME

s390_sthyi − emulate STHYI instruction

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <asm/sthyi.h>      /* Definition of STHYI_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_s390_sthyi, unsigned long  function_code,
        void *resp_buffer, uint64_t *return_code,
        unsigned long  flags);
```

Note: glibc provides no wrapper for **s390_sthyi**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **s390_sthyi**() system call emulates the STHYI (Store Hypervisor Information) instruction. It provides hardware resource information for the machine and its virtualization levels. This includes CPU type and capacity, as well as the machine model and other metrics.

The *function_code* argument indicates which function to perform. The following code(s) are supported:

STHYI_FC_CP_IFL_CAP
: Return CP (Central Processor) and IFL (Integrated Facility for Linux) capacity information.

The *resp_buffer* argument specifies the address of a response buffer. When the *function_code* is **STHYI_FC_CP_IFL_CAP**, the buffer must be one page (4K) in size. If the system call returns 0, the response buffer will be filled with CPU capacity information. Otherwise, the response buffer's content is unchanged.

The *return_code* argument stores the return code of the STHYI instruction, using one of the following values:

Success.

4 Unsupported function code.

For further details about *return_code*, *function_code*, and *resp_buffer*, see the reference given in NOTES.

The *flags* argument is provided to allow for future extensions and currently must be set to 0.

RETURN VALUE

On success (that is: emulation succeeded), the return value of **s390_sthyi**() matches the condition code of the STHYI instructions, which is a value in the range [0..3]. A return value of 0 indicates that CPU capacity information is stored in **resp_buffer*. A return value of 3 indicates "unsupported function code" and the content of **resp_buffer* is unchanged. The return values 1 and 2 are reserved.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: The value specified in *resp_buffer* or *return_code* is not a valid address.

EINVAL
: The value specified in *flags* is nonzero.

ENOMEM
: Allocating memory for handling the CPU capacity information failed.

EOPNOTSUPP
: The value specified in *function_code* is not valid.

STANDARDS

Linux on s390.

HISTORY

Linux 4.15.

NOTES

For details of the STHYI instruction, see the documentation page.

When the system call interface is used, the response buffer doesn't have to fulfill alignment requirements described in the STHYI instruction definition.

The kernel caches the response (for up to one second, as of Linux 4.16). Subsequent system call invocations may return the cached response.

SEE ALSO

syscall(2)

NAME

sched_get_priority_max, sched_get_priority_min − get static priority range

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION

sched_get_priority_max() returns the maximum priority value that can be used with the scheduling algorithm identified by *policy*. **sched_get_priority_min**() returns the minimum priority value that can be used with the scheduling algorithm identified by *policy*. Supported *policy* values are **SCHED_FIFO**, **SCHED_RR**, **SCHED_OTHER**, **SCHED_BATCH**, **SCHED_IDLE**, and **SCHED_DEADLINE**. Further details about these policies can be found in *sched*(7).

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. Thus, the value returned by **sched_get_priority_max**() will be greater than the value returned by **sched_get_priority_min**().

Linux allows the static priority range 1 to 99 for the **SCHED_FIFO** and **SCHED_RR** policies, and the priority 0 for the remaining policies. Scheduling priority ranges for the various policies are not alterable.

The range of scheduling priorities may vary on other POSIX systems, thus it is a good idea for portable applications to use a virtual priority range and map it to the interval given by **sched_get_priority_max**() and **sched_get_priority_min**() POSIX.1 requires a spread of at least 32 between the maximum and the minimum values for **SCHED_FIFO** and **SCHED_RR**.

POSIX systems on which **sched_get_priority_max**() and **sched_get_priority_min**() are available define **_POSIX_PRIORITY_SCHEDULING** in <*unistd.h*>.

RETURN VALUE

On success, **sched_get_priority_max**() and **sched_get_priority_min**() return the maximum/minimum priority value for the named scheduling policy. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: The argument *policy* does not identify a defined scheduling policy.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

sched_getaffinity(2), *sched_getparam*(2), *sched_getscheduler*(2), *sched_setaffinity*(2), *sched_setparam*(2), *sched_setscheduler*(2), *sched*(7)

NAME

sched_rr_get_interval − get the SCHED_RR interval for the named process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sched.h>

int sched_rr_get_interval(pid_t *pid***, struct timespec ****tp***);**

DESCRIPTION

sched_rr_get_interval() writes into the *timespec*(3) structure pointed to by *tp* the round-robin time quantum for the process identified by *pid*. The specified process should be running under the **SCHED_RR** scheduling policy.

If *pid* is zero, the time quantum for the calling process is written into **tp*.

RETURN VALUE

On success, **sched_rr_get_interval**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
Problem with copying information to user space.

EINVAL
Invalid pid.

ENOSYS
The system call is not yet implemented (only on rather old kernels).

ESRCH
Could not find a process with the ID *pid*.

VERSIONS

Linux

Linux 3.9 added a new mechanism for adjusting (and viewing) the **SCHED_RR** quantum: the */proc/sys/kernel/sched_rr_timeslice_ms* file exposes the quantum as a millisecond value, whose default is 100. Writing 0 to this file resets the quantum to the default value.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Linux

POSIX does not specify any mechanism for controlling the size of the round-robin time quantum. Older Linux kernels provide a (nonportable) method of doing this. The quantum can be controlled by adjusting the process's nice value (see *setpriority*(2)). Assigning a negative (i.e., high) nice value results in a longer quantum; assigning a positive (i.e., low) nice value results in a shorter quantum. The default quantum is 0.1 seconds; the degree to which changing the nice value affects the quantum has varied somewhat across kernel versions. This method of adjusting the quantum was removed starting with Linux 2.6.24.

NOTES

POSIX systems on which **sched_rr_get_interval**() is available define **_POSIX_PRIORITY_SCHEDULING** in <*unistd.h*>.

SEE ALSO

timespec(3), *sched*(7)

NAME

sched_setaffinity, sched_getaffinity − set and get a thread's CPU affinity mask

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t cpusetsize,
                      const cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);
```

DESCRIPTION

A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits. For example, by dedicating one CPU to a particular thread (i.e., setting the affinity mask of that thread to specify a single CPU, and setting the affinity mask of all other threads to exclude that CPU), it is possible to ensure maximum execution speed for that thread. Restricting a thread to run on a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to execute on one CPU and then recommences execution on a different CPU.

A CPU affinity mask is represented by the *cpu_set_t* structure, a "CPU set", pointed to by *mask*. A set of macros for manipulating CPU sets is described in *CPU_SET*(3).

sched_setaffinity() sets the CPU affinity mask of the thread whose ID is *pid* to the value specified by *mask*. If *pid* is zero, then the calling thread is used. The argument *cpusetsize* is the length (in bytes) of the data pointed to by *mask*. Normally this argument would be specified as *sizeof(cpu_set_t)*.

If the thread specified by *pid* is not currently running on one of the CPUs specified in *mask*, then that thread is migrated to one of the CPUs specified in *mask*.

sched_getaffinity() writes the affinity mask of the thread whose ID is *pid* into the *cpu_set_t* structure pointed to by *mask*. The *cpusetsize* argument specifies the size (in bytes) of *mask*. If *pid* is zero, then the mask of the calling thread is returned.

RETURN VALUE

On success, **sched_setaffinity**() and **sched_getaffinity**() return 0 (but see "C library/kernel differences" below, which notes that the underlying **sched_getaffinity**() differs in its return value). On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: A supplied memory address was invalid.

EINVAL
: The affinity bit mask *mask* contains no processors that are currently physically on the system and permitted to the thread according to any restrictions that may be imposed by *cpuset* cgroups or the "cpuset" mechanism described in *cpuset*(7).

EINVAL
: (**sched_getaffinity**() and, before Linux 2.6.9, *sched_setaffinity*()) *cpusetsize* is smaller than the size of the affinity mask used by the kernel.

EPERM
: (**sched_setaffinity**()) The calling thread does not have appropriate privileges. The caller needs an effective user ID equal to the real user ID or effective user ID of the thread identified by *pid*, or it must possess the **CAP_SYS_NICE** capability in the user namespace of the thread *pid*.

ESRCH
: The thread whose ID is *pid* could not be found.

STANDARDS

Linux.

HISTORY

Linux 2.5.8, glibc 2.3.

Initially, the glibc interfaces included a *cpusetsize* argument, typed as *unsigned int*. In glibc 2.3.3, the *cpusetsize* argument was removed, but was then restored in glibc 2.3.4, with type *size_t*.

NOTES

After a call to **sched_setaffinity**(), the set of CPUs on which the thread will actually run is the intersection of the set specified in the *mask* argument and the set of CPUs actually present on the system. The system may further restrict the set of CPUs on which the thread runs if the "cpuset" mechanism described in *cpuset*(7) is being used. These restrictions on the actual set of CPUs on which the thread will run are silently imposed by the kernel.

There are various ways of determining the number of CPUs available on the system, including: inspecting the contents of */proc/cpuinfo*; using *sysconf*(3) to obtain the values of the **_SC_NPROCESSORS_CONF** and **_SC_NPROCESSORS_ONLN** parameters; and inspecting the list of CPU directories under */sys/devices/system/cpu/*.

sched(7) has a description of the Linux scheduling scheme.

The affinity mask is a per-thread attribute that can be adjusted independently for each of the threads in a thread group. The value returned from a call to *gettid*(2) can be passed in the argument *pid*. Specifying *pid* as 0 will set the attribute for the calling thread, and passing the value returned from a call to *getpid*(2) will set the attribute for the main thread of the thread group. (If you are using the POSIX threads API, then use *pthread_setaffinity_np*(3) instead of *sched_setaffinity*().)

The *isolcpus* boot option can be used to isolate one or more CPUs at boot time, so that no processes are scheduled onto those CPUs. Following the use of this boot option, the only way to schedule processes onto the isolated CPUs is via **sched_setaffinity**() or the *cpuset*(7) mechanism. For further information, see the kernel source file *Documentation/admin−guide/kernel−parameters.txt*. As noted in that file, *isolcpus* is the preferred mechanism of isolating CPUs (versus the alternative of manually setting the CPU affinity of all processes on the system).

A child created via *fork*(2) inherits its parent's CPU affinity mask. The affinity mask is preserved across an *execve*(2).

C library/kernel differences

This manual page describes the glibc interface for the CPU affinity calls. The actual system call interface is slightly different, with the *mask* being typed as *unsigned long* *, reflecting the fact that the underlying implementation of CPU sets is a simple bit mask.

On success, the raw **sched_getaffinity**() system call returns the number of bytes placed copied into the *mask* buffer; this will be the minimum of *cpusetsize* and the size (in bytes) of the *cpumask_t* data type that is used internally by the kernel to represent the CPU set bit mask.

Handling systems with large CPU affinity masks

The underlying system calls (which represent CPU masks as bit masks of type *unsigned long* *) impose no restriction on the size of the CPU mask. However, the *cpu_set_t* data type used by glibc has a fixed size of 128 bytes, meaning that the maximum CPU number that can be represented is 1023. If the kernel CPU affinity mask is larger than 1024, then calls of the form:

```
sched_getaffinity(pid, sizeof(cpu_set_t), &mask);
```

fail with the error **EINVAL**, the error produced by the underlying system call for the case where the *mask* size specified in *cpusetsize* is smaller than the size of the affinity mask used by the kernel. (Depending on the system CPU topology, the kernel affinity mask can be substantially larger than the number of active CPUs in the system.)

When working on systems with large kernel CPU affinity masks, one must dynamically allocate the *mask* argument (see *CPU_ALLOC*(3)). Currently, the only way to do this is by probing for the size of the required mask using **sched_getaffinity**() calls with increasing mask sizes (until the call does not fail with the error **EINVAL**).

Be aware that *CPU_ALLOC*(3) may allocate a slightly larger CPU set than requested (because CPU sets are implemented as bit masks allocated in units of *sizeof(long)*). Consequently, **sched_getaffinity**() can set bits beyond the requested allocation size, because the kernel sees a few additional bits. Therefore, the caller should iterate over the bits in the returned set, counting those which are set, and

stop upon reaching the value returned by *CPU_COUNT*(3) (rather than iterating over the number of bits requested to be allocated).

EXAMPLES

The program below creates a child process. The parent and child then each assign themselves to a specified CPU and execute identical loops that consume some CPU time. Before terminating, the parent waits for the child to complete. The program takes three command-line arguments: the CPU number for the parent, the CPU number for the child, and the number of loop iterations that both processes should perform.

As the sample runs below demonstrate, the amount of real and CPU time consumed when running the program will depend on intra-core caching effects and whether the processes are using the same CPU.

We first employ *lscpu*(1) to determine that this (x86) system has two cores, each with two CPUs:

```
$ lscpu | egrep -i 'core.*:|socket'
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
```

We then time the operation of the example program for three cases: both processes running on the same CPU; both processes running on different CPUs on the same core; and both processes running on different CPUs on different cores.

```
$ time -p ./a.out 0 0 100000000
real 14.75
user 3.02
sys 11.73
$ time -p ./a.out 0 1 100000000
real 11.52
user 3.98
sys 19.06
$ time -p ./a.out 0 3 100000000
real 7.89
user 3.29
sys 12.07
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int           parentCPU, childCPU;
    cpu_set_t     set;
    unsigned int  nloops;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s parent-cpu child-cpu num-loops\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    parentCPU = atoi(argv[1]);
    childCPU = atoi(argv[2]);
    nloops = atoi(argv[3]);
```

```
        CPU_ZERO(&set);

        switch (fork()) {
        case -1:            /* Error */
            err(EXIT_FAILURE, "fork");

        case 0:             /* Child */
            CPU_SET(childCPU, &set);

            if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
                err(EXIT_FAILURE, "sched_setaffinity");

            for (unsigned int j = 0; j < nloops; j++)
                getppid();

            exit(EXIT_SUCCESS);

        default:            /* Parent */
            CPU_SET(parentCPU, &set);

            if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
                err(EXIT_FAILURE, "sched_setaffinity");

            for (unsigned int j = 0; j < nloops; j++)
                getppid();

            wait(NULL);     /* Wait for child to terminate */
            exit(EXIT_SUCCESS);
        }
    }
```

SEE ALSO

lscpu(1), *nproc*(1), *taskset*(1), *clone*(2), *getcpu*(2), *getpriority*(2), *gettid*(2), *nice*(2), *sched_get_priority_max*(2), *sched_get_priority_min*(2), *sched_getscheduler*(2), *sched_setscheduler*(2), *setpriority*(2), *CPU_SET*(3), *get_nprocs*(3), *pthread_setaffinity_np*(3), *sched_getcpu*(3), *capabilities*(7), *cpuset*(7), *sched*(7), *numactl*(8)

NAME

sched_setattr, sched_getattr − set and get scheduling policy and attributes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sched.h>          /* Definition of SCHED_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_sched_setattr, pid_t pid, struct sched_attr *attr,
        unsigned int flags);
int syscall(SYS_sched_getattr, pid_t pid, struct sched_attr *attr,
        unsigned int size, unsigned int flags);
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

sched_setattr()

The **sched_setattr**() system call sets the scheduling policy and associated attributes for the thread whose ID is specified in *pid*. If *pid* equals zero, the scheduling policy and attributes of the calling thread will be set.

Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies as values that may be specified in *policy*:

SCHED_OTHER
: the standard round-robin time-sharing policy;

SCHED_BATCH
: for "batch" style execution of processes; and

SCHED_IDLE for running *very* low priority background jobs.

Various "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable threads are selected for execution. For the rules governing when a process may use these policies, see *sched*(7). The real-time policies that may be specified in *policy* are:

SCHED_FIFO a first-in, first-out policy; and

SCHED_RR a round-robin policy.

Linux also provides the following policy:

SCHED_DEADLINE
: a deadline scheduling policy; see *sched*(7) for details.

The *attr* argument is a pointer to a structure that defines the new scheduling policy and attributes for the specified thread. This structure has the following form:

```
struct sched_attr {
    u32 size;              /* Size of this structure */
    u32 sched_policy;      /* Policy (SCHED_*) */
    u64 sched_flags;       /* Flags */
    s32 sched_nice;        /* Nice value (SCHED_OTHER,
                              SCHED_BATCH) */
    u32 sched_priority;    /* Static priority (SCHED_FIFO,
                              SCHED_RR) */
    /* Remaining fields are for SCHED_DEADLINE */
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;
};
```

The fields of the *sched_attr* structure are as follows:

size This field should be set to the size of the structure in bytes, as in *sizeof(struct sched_attr)*. If the provided structure is smaller than the kernel structure, any additional fields are assumed to be '0'. If the provided structure is larger than the kernel structure, the kernel verifies that all additional fields are 0; if they are not, **sched_setattr**() fails with the error **E2BIG** and updates *size* to contain the size of the kernel structure.

The above behavior when the size of the user-space *sched_attr* structure does not match the size of the kernel structure allows for future extensibility of the interface. Malformed applications that pass oversize structures won't break in the future if the size of the kernel *sched_attr* structure is increased. In the future, it could also allow applications that know about a larger user-space *sched_attr* structure to determine whether they are running on an older kernel that does not support the larger structure.

sched_policy
This field specifies the scheduling policy, as one of the **SCHED_*** values listed above.

sched_flags
This field contains zero or more of the following flags that are ORed together to control scheduling behavior:

SCHED_FLAG_RESET_ON_FORK
Children created by *fork*(2) do not inherit privileged scheduling policies. See *sched*(7) for details.

SCHED_FLAG_RECLAIM (since Linux 4.13)
This flag allows a **SCHED_DEADLINE** thread to reclaim bandwidth unused by other real-time threads.

SCHED_FLAG_DL_OVERRUN (since Linux 4.16)
This flag allows an application to get informed about run-time overruns in **SCHED_DEADLINE** threads. Such overruns may be caused by (for example) coarse execution time accounting or incorrect parameter assignment. Notification takes the form of a **SIGXCPU** signal which is generated on each overrun.

This **SIGXCPU** signal is *process-directed* (see *signal*(7)) rather than thread-directed. This is probably a bug. On the one hand, **sched_setattr**() is being used to set a per-thread attribute. On the other hand, if the process-directed signal is delivered to a thread inside the process other than the one that had a run-time overrun, the application has no way of knowing which thread overran.

sched_nice
This field specifies the nice value to be set when specifying *sched_policy* as **SCHED_OTHER** or **SCHED_BATCH**. The nice value is a number in the range −20 (high priority) to +19 (low priority); see *sched*(7).

sched_priority
This field specifies the static priority to be set when specifying *sched_policy* as **SCHED_FIFO** or **SCHED_RR**. The allowed range of priorities for these policies can be determined using *sched_get_priority_min*(2) and *sched_get_priority_max*(2). For other policies, this field must be specified as 0.

sched_runtime
This field specifies the "Runtime" parameter for deadline scheduling. The value is expressed in nanoseconds. This field, and the next two fields, are used only for **SCHED_DEADLINE** scheduling; for further details, see *sched*(7).

sched_deadline
This field specifies the "Deadline" parameter for deadline scheduling. The value is expressed in nanoseconds.

sched_period
This field specifies the "Period" parameter for deadline scheduling. The value is expressed in nanoseconds.

The *flags* argument is provided to allow for future extensions to the interface; in the current implementation it must be specified as 0.

sched_getattr()

The **sched_getattr**() system call fetches the scheduling policy and the associated attributes for the thread whose ID is specified in *pid*. If *pid* equals zero, the scheduling policy and attributes of the calling thread will be retrieved.

The *size* argument should be set to the size of the *sched_attr* structure as known to user space. The value must be at least as large as the size of the initially published *sched_attr* structure, or the call fails with the error **EINVAL**.

The retrieved scheduling attributes are placed in the fields of the *sched_attr* structure pointed to by *attr*. The kernel sets *attr.size* to the size of its *sched_attr* structure.

If the caller-provided *attr* buffer is larger than the kernel's *sched_attr* structure, the additional bytes in the user-space structure are not touched. If the caller-provided structure is smaller than the kernel *sched_attr* structure, the kernel will silently not return any values which would be stored outside the provided space. As with **sched_setattr**(), these semantics allow for future extensibility of the interface.

The *flags* argument is provided to allow for future extensions to the interface; in the current implementation it must be specified as 0.

RETURN VALUE

On success, **sched_setattr**() and **sched_getattr**() return 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

sched_getattr() and **sched_setattr**() can both fail for the following reasons:

EINVAL
attr is NULL; or *pid* is negative; or *flags* is not zero.

ESRCH
The thread whose ID is *pid* could not be found.

In addition, **sched_getattr**() can fail for the following reasons:

E2BIG The buffer specified by *size* and *attr* is too small.

EINVAL
size is invalid; that is, it is smaller than the initial version of the *sched_attr* structure (48 bytes) or larger than the system page size.

In addition, **sched_setattr**() can fail for the following reasons:

E2BIG The buffer specified by *size* and *attr* is larger than the kernel structure, and one or more of the excess bytes is nonzero.

EBUSY
SCHED_DEADLINE admission control failure, see *sched*(7).

EINVAL
attr.sched_policy is not one of the recognized policies; *attr.sched_flags* contains a flag other than **SCHED_FLAG_RESET_ON_FORK**; or *attr.sched_priority* is invalid; or *attr.sched_policy* is **SCHED_DEADLINE** and the deadline scheduling parameters in *attr* are invalid.

EPERM
The caller does not have appropriate privileges.

EPERM
The CPU affinity mask of the thread specified by *pid* does not include all CPUs in the system (see *sched_setaffinity*(2)).

STANDARDS

Linux.

HISTORY

Linux 3.14.

NOTES

glibc does not provide wrappers for these system calls; call them using *syscall*(2).

sched_setattr() provides a superset of the functionality of *sched_setscheduler*(2), *sched_setparam*(2), *nice*(2), and (other than the ability to set the priority of all processes belonging to a specified user or all processes in a specified group) *setpriority*(2). Analogously, **sched_getattr**() provides a superset of the functionality of *sched_getscheduler*(2), *sched_getparam*(2), and (partially) *getpriority*(2).

BUGS

In Linux versions up to 3.15, **sched_setattr**() failed with the error **EFAULT** instead of **E2BIG** for the case described in ERRORS.

Up to Linux 5.3, **sched_getattr**() failed with the error **EFBIG** if the in-kernel *sched_attr* structure was larger than the *size* passed by user space.

SEE ALSO

chrt(1), *nice*(2), *sched_get_priority_max*(2), *sched_get_priority_min*(2), *sched_getaffinity*(2), *sched_getparam*(2), *sched_getscheduler*(2), *sched_rr_get_interval*(2), *sched_setaffinity*(2), *sched_setparam*(2), *sched_setscheduler*(2), *sched_yield*(2), *setpriority*(2), *pthread_getschedparam*(3), *pthread_setschedparam*(3), *pthread_setschedprio*(3), *capabilities*(7), *cpuset*(7), *sched*(7)

NAME

sched_setparam, sched_getparam − set and get scheduling parameters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

DESCRIPTION

sched_setparam() sets the scheduling parameters associated with the scheduling policy for the thread whose thread ID is specified in *pid*. If *pid* is zero, then the parameters of the calling thread are set. The interpretation of the argument *param* depends on the scheduling policy of the thread identified by *pid*. See *sched*(7) for a description of the scheduling policies supported under Linux.

sched_getparam() retrieves the scheduling parameters for the thread identified by *pid*. If *pid* is zero, then the parameters of the calling thread are retrieved.

sched_setparam() checks the validity of *param* for the scheduling policy of the thread. The value *param−>sched_priority* must lie within the range given by *sched_get_priority_min*(2) and *sched_get_priority_max*(2).

For a discussion of the privileges and resource limits related to scheduling priority and policy, see *sched*(7).

POSIX systems on which **sched_setparam**() and **sched_getparam**() are available define **_POSIX_PRIORITY_SCHEDULING** in *<unistd.h>*.

RETURN VALUE

On success, **sched_setparam**() and **sched_getparam**() return 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: Invalid arguments: *param* is NULL or *pid* is negative

EINVAL
: (**sched_setparam**()) The argument *param* does not make sense for the current scheduling policy.

EPERM
: (**sched_setparam**()) The caller does not have appropriate privileges (Linux: does not have the **CAP_SYS_NICE** capability).

ESRCH
: The thread whose ID is *pid* could not be found.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

getpriority(2), *gettid*(2), *nice*(2), *sched_get_priority_max*(2), *sched_get_priority_min*(2), *sched_getaffinity*(2), *sched_getscheduler*(2), *sched_setaffinity*(2), *sched_setattr*(2), *sched_setscheduler*(2), *setpriority*(2), *capabilities*(7), *sched*(7)

NAME

sched_setscheduler, sched_getscheduler − set and get scheduling policy/parameters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

DESCRIPTION

The **sched_setscheduler**() system call sets both the scheduling policy and parameters for the thread whose ID is specified in *pid*. If *pid* equals zero, the scheduling policy and parameters of the calling thread will be set.

The scheduling parameters are specified in the *param* argument, which is a pointer to a structure of the following form:

```
struct sched_param {
    ...
    int sched_priority;
    ...
};
```

In the current implementation, the structure contains only one field, *sched_priority*. The interpretation of *param* depends on the selected policy.

Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies as values that may be specified in *policy*:

SCHED_OTHER
the standard round-robin time-sharing policy;

SCHED_BATCH
for "batch" style execution of processes; and

SCHED_IDLE for running *very* low priority background jobs.

For each of the above policies, *param−>sched_priority* must be 0.

Various "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable threads are selected for execution. For the rules governing when a process may use these policies, see *sched*(7). The real-time policies that may be specified in *policy* are:

SCHED_FIFO a first-in, first-out policy; and

SCHED_RR a round-robin policy.

For each of the above policies, *param−>sched_priority* specifies a scheduling priority for the thread. This is a number in the range returned by calling *sched_get_priority_min*(2) and *sched_get_priority_max*(2) with the specified *policy*. On Linux, these system calls return, respectively, 1 and 99.

Since Linux 2.6.32, the **SCHED_RESET_ON_FORK** flag can be ORed in *policy* when calling **sched_setscheduler**(). As a result of including this flag, children created by *fork*(2) do not inherit privileged scheduling policies. See *sched*(7) for details.

sched_getscheduler() returns the current scheduling policy of the thread identified by *pid*. If *pid* equals zero, the policy of the calling thread will be retrieved.

RETURN VALUE

On success, **sched_setscheduler**() returns zero. On success, **sched_getscheduler**() returns the policy for the thread (a nonnegative integer). On error, both calls return −1, and *errno* is set to indicate the error.

ERRORS

EINVAL
Invalid arguments: *pid* is negative or *param* is NULL.

EINVAL
(**sched_setscheduler**()) *policy* is not one of the recognized policies.

EINVAL
(**sched_setscheduler**()) *param* does not make sense for the specified *policy*.

EPERM
The calling thread does not have appropriate privileges.

ESRCH
The thread whose ID is *pid* could not be found.

VERSIONS

POSIX.1 does not detail the permissions that an unprivileged thread requires in order to call **sched_setscheduler**(), and details vary across systems. For example, the Solaris 7 manual page says that the real or effective user ID of the caller must match the real user ID or the save set-user-ID of the target.

The scheduling policy and parameters are in fact per-thread attributes on Linux. The value returned from a call to *gettid*(2) can be passed in the argument *pid*. Specifying *pid* as 0 will operate on the attributes of the calling thread, and passing the value returned from a call to *getpid*(2) will operate on the attributes of the main thread of the thread group. (If you are using the POSIX threads API, then use *pthread_setschedparam*(3), *pthread_getschedparam*(3), and *pthread_setschedprio*(3), instead of the **sched_***(2) system calls.)

STANDARDS

POSIX.1-2008 (but see BUGS below).

SCHED_BATCH and **SCHED_IDLE** are Linux-specific.

HISTORY

POSIX.1-2001.

NOTES

Further details of the semantics of all of the above "normal" and "real-time" scheduling policies can be found in the *sched*(7) manual page. That page also describes an additional policy, **SCHED_DEADLINE**, which is settable only via *sched_setattr*(2).

POSIX systems on which **sched_setscheduler**() and **sched_getscheduler**() are available define **_POSIX_PRIORITY_SCHEDULING** in <*unistd.h*>.

BUGS

POSIX.1 says that on success, **sched_setscheduler**() should return the previous scheduling policy. Linux **sched_setscheduler**() does not conform to this requirement, since it always returns 0 on success.

SEE ALSO

chrt(1), *nice*(2), *sched_get_priority_max*(2), *sched_get_priority_min*(2), *sched_getaffinity*(2), *sched_getattr*(2), *sched_getparam*(2), *sched_rr_get_interval*(2), *sched_setaffinity*(2), *sched_setattr*(2), *sched_setparam*(2), *sched_yield*(2), *setpriority*(2), *capabilities*(7), *cpuset*(7), *sched*(7)

NAME

sched_yield − yield the processor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sched.h>

int sched_yield(void);

DESCRIPTION

sched_yield() causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

RETURN VALUE

On success, **sched_yield**() returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

In the Linux implementation, **sched_yield**() always succeeds.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001 (but optional). POSIX.1-2008.

Before POSIX.1-2008, systems on which **sched_yield**() is available defined **_POSIX_PRIORITY_SCHEDULING** in *<unistd.h>*.

CAVEATS

sched_yield() is intended for use with real-time scheduling policies (i.e., **SCHED_FIFO** or **SCHED_RR**). Use of **sched_yield**() with nondeterministic scheduling policies such as **SCHED_OTHER** is unspecified and very likely means your application design is broken.

If the calling thread is the only thread in the highest priority list at that time, it will continue to run after a call to **sched_yield**().

Avoid calling **sched_yield**() unnecessarily or inappropriately (e.g., when resources needed by other schedulable threads are still held by the caller), since doing so will result in unnecessary context switches, which will degrade system performance.

SEE ALSO

sched(7)

NAME

seccomp − operate on Secure Computing state of the process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/seccomp.h>  /* Definition of SECCOMP_* constants */
#include <linux/filter.h>   /* Definition of struct sock_fprog */
#include <linux/audit.h>    /* Definition of AUDIT_* constants */
#include <linux/signal.h>   /* Definition of SIG* constants */
#include <sys/ptrace.h>     /* Definition of PTRACE_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_seccomp, unsigned int operation, unsigned int flags,
        void *args);
```

Note: glibc provides no wrapper for **seccomp**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **seccomp**() system call operates on the Secure Computing (seccomp) state of the calling process.

Currently, Linux supports the following *operation* values:

SECCOMP_SET_MODE_STRICT

The only system calls that the calling thread is permitted to make are *read*(2), *write*(2), *_exit*(2) (but not *exit_group*(2)), and *sigreturn*(2). Other system calls result in the termination of the calling thread, or termination of the entire process with the **SIGKILL** signal when there is only one thread. Strict secure computing mode is useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket.

Note that although the calling thread can no longer call *sigprocmask*(2), it can use *sigreturn*(2) to block all signals apart from **SIGKILL** and **SIGSTOP**. This means that *alarm*(2) (for example) is not sufficient for restricting the process's execution time. Instead, to reliably terminate the process, **SIGKILL** must be used. This can be done by using *timer_create*(2) with **SIGEV_SIGNAL** and *sigev_signo* set to **SIGKILL**, or by using *setrlimit*(2) to set the hard limit for **RLIMIT_CPU**.

This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

The value of *flags* must be 0, and *args* must be NULL.

This operation is functionally identical to the call:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
```

SECCOMP_SET_MODE_FILTER

The system calls allowed are defined by a pointer to a Berkeley Packet Filter (BPF) passed via *args*. This argument is a pointer to a *struct sock_fprog*; it can be designed to filter arbitrary system calls and system call arguments. If the filter is invalid, **seccomp**() fails, returning **EINVAL** in *errno*.

If *fork*(2) or *clone*(2) is allowed by the filter, any child processes will be constrained to the same system call filters as the parent. If *execve*(2) is allowed, the existing filters will be preserved across a call to *execve*(2).

In order to use the **SECCOMP_SET_MODE_FILTER** operation, either the calling thread must have the **CAP_SYS_ADMIN** capability in its user namespace, or the thread must already have the *no_new_privs* bit set. If that bit was not already set by an ancestor of this thread, the thread must make the following call:

```
prctl(PR_SET_NO_NEW_PRIVS, 1);
```

Otherwise, the **SECCOMP_SET_MODE_FILTER** operation fails and returns **EACCES** in *errno*. This requirement ensures that an unprivileged process cannot apply a malicious filter and then invoke a set-user-ID or other privileged program using *execve*(2), thus potentially

compromising that program. (Such a malicious filter might, for example, cause an attempt to use *setuid*(2) to set the caller's user IDs to nonzero values to instead return 0 without actually making the system call. Thus, the program might be tricked into retaining superuser privileges in circumstances where it is possible to influence it to do dangerous things because it did not actually drop privileges.)

If *prctl*(2) or **seccomp**() is allowed by the attached filter, further filters may be added. This will increase evaluation time, but allows for further reduction of the attack surface during execution of a thread.

The **SECCOMP_SET_MODE_FILTER** operation is available only if the kernel is configured with **CONFIG_SECCOMP_FILTER** enabled.

When *flags* is 0, this operation is functionally identical to the call:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, args);
```

The recognized *flags* are:

SECCOMP_FILTER_FLAG_LOG (since Linux 4.14)
All filter return actions except **SECCOMP_RET_ALLOW** should be logged. An administrator may override this filter flag by preventing specific actions from being logged via the */proc/sys/kernel/seccomp/actions_logged* file.

SECCOMP_FILTER_FLAG_NEW_LISTENER (since Linux 5.0)
After successfully installing the filter program, return a new user-space notification file descriptor. (The close-on-exec flag is set for the file descriptor.) When the filter returns **SECCOMP_RET_USER_NOTIF** a notification will be sent to this file descriptor.

At most one seccomp filter using the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag can be installed for a thread.

See *seccomp_unotify*(2) for further details.

SECCOMP_FILTER_FLAG_SPEC_ALLOW (since Linux 4.17)
Disable Speculative Store Bypass mitigation.

SECCOMP_FILTER_FLAG_TSYNC
When adding a new filter, synchronize all other threads of the calling process to the same seccomp filter tree. A "filter tree" is the ordered list of filters attached to a thread. (Attaching identical filters in separate **seccomp**() calls results in different filters from this perspective.)

If any thread cannot synchronize to the same filter tree, the call will not attach the new seccomp filter, and will fail, returning the first thread ID found that cannot synchronize. Synchronization will fail if another thread in the same process is in **SECCOMP_MODE_STRICT** or if it has attached new seccomp filters to itself, diverging from the calling thread's filter tree.

SECCOMP_GET_ACTION_AVAIL (since Linux 4.14)
Test to see if an action is supported by the kernel. This operation is helpful to confirm that the kernel knows of a more recently added filter return action since the kernel treats all unknown actions as **SECCOMP_RET_KILL_PROCESS**.

The value of *flags* must be 0, and *args* must be a pointer to an unsigned 32-bit filter return action.

SECCOMP_GET_NOTIF_SIZES (since Linux 5.0)
Get the sizes of the seccomp user-space notification structures. Since these structures may evolve and grow over time, this command can be used to determine how much memory to allocate for sending and receiving notifications.

The value of *flags* must be 0, and *args* must be a pointer to a *struct seccomp_notif_sizes*, which has the following form:

```
struct seccomp_notif_sizes
    __u16 seccomp_notif;      /* Size of notification structure */
    __u16 seccomp_notif_resp; /* Size of response structure */
```

```
        __u16 seccomp_data;       /* Size of 'struct seccomp_data' */
    };
```

See *seccomp_unotify*(2) for further details.

Filters

When adding filters via **SECCOMP_SET_MODE_FILTER**, *args* points to a filter program:

```
struct sock_fprog {
    unsigned short      len;    /* Number of BPF instructions */
    struct sock_filter *filter; /* Pointer to array of
                                   BPF instructions */
};
```

Each program must contain one or more BPF instructions:

```
struct sock_filter {            /* Filter block */
    __u16 code;                 /* Actual filter code */
    __u8  jt;                   /* Jump true */
    __u8  jf;                   /* Jump false */
    __u32 k;                    /* Generic multiuse field */
};
```

When executing the instructions, the BPF program operates on the system call information made available (i.e., use the **BPF_ABS** addressing mode) as a (read-only) buffer of the following form:

```
struct seccomp_data {
    int   nr;                   /* System call number */
    __u32 arch;                 /* AUDIT_ARCH_* value
                                   (see <linux/audit.h>) */
    __u64 instruction_pointer;  /* CPU instruction pointer */
    __u64 args[6];              /* Up to 6 system call arguments */
};
```

Because numbering of system calls varies between architectures and some architectures (e.g., x86-64) allow user-space code to use the calling conventions of multiple architectures (and the convention being used may vary over the life of a process that uses *execve*(2) to execute binaries that employ the different conventions), it is usually necessary to verify the value of the *arch* field.

It is strongly recommended to use an allow-list approach whenever possible because such an approach is more robust and simple. A deny-list will have to be updated whenever a potentially dangerous system call is added (or a dangerous flag or option if those are deny-listed), and it is often possible to alter the representation of a value without altering its meaning, leading to a deny-list bypass. See also *Caveats* below.

The *arch* field is not unique for all calling conventions. The x86-64 ABI and the x32 ABI both use **AUDIT_ARCH_X86_64** as *arch*, and they run on the same processors. Instead, the mask **__X32_SYSCALL_BIT** is used on the system call number to tell the two ABIs apart.

This means that a policy must either deny all syscalls with **__X32_SYSCALL_BIT** or it must recognize syscalls with and without **__X32_SYSCALL_BIT** set. A list of system calls to be denied based on *nr* that does not also contain *nr* values with **__X32_SYSCALL_BIT** set can be bypassed by a malicious program that sets **__X32_SYSCALL_BIT**.

Additionally, kernels prior to Linux 5.4 incorrectly permitted *nr* in the ranges 512-547 as well as the corresponding non-x32 syscalls ORed with **__X32_SYSCALL_BIT**. For example, *nr* == 521 and *nr* == (101 | **__X32_SYSCALL_BIT**) would result in invocations of *ptrace*(2) with potentially confused x32-vs-x86_64 semantics in the kernel. Policies intended to work on kernels before Linux 5.4 must ensure that they deny or otherwise correctly handle these system calls. On Linux 5.4 and newer, such system calls will fail with the error **ENOSYS**, without doing anything.

The *instruction_pointer* field provides the address of the machine-language instruction that performed the system call. This might be useful in conjunction with the use of */proc/*pid*/maps* to perform checks based on which region (mapping) of the program made the system call. (Probably, it is wise to lock down the *mmap*(2) and *mprotect*(2) system calls to prevent the program from subverting such checks.)

When checking values from *args*, keep in mind that arguments are often silently truncated before being

processed, but after the seccomp check. For example, this happens if the i386 ABI is used on an x86-64 kernel: although the kernel will normally not look beyond the 32 lowest bits of the arguments, the values of the full 64-bit registers will be present in the seccomp data. A less surprising example is that if the x86-64 ABI is used to perform a system call that takes an argument of type *int*, the more-significant half of the argument register is ignored by the system call, but visible in the seccomp data.

A seccomp filter returns a 32-bit value consisting of two parts: the most significant 16 bits (corresponding to the mask defined by the constant **SECCOMP_RET_ACTION_FULL**) contain one of the "action" values listed below; the least significant 16-bits (defined by the constant **SECCOMP_RET_DATA**) are "data" to be associated with this return value.

If multiple filters exist, they are *all* executed, in reverse order of their addition to the filter tree—that is, the most recently installed filter is executed first. (Note that all filters will be called even if one of the earlier filters returns **SECCOMP_RET_KILL**. This is done to simplify the kernel code and to provide a tiny speed-up in the execution of sets of filters by avoiding a check for this uncommon case.) The return value for the evaluation of a given system call is the first-seen action value of highest precedence (along with its accompanying data) returned by execution of all of the filters.

In decreasing order of precedence, the action values that may be returned by a seccomp filter are:

SECCOMP_RET_KILL_PROCESS (since Linux 4.14)
: This value results in immediate termination of the process, with a core dump. The system call is not executed. By contrast with **SECCOMP_RET_KILL_THREAD** below, all threads in the thread group are terminated. (For a discussion of thread groups, see the description of the **CLONE_THREAD** flag in *clone*(2).)

  The process terminates *as though* killed by a **SIGSYS** signal. Even if a signal handler has been registered for **SIGSYS**, the handler will be ignored in this case and the process always terminates. To a parent process that is waiting on this process (using *waitpid*(2) or similar), the returned *wstatus* will indicate that its child was terminated as though by a **SIGSYS** signal.

SECCOMP_RET_KILL_THREAD (or **SECCOMP_RET_KILL**)
: This value results in immediate termination of the thread that made the system call. The system call is not executed. Other threads in the same thread group will continue to execute.

  The thread terminates *as though* killed by a **SIGSYS** signal. See **SECCOMP_RET_KILL_PROCESS** above.

  Before Linux 4.11, any process terminated in this way would not trigger a coredump (even though **SIGSYS** is documented in *signal*(7) as having a default action of termination with a core dump). Since Linux 4.11, a single-threaded process will dump core if terminated in this way.

  With the addition of **SECCOMP_RET_KILL_PROCESS** in Linux 4.14, **SECCOMP_RET_KILL_THREAD** was added as a synonym for **SECCOMP_RET_KILL**, in order to more clearly distinguish the two actions.

  Note: the use of **SECCOMP_RET_KILL_THREAD** to kill a single thread in a multi-threaded process is likely to leave the process in a permanently inconsistent and possibly corrupt state.

SECCOMP_RET_TRAP
: This value results in the kernel sending a thread-directed **SIGSYS** signal to the triggering thread. (The system call is not executed.) Various fields will be set in the *siginfo_t* structure (see *sigaction*(2)) associated with signal:

  - *si_signo* will contain **SIGSYS**.
  - *si_call_addr* will show the address of the system call instruction.
  - *si_syscall* and *si_arch* will indicate which system call was attempted.
  - *si_code* will contain **SYS_SECCOMP**.
  - *si_errno* will contain the **SECCOMP_RET_DATA** portion of the filter return value.

  The program counter will be as though the system call happened (i.e., the program counter will not point to the system call instruction). The return value register will contain an architecture−dependent value; if resuming execution, set it to something appropriate for the system

call. (The architecture dependency is because replacing it with **ENOSYS** could overwrite some useful information.)

SECCOMP_RET_ERRNO
This value results in the **SECCOMP_RET_DATA** portion of the filter's return value being passed to user space as the *errno* value without executing the system call.

SECCOMP_RET_USER_NOTIF (since Linux 5.0)
Forward the system call to an attached user-space supervisor process to allow that process to decide what to do with the system call. If there is no attached supervisor (either because the filter was not installed with the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag or because the file descriptor was closed), the filter returns **ENOSYS** (similar to what happens when a filter returns **SECCOMP_RET_TRACE** and there is no tracer). See *seccomp_unotify*(2) for further details.

Note that the supervisor process will not be notified if another filter returns an action value with a precedence greater than **SECCOMP_RET_USER_NOTIF**.

SECCOMP_RET_TRACE
When returned, this value will cause the kernel to attempt to notify a *ptrace*(2)-based tracer prior to executing the system call. If there is no tracer present, the system call is not executed and returns a failure status with *errno* set to **ENOSYS**.

A tracer will be notified if it requests **PTRACE_O_TRACESECCOMP** using *ptrace(PTRACE_SETOPTIONS)*. The tracer will be notified of a **PTRACE_EVENT_SECCOMP** and the **SECCOMP_RET_DATA** portion of the filter's return value will be available to the tracer via **PTRACE_GETEVENTMSG**.

The tracer can skip the system call by changing the system call number to −1. Alternatively, the tracer can change the system call requested by changing the system call to a valid system call number. If the tracer asks to skip the system call, then the system call will appear to return the value that the tracer puts in the return value register.

Before Linux 4.8, the seccomp check will not be run again after the tracer is notified. (This means that, on older kernels, seccomp-based sandboxes **must not** allow use of *ptrace*(2)—even of other sandboxed processes—without extreme care; ptracers can use this mechanism to escape from the seccomp sandbox.)

Note that a tracer process will not be notified if another filter returns an action value with a precedence greater than **SECCOMP_RET_TRACE**.

SECCOMP_RET_LOG (since Linux 4.14)
This value results in the system call being executed after the filter return action is logged. An administrator may override the logging of this action via the */proc/sys/kernel/seccomp/actions_logged* file.

SECCOMP_RET_ALLOW
This value results in the system call being executed.

If an action value other than one of the above is specified, then the filter action is treated as either **SECCOMP_RET_KILL_PROCESS** (since Linux 4.14) or **SECCOMP_RET_KILL_THREAD** (in Linux 4.13 and earlier).

/proc interfaces
The files in the directory */proc/sys/kernel/seccomp* provide additional seccomp information and configuration:

actions_avail (since Linux 4.14)
A read-only ordered list of seccomp filter return actions in string form. The ordering, from left-to-right, is in decreasing order of precedence. The list represents the set of seccomp filter return actions supported by the kernel.

actions_logged (since Linux 4.14)
A read-write ordered list of seccomp filter return actions that are allowed to be logged. Writes to the file do not need to be in ordered form but reads from the file will be ordered in the same way as the *actions_avail* file.

It is important to note that the value of *actions_logged* does not prevent certain filter return actions from being logged when the audit subsystem is configured to audit a task. If the action is not found in the *actions_logged* file, the final decision on whether to audit the action for that task is ultimately left up to the audit subsystem to decide for all filter return actions other than **SECCOMP_RET_ALLOW**.

The "allow" string is not accepted in the *actions_logged* file as it is not possible to log **SECCOMP_RET_ALLOW** actions. Attempting to write "allow" to the file will fail with the error **EINVAL**.

Audit logging of seccomp actions

Since Linux 4.14, the kernel provides the facility to log the actions returned by seccomp filters in the audit log. The kernel makes the decision to log an action based on the action type, whether or not the action is present in the *actions_logged* file, and whether kernel auditing is enabled (e.g., via the kernel boot option *audit=1*). The rules are as follows:

- If the action is **SECCOMP_RET_ALLOW**, the action is not logged.
- Otherwise, if the action is either **SECCOMP_RET_KILL_PROCESS** or **SECCOMP_RET_KILL_THREAD**, and that action appears in the *actions_logged* file, the action is logged.
- Otherwise, if the filter has requested logging (the **SECCOMP_FILTER_FLAG_LOG** flag) and the action appears in the *actions_logged* file, the action is logged.
- Otherwise, if kernel auditing is enabled and the process is being audited (**autrace**(8)), the action is logged.
- Otherwise, the action is not logged.

RETURN VALUE

On success, **seccomp**() returns 0. On error, if **SECCOMP_FILTER_FLAG_TSYNC** was used, the return value is the ID of the thread that caused the synchronization failure. (This ID is a kernel thread ID of the type returned by *clone*(2) and *gettid*(2).) On other errors, −1 is returned, and *errno* is set to indicate the error.

ERRORS

seccomp() can fail for the following reasons:

EACCES
The caller did not have the **CAP_SYS_ADMIN** capability in its user namespace, or had not set *no_new_privs* before using **SECCOMP_SET_MODE_FILTER**.

EBUSY
While installing a new filter, the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag was specified, but a previous filter had already been installed with that flag.

EFAULT
args was not a valid address.

EINVAL
operation is unknown or is not supported by this kernel version or configuration.

EINVAL
The specified *flags* are invalid for the given *operation*.

EINVAL
operation included **BPF_ABS**, but the specified offset was not aligned to a 32-bit boundary or exceeded *sizeof(struct seccomp_data)*.

EINVAL
A secure computing mode has already been set, and *operation* differs from the existing setting.

EINVAL
operation specified **SECCOMP_SET_MODE_FILTER**, but the filter program pointed to by *args* was not valid or the length of the filter program was zero or exceeded **BPF_MAXINSNS** (4096) instructions.

ENOMEM
Out of memory.

ENOMEM
The total length of all filter programs attached to the calling thread would exceed **MAX_INSNS_PER_PATH** (32768) instructions. Note that for the purposes of calculating this limit, each already existing filter program incurs an overhead penalty of 4 instructions.

EOPNOTSUPP
operation specified **SECCOMP_GET_ACTION_AVAIL**, but the kernel does not support the filter return action specified by *args*.

ESRCH
Another thread caused a failure during thread sync, but its ID could not be determined.

STANDARDS
Linux.

HISTORY
Linux 3.17.

NOTES
Rather than hand-coding seccomp filters as shown in the example below, you may prefer to employ the *libseccomp* library, which provides a front-end for generating seccomp filters.

The *Seccomp* field of the */proc/*pid*/status* file provides a method of viewing the seccomp mode of a process; see *proc*(5).

seccomp() provides a superset of the functionality provided by the *prctl*(2) **PR_SET_SECCOMP** operation (which does not support *flags*).

Since Linux 4.4, the *ptrace*(2) **PTRACE_SECCOMP_GET_FILTER** operation can be used to dump a process's seccomp filters.

Architecture support for seccomp BPF
Architecture support for seccomp BPF filtering is available on the following architectures:

- x86-64, i386, x32 (since Linux 3.5)
- ARM (since Linux 3.8)
- s390 (since Linux 3.8)
- MIPS (since Linux 3.16)
- ARM-64 (since Linux 3.19)
- PowerPC (since Linux 4.3)
- Tile (since Linux 4.3)
- PA-RISC (since Linux 4.6)

Caveats
There are various subtleties to consider when applying seccomp filters to a program, including the following:

- Some traditional system calls have user-space implementations in the *vdso*(7) on many architectures. Notable examples include *clock_gettime*(2), *gettimeofday*(2), and *time*(2). On such architectures, seccomp filtering for these system calls will have no effect. (However, there are cases where the *vdso*(7) implementations may fall back to invoking the true system call, in which case seccomp filters would see the system call.)
- Seccomp filtering is based on system call numbers. However, applications typically do not directly invoke system calls, but instead call wrapper functions in the C library which in turn invoke the system calls. Consequently, one must be aware of the following:
  - The glibc wrappers for some traditional system calls may actually employ system calls with different names in the kernel. For example, the *exit*(2) wrapper function actually employs the *exit_group*(2) system call, and the *fork*(2) wrapper function actually calls *clone*(2).
  - The behavior of wrapper functions may vary across architectures, according to the range of system calls provided on those architectures. In other words, the same wrapper function may invoke different system calls on different architectures.

- Finally, the behavior of wrapper functions can change across glibc versions. For example, in older versions, the glibc wrapper function for *open*(2) invoked the system call of the same name, but starting in glibc 2.26, the implementation switched to calling *openat*(2) on all architectures.

The consequence of the above points is that it may be necessary to filter for a system call other than might be expected. Various manual pages in Section 2 provide helpful details about the differences between wrapper functions and the underlying system calls in subsections entitled *C library/kernel differences*.

Furthermore, note that the application of seccomp filters even risks causing bugs in an application, when the filters cause unexpected failures for legitimate operations that the application might need to perform. Such bugs may not easily be discovered when testing the seccomp filters if the bugs occur in rarely used application code paths.

Seccomp-specific BPF details

Note the following BPF details specific to seccomp filters:

- The **BPF_H** and **BPF_B** size modifiers are not supported: all operations must load and store (4-byte) words (**BPF_W**).
- To access the contents of the *seccomp_data* buffer, use the **BPF_ABS** addressing mode modifier.
- The **BPF_LEN** addressing mode modifier yields an immediate mode operand whose value is the size of the *seccomp_data* buffer.

EXAMPLES

The program below accepts four or more arguments. The first three arguments are a system call number, a numeric architecture identifier, and an error number. The program uses these values to construct a BPF filter that is used at run time to perform the following checks:

- If the program is not running on the specified architecture, the BPF filter causes system calls to fail with the error **ENOSYS**.
- If the program attempts to execute the system call with the specified number, the BPF filter causes the system call to fail, with *errno* being set to the specified error number.

The remaining command-line arguments specify the pathname and additional arguments of a program that the example program should attempt to execute using *execv*(3) (a library function that employs the *execve*(2) system call). Some example runs of the program are shown below.

First, we display the architecture that we are running on (x86-64) and then construct a shell function that looks up system call numbers on this architecture:

```
$ uname -m
x86_64
$ syscall_nr() {
    cat /usr/src/linux/arch/x86/syscalls/syscall_64.tbl | \
    awk '$2 != "x32" && $3 == "'$1'" { print $1 }'
}
```

When the BPF filter rejects a system call (case [2] above), it causes the system call to fail with the error number specified on the command line. In the experiments shown here, we'll use error number 99:

```
$ errno 99
EADDRNOTAVAIL 99 Cannot assign requested address
```

In the following example, we attempt to run the command *whoami*(1), but the BPF filter rejects the *execve*(2) system call, so that the command is not even executed:

```
$ syscall_nr execve
59
$ ./a.out
Usage: ./a.out <syscall_nr> <arch> <errno> <prog> [<args>]
Hint for <arch>: AUDIT_ARCH_I386: 0x40000003
                 AUDIT_ARCH_X86_64: 0xC000003E
$ ./a.out 59 0xC000003E 99 /bin/whoami
execv: Cannot assign requested address
```

In the next example, the BPF filter rejects the *write*(2) system call, so that, although it is successfully started, the *whoami*(1) command is not able to write output:

```
$ syscall_nr write
1
$ ./a.out 1 0xC000003E 99 /bin/whoami
```

In the final example, the BPF filter rejects a system call that is not used by the *whoami*(1) command, so it is able to successfully execute and produce output:

```
$ syscall_nr preadv
295
$ ./a.out 295 0xC000003E 99 /bin/whoami
cecilia
```

Program source

```
#include <linux/audit.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/prctl.h>
#include <sys/syscall.h>
#include <unistd.h>

#define X32_SYSCALL_BIT 0x40000000
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

static int
install_filter(int syscall_nr, unsigned int t_arch, int f_errno)
{
    unsigned int upper_nr_limit = 0xffffffff;

    /* Assume that AUDIT_ARCH_X86_64 means the normal x86-64 ABI
       (in the x32 ABI, all system calls have bit 30 set in the
       'nr' field, meaning the numbers are >= X32_SYSCALL_BIT). */
    if (t_arch == AUDIT_ARCH_X86_64)
        upper_nr_limit = X32_SYSCALL_BIT - 1;

    struct sock_filter filter[] = {
        /* [0] Load architecture from 'seccomp_data' buffer into
               accumulator. */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
                 (offsetof(struct seccomp_data, arch))),

        /* [1] Jump forward 5 instructions if architecture does not
               match 't_arch'. */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, t_arch, 0, 5),

        /* [2] Load system call number from 'seccomp_data' buffer into
               accumulator. */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
                 (offsetof(struct seccomp_data, nr))),

        /* [3] Check ABI - only needed for x86-64 in deny-list use
               cases.  Use BPF_JGT instead of checking against the bit
               mask to avoid having to reload the syscall number. */
        BPF_JUMP(BPF_JMP | BPF_JGT | BPF_K, upper_nr_limit, 3, 0),

        /* [4] Jump forward 1 instruction if system call number
```

```
                does not match 'syscall_nr'. */
            BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, syscall_nr, 0, 1),

            /* [5] Matching architecture and system call: don't execute
               the system call, and return 'f_errno' in 'errno'. */
            BPF_STMT(BPF_RET | BPF_K,
                     SECCOMP_RET_ERRNO | (f_errno & SECCOMP_RET_DATA)),

            /* [6] Destination of system call number mismatch: allow other
                system calls. */
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

            /* [7] Destination of architecture mismatch: kill process. */
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
        };

        struct sock_fprog prog = {
            .len = ARRAY_SIZE(filter),
            .filter = filter,
        };

        if (syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, 0, &prog)) {
            perror("seccomp");
            return 1;
        }

        return 0;
    }

    int
    main(int argc, char *argv[])
    {
        if (argc < 5) {
            fprintf(stderr, "Usage: "
                    "%s <syscall_nr> <arch> <errno> <prog> [<args>]\n"
                    "Hint for <arch>: AUDIT_ARCH_I386: 0x%X\n"
                    "                 AUDIT_ARCH_X86_64: 0x%X\n"
                    "\n", argv[0], AUDIT_ARCH_I386, AUDIT_ARCH_X86_64);
            exit(EXIT_FAILURE);
        }

        if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
            perror("prctl");
            exit(EXIT_FAILURE);
        }

        if (install_filter(strtol(argv[1], NULL, 0),
                           strtoul(argv[2], NULL, 0),
                           strtol(argv[3], NULL, 0)))
            exit(EXIT_FAILURE);

        execv(argv[4], &argv[4]);
        perror("execv");
        exit(EXIT_FAILURE);
    }
```

SEE ALSO

bpfc(1), *strace*(1), *bpf*(2), *prctl*(2), *ptrace*(2), *seccomp_unotify*(2), *sigaction*(2), *proc*(5), *signal*(7), *socket*(7)

Various pages from the *libseccomp* library, including: *scmp_sys_resolver*(1), *seccomp_export_bpf*(3), *seccomp_init*(3), *seccomp_load*(3), and *seccomp_rule_add*(3)

The kernel source files *Documentation/networking/filter.txt* and *Documentation/userspace−api/seccomp_filter.rst* (or *Documentation/prctl/seccomp_filter.txt* before Linux 4.13).

McCanne, S. and Jacobson, V. (1992) *The BSD Packet Filter: A New Architecture for User-level Packet Capture*, Proceedings of the USENIX Winter 1993 Conference

NAME

seccomp_unotify − Seccomp user-space notification mechanism

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>

int seccomp(unsigned int operation, unsigned int  flags, void *args);

#include <sys/ioctl.h>

int ioctl(int  fd, SECCOMP_IOCTL_NOTIF_RECV,
        struct seccomp_notif *req);
int ioctl(int  fd, SECCOMP_IOCTL_NOTIF_SEND,
        struct seccomp_notif_resp *resp);
int ioctl(int  fd, SECCOMP_IOCTL_NOTIF_ID_VALID, __u64 *id);
int ioctl(int  fd, SECCOMP_IOCTL_NOTIF_ADDFD,
        struct seccomp_notif_addfd *addfd);
```

DESCRIPTION

This page describes the user-space notification mechanism provided by the Secure Computing (seccomp) facility. As well as the use of the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag, the **SECCOMP_RET_USER_NOTIF** action value, and the **SECCOMP_GET_NOTIF_SIZES** operation described in *seccomp*(2), this mechanism involves the use of a number of related *ioctl*(2) operations (described below).

Overview

In conventional usage of a seccomp filter, the decision about how to treat a system call is made by the filter itself. By contrast, the user-space notification mechanism allows the seccomp filter to delegate the handling of the system call to another user-space process. Note that this mechanism is explicitly **not** intended as a method implementing security policy; see NOTES.

In the discussion that follows, the thread(s) on which the seccomp filter is installed is (are) referred to as the *target*, and the process that is notified by the user-space notification mechanism is referred to as the *supervisor*.

A suitably privileged supervisor can use the user-space notification mechanism to perform actions on behalf of the target. The advantage of the user-space notification mechanism is that the supervisor will usually be able to retrieve information about the target and the performed system call that the seccomp filter itself cannot. (A seccomp filter is limited in the information it can obtain and the actions that it can perform because it is running on a virtual machine inside the kernel.)

An overview of the steps performed by the target and the supervisor is as follows:

(1) The target establishes a seccomp filter in the usual manner, but with two differences:

- The *seccomp*(2) *flags* argument includes the flag **SECCOMP_FILTER_FLAG_NEW_LISTENER**. Consequently, the return value of the (successful) *seccomp*(2) call is a new "listening" file descriptor that can be used to receive notifications. Only one "listening" seccomp filter can be installed for a thread.
- In cases where it is appropriate, the seccomp filter returns the action value **SECCOMP_RET_USER_NOTIF**. This return value will trigger a notification event.

(2) In order that the supervisor can obtain notifications using the listening file descriptor, (a duplicate of) that file descriptor must be passed from the target to the supervisor. One way in which this could be done is by passing the file descriptor over a UNIX domain socket connection between the target and the supervisor (using the **SCM_RIGHTS** ancillary message type described in *unix*(7)). Another way to do this is through the use of *pidfd_getfd*(2).

(3) The supervisor will receive notification events on the listening file descriptor. These events are returned as structures of type *seccomp_notif*. Because this structure and its size may evolve over kernel versions, the supervisor must first determine the size of this structure using the *seccomp*(2) **SECCOMP_GET_NOTIF_SIZES** operation, which returns a structure of type

seccomp_notif_sizes. The supervisor allocates a buffer of size *seccomp_notif_sizes.seccomp_notif* bytes to receive notification events. In addition,the supervisor allocates another buffer of size *seccomp_notif_sizes.seccomp_notif_resp* bytes for the response (a *struct seccomp_notif_resp* structure) that it will provide to the kernel (and thus the target).

(4) The target then performs its workload, which includes system calls that will be controlled by the seccomp filter. Whenever one of these system calls causes the filter to return the **SECCOMP_RET_USER_NOTIF** action value, the kernel does *not* (yet) execute the system call; instead, execution of the target is temporarily blocked inside the kernel (in a sleep state that is interruptible by signals) and a notification event is generated on the listening file descriptor.

(5) The supervisor can now repeatedly monitor the listening file descriptor for **SECCOMP_RET_USER_NOTIF**-triggered events. To do this, the supervisor uses the **SECCOMP_IOCTL_NOTIF_RECV** *ioctl*(2) operation to read information about a notification event; this operation blocks until an event is available. The operation returns a *seccomp_notif* structure containing information about the system call that is being attempted by the target. (As described in NOTES, the file descriptor can also be monitored with *select*(2), *poll*(2), or *epoll*(7).)

(6) The *seccomp_notif* structure returned by the **SECCOMP_IOCTL_NOTIF_RECV** operation includes the same information (a *seccomp_data* structure) that was passed to the seccomp filter. This information allows the supervisor to discover the system call number and the arguments for the target's system call. In addition, the notification event contains the ID of the thread that triggered the notification and a unique cookie value that is used in subsequent **SECCOMP_IOCTL_NOTIF_ID_VALID** and **SECCOMP_IOCTL_NOTIF_SEND** operations.

The information in the notification can be used to discover the values of pointer arguments for the target's system call. (This is something that can't be done from within a seccomp filter.) One way in which the supervisor can do this is to open the corresponding */proc/*tid*/mem* file (see *proc*(5)) and read bytes from the location that corresponds to one of the pointer arguments whose value is supplied in the notification event. (The supervisor must be careful to avoid a race condition that can occur when doing this; see the description of the **SECCOMP_IOCTL_NOTIF_ID_VALID** *ioctl*(2) operation below.) In addition, the supervisor can access other system information that is visible in user space but which is not accessible from a seccomp filter.

(7) Having obtained information as per the previous step, the supervisor may then choose to perform an action in response to the target's system call (which, as noted above, is not executed when the seccomp filter returns the **SECCOMP_RET_USER_NOTIF** action value).

One example use case here relates to containers. The target may be located inside a container where it does not have sufficient capabilities to mount a filesystem in the container's mount namespace. However, the supervisor may be a more privileged process that does have sufficient capabilities to perform the mount operation.

(8) The supervisor then sends a response to the notification. The information in this response is used by the kernel to construct a return value for the target's system call and provide a value that will be assigned to the *errno* variable of the target.

The response is sent using the **SECCOMP_IOCTL_NOTIF_SEND** *ioctl*(2) operation, which is used to transmit a *seccomp_notif_resp* structure to the kernel. This structure includes a cookie value that the supervisor obtained in the *seccomp_notif* structure returned by the **SECCOMP_IOCTL_NOTIF_RECV** operation. This cookie value allows the kernel to associate the response with the target. This structure must include the cookie value that the supervisor obtained in the *seccomp_notif* structure returned by the **SECCOMP_IOCTL_NOTIF_RECV** operation; the cookie allows the kernel to associate the response with the target.

(9) Once the notification has been sent, the system call in the target thread unblocks, returning the information that was provided by the supervisor in the notification response.

As a variation on the last two steps, the supervisor can send a response that tells the kernel that it should execute the target thread's system call; see the discussion of **SECCOMP_USER_NOTIF_FLAG_CONTINUE**, below.

IOCTL OPERATIONS

The following *ioctl*(2) operations are supported by the seccomp user-space notification file descriptor. For each of these operations, the first (file descriptor) argument of *ioctl*(2) is the listening file descriptor returned by a call to *seccomp*(2) with the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag.

SECCOMP_IOCTL_NOTIF_RECV

The **SECCOMP_IOCTL_NOTIF_RECV** operation (available since Linux 5.0) is used to obtain a user-space notification event. If no such event is currently pending, the operation blocks until an event occurs. The third *ioctl*(2) argument is a pointer to a structure of the following form which contains information about the event. This structure must be zeroed out before the call.

```
struct seccomp_notif {
    __u64  id;              /* Cookie */
    __u32  pid;             /* TID of target thread */
    __u32  flags;           /* Currently unused (0) */
    struct seccomp_data data;   /* See seccomp(2) */
};
```

The fields in this structure are as follows:

id This is a cookie for the notification. Each such cookie is guaranteed to be unique for the corresponding seccomp filter.

- The cookie can be used with the **SECCOMP_IOCTL_NOTIF_ID_VALID** *ioctl*(2) operation described below.
- When returning a notification response to the kernel, the supervisor must include the cookie value in the *seccomp_notif_resp* structure that is specified as the argument of the **SECCOMP_IOCTL_NOTIF_SEND** operation.

pid This is the thread ID of the target thread that triggered the notification event.

flags This is a bit mask of flags providing further information on the event. In the current implementation, this field is always zero.

data This is a *seccomp_data* structure containing information about the system call that triggered the notification. This is the same structure that is passed to the seccomp filter. See *seccomp*(2) for details of this structure.

On success, this operation returns 0; on failure, −1 is returned, and *errno* is set to indicate the cause of the error. This operation can fail with the following errors:

EINVAL (since Linux 5.5)
The *seccomp_notif* structure that was passed to the call contained nonzero fields.

ENOENT
The target thread was killed by a signal as the notification information was being generated, or the target's (blocked) system call was interrupted by a signal handler.

SECCOMP_IOCTL_NOTIF_ID_VALID

The **SECCOMP_IOCTL_NOTIF_ID_VALID** operation (available since Linux 5.0) is used to check that a notification ID returned by an earlier **SECCOMP_IOCTL_NOTIF_RECV** operation is still valid (i.e., that the target still exists and its system call is still blocked waiting for a response).

The third *ioctl*(2) argument is a pointer to the cookie (*id*) returned by the **SECCOMP_IOCTL_NOTIF_RECV** operation.

This operation is necessary to avoid race conditions that can occur when the *pid* returned by the **SECCOMP_IOCTL_NOTIF_RECV** operation terminates, and that process ID is reused by another process. An example of this kind of race is the following

(1) A notification is generated on the listening file descriptor. The returned *seccomp_notif* contains the TID of the target thread (in the *pid* field of the structure).

(2) The target terminates.

(3) Another thread or process is created on the system that by chance reuses the TID that was freed when the target terminated.

(4) The supervisor *open*(2)s the */proc/*tid*/mem* file for the TID obtained in step 1, with the intention of (say) inspecting the memory location(s) that containing the argument(s) of the system call that triggered the notification in step 1.

In the above scenario, the risk is that the supervisor may try to access the memory of a process other than the target. This race can be avoided by following the call to *open*(2) with a **SECCOMP_IOCTL_NOTIF_ID_VALID** operation to verify that the process that generated the notification is still alive. (Note that if the target terminates after the latter step, a subsequent *read*(2) from the file descriptor may return 0, indicating end of file.)

See NOTES for a discussion of other cases where **SECCOMP_IOCTL_NOTIF_ID_VALID** checks must be performed.

On success (i.e., the notification ID is still valid), this operation returns 0. On failure (i.e., the notification ID is no longer valid), −1 is returned, and *errno* is set to **ENOENT**.

SECCOMP_IOCTL_NOTIF_SEND

The **SECCOMP_IOCTL_NOTIF_SEND** operation (available since Linux 5.0) is used to send a notification response back to the kernel. The third *ioctl*(2) argument of this structure is a pointer to a structure of the following form:

```
struct seccomp_notif_resp {
    __u64 id;           /* Cookie value */
    __s64 val;          /* Success return value */
    __s32 error;        /* 0 (success) or negative error number */
    __u32 flags;        /* See below */
};
```

The fields of this structure are as follows:

id This is the cookie value that was obtained using the **SECCOMP_IOCTL_NOTIF_RECV** operation. This cookie value allows the kernel to correctly associate this response with the system call that triggered the user-space notification.

val This is the value that will be used for a spoofed success return for the target's system call; see below.

error This is the value that will be used as the error number (*errno*) for a spoofed error return for the target's system call; see below.

flags This is a bit mask that includes zero or more of the following flags:

SECCOMP_USER_NOTIF_FLAG_CONTINUE (since Linux 5.5)
Tell the kernel to execute the target's system call.

Two kinds of response are possible:

- A response to the kernel telling it to execute the target's system call. In this case, the *flags* field includes **SECCOMP_USER_NOTIF_FLAG_CONTINUE** and the *error* and *val* fields must be zero.

  This kind of response can be useful in cases where the supervisor needs to do deeper analysis of the target's system call than is possible from a seccomp filter (e.g., examining the values of pointer arguments), and, having decided that the system call does not require emulation by the supervisor, the supervisor wants the system call to be executed normally in the target.

  The **SECCOMP_USER_NOTIF_FLAG_CONTINUE** flag should be used with caution; see NOTES.

- A spoofed return value for the target's system call. In this case, the kernel does not execute the target's system call, instead causing the system call to return a spoofed value as specified by fields of the *seccomp_notif_resp* structure. The supervisor should set the fields of this structure as follows:

  - *flags* does not contain **SECCOMP_USER_NOTIF_FLAG_CONTINUE**.
  - *error* is set either to 0 for a spoofed "success" return or to a negative error number for a spoofed "failure" return. In the former case, the kernel causes the target's system call to return the value specified in the *val* field. In the latter case, the kernel causes the target's system call to return −1, and *errno* is assigned the negated *error* value.

- *val* is set to a value that will be used as the return value for a spoofed "success" return for the target's system call. The value in this field is ignored if the *error* field contains a nonzero value.

On success, this operation returns 0; on failure, −1 is returned, and *errno* is set to indicate the cause of the error. This operation can fail with the following errors:

EINPROGRESS
A response to this notification has already been sent.

EINVAL
An invalid value was specified in the *flags field.*

EINVAL
The *flags* field contained **SECCOMP_USER_NOTIF_FLAG_CONTINUE**, and the *error* or *val* field was not zero.

ENOENT
The blocked system call in the target has been interrupted by a signal handler or the target has terminated.

SECCOMP_IOCTL_NOTIF_ADDFD
The **SECCOMP_IOCTL_NOTIF_ADDFD** operation (available since Linux 5.9) allows the supervisor to install a file descriptor into the target's file descriptor table. Much like the use of **SCM_RIGHTS** messages described in *unix*(7), this operation is semantically equivalent to duplicating a file descriptor from the supervisor's file descriptor table into the target's file descriptor table.

The **SECCOMP_IOCTL_NOTIF_ADDFD** operation permits the supervisor to emulate a target system call (such as *socket*(2) or *openat*(2)) that generates a file descriptor. The supervisor can perform the system call that generates the file descriptor (and associated open file description) and then use this operation to allocate a file descriptor that refers to the same open file description in the target. (For an explanation of open file descriptions, see *open*(2).)

Once this operation has been performed, the supervisor can close its copy of the file descriptor.

In the target, the received file descriptor is subject to the same Linux Security Module (LSM) checks as are applied to a file descriptor that is received in an **SCM_RIGHTS** ancillary message. If the file descriptor refers to a socket, it inherits the cgroup version 1 network controller settings (*classid* and *netprioidx*) of the target.

The third *ioctl*(2) argument is a pointer to a structure of the following form:

```
struct seccomp_notif_addfd {
    __u64 id;           /* Cookie value */
    __u32 flags;        /* Flags */
    __u32 srcfd;        /* Local file descriptor number */
    __u32 newfd;        /* 0 or desired file descriptor
                           number in target */
    __u32 newfd_flags;  /* Flags to set on target file
                           descriptor */
};
```

The fields in this structure are as follows:

id This field should be set to the notification ID (cookie value) that was obtained via **SECCOMP_IOCTL_NOTIF_RECV**.

flags This field is a bit mask of flags that modify the behavior of the operation. Currently, only one flag is supported:

SECCOMP_ADDFD_FLAG_SETFD
When allocating the file descriptor in the target, use the file descriptor number specified in the *newfd* field.

SECCOMP_ADDFD_FLAG_SEND (since Linux 5.14)
Perform the equivalent of **SECCOMP_IOCTL_NOTIF_ADDFD** plus **SECCOMP_IOCTL_NOTIF_SEND** as an atomic operation. On successful invocation, the target process's *errno* will be 0 and the return value will be the file descriptor number that was allocated in the target. If allocating the file descriptor in the target

fails, the target's system call continues to be blocked until a successful response is sent.

srcfd This field should be set to the number of the file descriptor in the supervisor that is to be duplicated.

newfd This field determines which file descriptor number is allocated in the target. If the **SECCOMP_ADDFD_FLAG_SETFD** flag is set, then this field specifies which file descriptor number should be allocated. If this file descriptor number is already open in the target, it is atomically closed and reused. If the descriptor duplication fails due to an LSM check, or if *srcfd* is not a valid file descriptor, the file descriptor *newfd* will not be closed in the target process.

If the **SECCOMP_ADDFD_FLAG_SETFD** flag it not set, then this field must be 0, and the kernel allocates the lowest unused file descriptor number in the target.

newfd_flags
This field is a bit mask specifying flags that should be set on the file descriptor that is received in the target process. Currently, only the following flag is implemented:

O_CLOEXEC
Set the close-on-exec flag on the received file descriptor.

On success, this *ioctl*(2) call returns the number of the file descriptor that was allocated in the target. Assuming that the emulated system call is one that returns a file descriptor as its function result (e.g., *socket*(2)), this value can be used as the return value (*resp.val*) that is supplied in the response that is subsequently sent with the **SECCOMP_IOCTL_NOTIF_SEND** operation.

On error, −1 is returned and *errno* is set to indicate the cause of the error.

This operation can fail with the following errors:

EBADF
Allocating the file descriptor in the target would cause the target's **RLIMIT_NOFILE** limit to be exceeded (see *getrlimit*(2)).

EBUSY
If the flag **SECCOMP_IOCTL_NOTIF_SEND** is used, this means the operation can't proceed until other **SECCOMP_IOCTL_NOTIF_ADDFD** requests are processed.

EINPROGRESS
The user-space notification specified in the *id* field exists but has not yet been fetched (by a **SECCOMP_IOCTL_NOTIF_RECV**) or has already been responded to (by a **SECCOMP_IOCTL_NOTIF_SEND**).

EINVAL
An invalid flag was specified in the *flags* or *newfd_flags* field, or the *newfd* field is nonzero and the **SECCOMP_ADDFD_FLAG_SETFD** flag was not specified in the *flags* field.

EMFILE
The file descriptor number specified in *newfd* exceeds the limit specified in */proc/sys/fs/nr_open*.

ENOENT
The blocked system call in the target has been interrupted by a signal handler or the target has terminated.

Here is some sample code (with error handling omitted) that uses the **SECCOMP_ADDFD_FLAG_SETFD** operation (here, to emulate a call to *openat*(2)):

```
int fd, removeFd;

fd = openat(req->data.args[0], path, req->data.args[2],
            req->data.args[3]);

struct seccomp_notif_addfd addfd;
addfd.id = req->id; /* Cookie from SECCOMP_IOCTL_NOTIF_RECV */
addfd.srcfd = fd;
```

```
        addfd.newfd = 0;
        addfd.flags = 0;
        addfd.newfd_flags = O_CLOEXEC;

        targetFd = ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_ADDFD, &addfd);

        close(fd);          /* No longer needed in supervisor */

        struct seccomp_notif_resp *resp;
            /* Code to allocate 'resp' omitted */
        resp->id = req->id;
        resp->error = 0;        /* "Success" */
        resp->val = targetFd;
        resp->flags = 0;
        ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_SEND, resp);
```

NOTES

One example use case for the user-space notification mechanism is to allow a container manager (a process which is typically running with more privilege than the processes inside the container) to mount block devices or create device nodes for the container. The mount use case provides an example of where the **SECCOMP_USER_NOTIF_FLAG_CONTINUE** *ioctl*(2) operation is useful. Upon receiving a notification for the *mount*(2) system call, the container manager (the "supervisor") can distinguish a request to mount a block filesystem (which would not be possible for a "target" process inside the container) and mount that file system. If, on the other hand, the container manager detects that the operation could be performed by the process inside the container (e.g., a mount of a *tmpfs*(5) filesystem), it can notify the kernel that the target process's *mount*(2) system call can continue.

select()/poll()/epoll semantics

The file descriptor returned when *seccomp*(2) is employed with the **SECCOMP_FILTER_FLAG_NEW_LISTENER** flag can be monitored using *poll*(2), *epoll*(7), and *select*(2). These interfaces indicate that the file descriptor is ready as follows:

- When a notification is pending, these interfaces indicate that the file descriptor is readable. Following such an indication, a subsequent **SECCOMP_IOCTL_NOTIF_RECV** *ioctl*(2) will not block, returning either information about a notification or else failing with the error **EINTR** if the target has been killed by a signal or its system call has been interrupted by a signal handler.
- After the notification has been received (i.e., by the **SECCOMP_IOCTL_NOTIF_RECV** *ioctl*(2) operation), these interfaces indicate that the file descriptor is writable, meaning that a notification response can be sent using the **SECCOMP_IOCTL_NOTIF_SEND** *ioctl*(2) operation.
- After the last thread using the filter has terminated and been reaped using *waitpid*(2) (or similar), the file descriptor indicates an end-of-file condition (readable in *select*(2); **POLLHUP**/**EPOLLHUP** in *poll*(2)/ *epoll_wait*(2)).

Design goals; use of SECCOMP_USER_NOTIF_FLAG_CONTINUE

The intent of the user-space notification feature is to allow system calls to be performed on behalf of the target. The target's system call should either be handled by the supervisor or allowed to continue normally in the kernel (where standard security policies will be applied).

Note well: this mechanism must not be used to make security policy decisions about the system call, which would be inherently race-prone for reasons described next.

The **SECCOMP_USER_NOTIF_FLAG_CONTINUE** flag must be used with caution. If set by the supervisor, the target's system call will continue. However, there is a time-of-check, time-of-use race here, since an attacker could exploit the interval of time where the target is blocked waiting on the "continue" response to do things such as rewriting the system call arguments.

Note furthermore that a user-space notifier can be bypassed if the existing filters allow the use of *seccomp*(2) or *prctl*(2) to install a filter that returns an action value with a higher precedence than **SECCOMP_RET_USER_NOTIF** (see *seccomp*(2)).

It should thus be absolutely clear that the seccomp user-space notification mechanism **can not** be used to implement a security policy! It should only ever be used in scenarios where a more privileged process supervises the system calls of a lesser privileged target to get around kernel-enforced security

restrictions when the supervisor deems this safe. In other words, in order to continue a system call, the supervisor should be sure that another security mechanism or the kernel itself will sufficiently block the system call if its arguments are rewritten to something unsafe.

Caveats regarding the use of */proc/*tid*/mem*

The discussion above noted the need to use the **SECCOMP_IOCTL_NOTIF_ID_VALID** *ioctl*(2) when opening the */proc/*tid*/mem* file of the target to avoid the possibility of accessing the memory of the wrong process in the event that the target terminates and its ID is recycled by another (unrelated) thread. However, the use of this *ioctl*(2) operation is also necessary in other situations, as explained in the following paragraphs.

Consider the following scenario, where the supervisor tries to read the pathname argument of a target's blocked *mount*(2) system call:

(1) From one of its functions (*func()*), the target calls *mount*(2), which triggers a user-space notification and causes the target to block.

(2) The supervisor receives the notification, opens */proc/*tid*/mem*, and (successfully) performs the **SECCOMP_IOCTL_NOTIF_ID_VALID** check.

(3) The target receives a signal, which causes the *mount*(2) to abort.

(4) The signal handler executes in the target, and returns.

(5) Upon return from the handler, the execution of *func()* resumes, and it returns (and perhaps other functions are called, overwriting the memory that had been used for the stack frame of *func()*).

(6) Using the address provided in the notification information, the supervisor reads from the target's memory location that used to contain the pathname.

(7) The supervisor now calls *mount*(2) with some arbitrary bytes obtained in the previous step.

The conclusion from the above scenario is this: since the target's blocked system call may be interrupted by a signal handler, the supervisor must be written to expect that the target may abandon its system call at **any** time; in such an event, any information that the supervisor obtained from the target's memory must be considered invalid.

To prevent such scenarios, every read from the target's memory must be separated from use of the bytes so obtained by a **SECCOMP_IOCTL_NOTIF_ID_VALID** check. In the above example, the check would be placed between the two final steps. An example of such a check is shown in EXAMPLES.

Following on from the above, it should be clear that a write by the supervisor into the target's memory can **never** be considered safe.

Caveats regarding blocking system calls

Suppose that the target performs a blocking system call (e.g., *accept*(2)) that the supervisor should handle. The supervisor might then in turn execute the same blocking system call.

In this scenario, it is important to note that if the target's system call is now interrupted by a signal, the supervisor is *not* informed of this. If the supervisor does not take suitable steps to actively discover that the target's system call has been canceled, various difficulties can occur. Taking the example of *accept*(2), the supervisor might remain blocked in its *accept*(2) holding a port number that the target (which, after the interruption by the signal handler, perhaps closed its listening socket) might expect to be able to reuse in a *bind*(2) call.

Therefore, when the supervisor wishes to emulate a blocking system call, it must do so in such a way that it gets informed if the target's system call is interrupted by a signal handler. For example, if the supervisor itself executes the same blocking system call, then it could employ a separate thread that uses the **SECCOMP_IOCTL_NOTIF_ID_VALID** operation to check if the target is still blocked in its system call. Alternatively, in the *accept*(2) example, the supervisor might use *poll*(2) to monitor both the notification file descriptor (so as to discover when the target's *accept*(2) call has been interrupted) and the listening file descriptor (so as to know when a connection is available).

If the target's system call is interrupted, the supervisor must take care to release resources (e.g., file descriptors) that it acquired on behalf of the target.

Interaction with SA_RESTART signal handlers

Consider the following scenario:

(1) The target process has used *sigaction*(2) to install a signal handler with the **SA_RESTART** flag.

(2) The target has made a system call that triggered a seccomp user-space notification and the target is currently blocked until the supervisor sends a notification response.

(3) A signal is delivered to the target and the signal handler is executed.

(4) When (if) the supervisor attempts to send a notification response, the **SECCOMP_IOCTL_NOTIF_SEND** *ioctl*(2)) operation will fail with the **ENOENT** error.

In this scenario, the kernel will restart the target's system call. Consequently, the supervisor will receive another user-space notification. Thus, depending on how many times the blocked system call is interrupted by a signal handler, the supervisor may receive multiple notifications for the same instance of a system call in the target.

One oddity is that system call restarting as described in this scenario will occur even for the blocking system calls listed in *signal*(7) that would **never** normally be restarted by the **SA_RESTART** flag.

Furthermore, if the supervisor response is a file descriptor added with **SECCOMP_IOCTL_NOTIF_ADDFD**, then the flag **SECCOMP_ADDFD_FLAG_SEND** can be used to atomically add the file descriptor and return that value, making sure no file descriptors are inadvertently leaked into the target.

BUGS

If a **SECCOMP_IOCTL_NOTIF_RECV** *ioctl*(2) operation is performed after the target terminates, then the *ioctl*(2) call simply blocks (rather than returning an error to indicate that the target no longer exists).

EXAMPLES

The (somewhat contrived) program shown below demonstrates the use of the interfaces described in this page. The program creates a child process that serves as the "target" process. The child process installs a seccomp filter that returns the **SECCOMP_RET_USER_NOTIF** action value if a call is made to *mkdir*(2). The child process then calls *mkdir*(2) once for each of the supplied command-line arguments, and reports the result returned by the call. After processing all arguments, the child process terminates.

The parent process acts as the supervisor, listening for the notifications that are generated when the target process calls *mkdir*(2). When such a notification occurs, the supervisor examines the memory of the target process (using */proc/*pid*/mem*) to discover the pathname argument that was supplied to the *mkdir*(2) call, and performs one of the following actions:

- If the pathname begins with the prefix "/tmp/", then the supervisor attempts to create the specified directory, and then spoofs a return for the target process based on the return value of the supervisor's *mkdir*(2) call. In the event that that call succeeds, the spoofed success return value is the length of the pathname.
- If the pathname begins with "./" (i.e., it is a relative pathname), the supervisor sends a **SECCOMP_USER_NOTIF_FLAG_CONTINUE** response to the kernel to say that the kernel should execute the target process's *mkdir*(2) call.
- If the pathname begins with some other prefix, the supervisor spoofs an error return for the target process, so that the target process's *mkdir*(2) call appears to fail with the error **EOPNOTSUPP** ("Operation not supported"). Additionally, if the specified pathname is exactly "/bye", then the supervisor terminates.

This program can be used to demonstrate various aspects of the behavior of the seccomp user-space notification mechanism. To help aid such demonstrations, the program logs various messages to show the operation of the target process (lines prefixed "T:") and the supervisor (indented lines prefixed "S:").

In the following example, the target attempts to create the directory */tmp/x*. Upon receiving the notification, the supervisor creates the directory on the target's behalf, and spoofs a success return to be received by the target process's *mkdir*(2) call.

```
$ ./seccomp_unotify /tmp/x
T: PID = 23168

T: about to mkdir("/tmp/x")
        S: got notification (ID 0x17445c4a0f4e0e3c) for PID 23168
```

```
            S: executing: mkdir("/tmp/x", 0700)
            S: success! spoofed return = 6
            S: sending response (flags = 0; val = 6; error = 0)
    T: SUCCESS: mkdir(2) returned 6

    T: terminating
            S: target has terminated; bye
```

In the above output, note that the spoofed return value seen by the target process is 6 (the length of the pathname */tmp/x*), whereas a normal *mkdir*(2) call returns 0 on success.

In the next example, the target attempts to create a directory using the relative pathname *./sub*. Since this pathname starts with "./", the supervisor sends a **SECCOMP_USER_NOTIF_FLAG_CONTINUE** response to the kernel, and the kernel then (successfully) executes the target process's *mkdir*(2) call.

```
    $ ./seccomp_unotify ./sub
    T: PID = 23204

    T: about to mkdir("./sub")
            S: got notification (ID 0xddb16abe25b4c12) for PID 23204
            S: target can execute system call
            S: sending response (flags = 0x1; val = 0; error = 0)
    T: SUCCESS: mkdir(2) returned 0

    T: terminating
            S: target has terminated; bye
```

If the target process attempts to create a directory with a pathname that doesn't start with "." and doesn't begin with the prefix "/tmp/", then the supervisor spoofs an error return (**EOPNOTSUPP**, "Operation not supported") for the target's *mkdir*(2) call (which is not executed):

```
    $ ./seccomp_unotify /xxx
    T: PID = 23178

    T: about to mkdir("/xxx")
            S: got notification (ID 0xe7dc095d1c524e80) for PID 23178
            S: spoofing error response (Operation not supported)
            S: sending response (flags = 0; val = 0; error = -95)
    T: ERROR: mkdir(2): Operation not supported

    T: terminating
            S: target has terminated; bye
```

In the next example, the target process attempts to create a directory with the pathname **/tmp/nosuchdir/b**. Upon receiving the notification, the supervisor attempts to create that directory, but the *mkdir*(2) call fails because the directory **/tmp/nosuchdir** does not exist. Consequently, the supervisor spoofs an error return that passes the error that it received back to the target process's *mkdir*(2) call.

```
    $ ./seccomp_unotify /tmp/nosuchdir/b
    T: PID = 23199

    T: about to mkdir("/tmp/nosuchdir/b")
            S: got notification (ID 0x8744454293506046) for PID 23199
            S: executing: mkdir("/tmp/nosuchdir/b", 0700)
            S: failure! (errno = 2; No such file or directory)
            S: sending response (flags = 0; val = 0; error = -2)
    T: ERROR: mkdir(2): No such file or directory

    T: terminating
            S: target has terminated; bye
```

If the supervisor receives a notification and sees that the argument of the target's *mkdir*(2) is the string

"/bye", then (as well as spoofing an **EOPNOTSUPP** error), the supervisor terminates. If the target process subsequently executes another *mkdir*(2) that triggers its seccomp filter to return the **SECCOMP_RET_USER_NOTIF** action value, then the kernel causes the target process's system call to fail with the error **ENOSYS** ("Function not implemented"). This is demonstrated by the following example:

```
$ ./seccomp_unotify /bye /tmp/y
T: PID = 23185

T: about to mkdir("/bye")
        S: got notification (ID 0xa81236b1d2f7b0f4) for PID 23185
        S: spoofing error response (Operation not supported)
        S: sending response (flags = 0; val = 0; error = -95)
        S: terminating **********
T: ERROR: mkdir(2): Operation not supported

T: about to mkdir("/tmp/y")
T: ERROR: mkdir(2): Function not implemented

T: terminating
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <linux/audit.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <signal.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/prctl.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>

#define ARRAY_SIZE(arr)  (sizeof(arr) / sizeof((arr)[0]))

/* Send the file descriptor 'fd' over the connected UNIX domain socket
   'sockfd'. Returns 0 on success, or -1 on error. */

static int
sendfd(int sockfd, int fd)
{
    int             data;
    struct iovec    iov;
    struct msghdr   msgh;
    struct cmsghdr  *cmsgp;

    /* Allocate a char array of suitable size to hold the ancillary data.
```

```
           However, since this buffer is in reality a 'struct cmsghdr', use a
           union to ensure that it is suitably aligned. */
        union {
            char   buf[CMSG_SPACE(sizeof(int))];
                        /* Space large enough to hold an 'int' */
            struct cmsghdr align;
        } controlMsg;

        /* The 'msg_name' field can be used to specify the address of the
           destination socket when sending a datagram. However, we do not
           need to use this field because 'sockfd' is a connected socket. */

        msgh.msg_name = NULL;
        msgh.msg_namelen = 0;

        /* On Linux, we must transmit at least one byte of real data in
           order to send ancillary data. We transmit an arbitrary integer
           whose value is ignored by recvfd(). */

        msgh.msg_iov = &iov;
        msgh.msg_iovlen = 1;
        iov.iov_base = &data;
        iov.iov_len = sizeof(int);
        data = 12345;

        /* Set 'msghdr' fields that describe ancillary data */

        msgh.msg_control = controlMsg.buf;
        msgh.msg_controllen = sizeof(controlMsg.buf);

        /* Set up ancillary data describing file descriptor to send */

        cmsgp = CMSG_FIRSTHDR(&msgh);
        cmsgp->cmsg_level = SOL_SOCKET;
        cmsgp->cmsg_type = SCM_RIGHTS;
        cmsgp->cmsg_len = CMSG_LEN(sizeof(int));
        memcpy(CMSG_DATA(cmsgp), &fd, sizeof(int));

        /* Send real plus ancillary data */

        if (sendmsg(sockfd, &msgh, 0) == -1)
            return -1;

        return 0;
    }

    /* Receive a file descriptor on a connected UNIX domain socket. Returns
       the received file descriptor on success, or -1 on error. */

    static int
    recvfd(int sockfd)
    {
        int            data, fd;
        ssize_t        nr;
        struct iovec   iov;
        struct msghdr  msgh;

        /* Allocate a char buffer for the ancillary data. See the comments
           in sendfd() */
```

```
    union {
        char   buf[CMSG_SPACE(sizeof(int))];
        struct cmsghdr align;
    } controlMsg;
    struct cmsghdr *cmsgp;

    /* The 'msg_name' field can be used to obtain the address of the
       sending socket. However, we do not need this information. */

    msgh.msg_name = NULL;
    msgh.msg_namelen = 0;

    /* Specify buffer for receiving real data */

    msgh.msg_iov = &iov;
    msgh.msg_iovlen = 1;
    iov.iov_base = &data;       /* Real data is an 'int' */
    iov.iov_len = sizeof(int);

    /* Set 'msghdr' fields that describe ancillary data */

    msgh.msg_control = controlMsg.buf;
    msgh.msg_controllen = sizeof(controlMsg.buf);

    /* Receive real plus ancillary data; real data is ignored */

    nr = recvmsg(sockfd, &msgh, 0);
    if (nr == -1)
        return -1;

    cmsgp = CMSG_FIRSTHDR(&msgh);

    /* Check the validity of the 'cmsghdr' */

    if (cmsgp == NULL
        || cmsgp->cmsg_len != CMSG_LEN(sizeof(int))
        || cmsgp->cmsg_level != SOL_SOCKET
        || cmsgp->cmsg_type != SCM_RIGHTS)
    {
        errno = EINVAL;
        return -1;
    }

    /* Return the received file descriptor to our caller */

    memcpy(&fd, CMSG_DATA(cmsgp), sizeof(int));
    return fd;
}

static void
sigchldHandler(int sig)
{
    char msg[] = "\tS: target has terminated; bye\n";

    write(STDOUT_FILENO, msg, sizeof(msg) - 1);
    _exit(EXIT_SUCCESS);
}

static int
```

```
seccomp(unsigned int operation, unsigned int flags, void *args)
{
    return syscall(SYS_seccomp, operation, flags, args);
}

/* The following is the x86-64-specific BPF boilerplate code for checking
   that the BPF program is running on the right architecture + ABI. At
   completion of these instructions, the accumulator contains the system
   call number. */

/* For the x32 ABI, all system call numbers have bit 30 set */

#define X32_SYSCALL_BIT         0x40000000

#define X86_64_CHECK_ARCH_AND_LOAD_SYSCALL_NR \
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, \
                 (offsetof(struct seccomp_data, arch))), \
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 0, 2), \
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, \
                 (offsetof(struct seccomp_data, nr))), \
        BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, X32_SYSCALL_BIT, 0, 1), \
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)

/* installNotifyFilter() installs a seccomp filter that generates
   user-space notifications (SECCOMP_RET_USER_NOTIF) when the process
   calls mkdir(2); the filter allows all other system calls.

   The function return value is a file descriptor from which the
   user-space notifications can be fetched. */

static int
installNotifyFilter(void)
{
    int notifyFd;

    struct sock_filter filter[] = {
        X86_64_CHECK_ARCH_AND_LOAD_SYSCALL_NR,

        /* mkdir() triggers notification to user-space supervisor */

        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_mkdir, 0, 1),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_USER_NOTIF),

        /* Every other system call is allowed */

        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
    };

    struct sock_fprog prog = {
        .len = ARRAY_SIZE(filter),
        .filter = filter,
    };

    /* Install the filter with the SECCOMP_FILTER_FLAG_NEW_LISTENER flag;
       as a result, seccomp() returns a notification file descriptor. */

    notifyFd = seccomp(SECCOMP_SET_MODE_FILTER,
                       SECCOMP_FILTER_FLAG_NEW_LISTENER, &prog);
    if (notifyFd == -1)
```

```
        err(EXIT_FAILURE, "seccomp-install-notify-filter");

    return notifyFd;
}

/* Close a pair of sockets created by socketpair() */

static void
closeSocketPair(int sockPair[2])
{
    if (close(sockPair[0]) == -1)
        err(EXIT_FAILURE, "closeSocketPair-close-0");
    if (close(sockPair[1]) == -1)
        err(EXIT_FAILURE, "closeSocketPair-close-1");
}

/* Implementation of the target process; create a child process that:

   (1) installs a seccomp filter with the
       SECCOMP_FILTER_FLAG_NEW_LISTENER flag;
   (2) writes the seccomp notification file descriptor returned from
       the previous step onto the UNIX domain socket, 'sockPair[0]';
   (3) calls mkdir(2) for each element of 'argv'.

   The function return value in the parent is the PID of the child
   process; the child does not return from this function. */

static pid_t
targetProcess(int sockPair[2], char *argv[])
{
    int    notifyFd, s;
    pid_t  targetPid;

    targetPid = fork();

    if (targetPid == -1)
        err(EXIT_FAILURE, "fork");

    if (targetPid > 0)          /* In parent, return PID of child */
        return targetPid;

    /* Child falls through to here */

    printf("T: PID = %ld\n", (long) getpid());

    /* Install seccomp filter(s) */

    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
        err(EXIT_FAILURE, "prctl");

    notifyFd = installNotifyFilter();

    /* Pass the notification file descriptor to the tracing process over
       a UNIX domain socket */

    if (sendfd(sockPair[0], notifyFd) == -1)
        err(EXIT_FAILURE, "sendfd");

    /* Notification and socket FDs are no longer needed in target */
```

```
    if (close(notifyFd) == -1)
        err(EXIT_FAILURE, "close-target-notify-fd");

    closeSocketPair(sockPair);

    /* Perform a mkdir() call for each of the command-line arguments */

    for (char **ap = argv; *ap != NULL; ap++) {
        printf("\nT: about to mkdir(\"%s\")\n", *ap);

        s = mkdir(*ap, 0700);
        if (s == -1)
            perror("T: ERROR: mkdir(2)");
        else
            printf("T: SUCCESS: mkdir(2) returned %d\n", s);
    }

    printf("\nT: terminating\n");
    exit(EXIT_SUCCESS);
}

/* Check that the notification ID provided by a SECCOMP_IOCTL_NOTIF_RECV
   operation is still valid. It will no longer be valid if the target
   process has terminated or is no longer blocked in the system call that
   generated the notification (because it was interrupted by a signal).

   This operation can be used when doing such things as accessing
   /proc/PID files in the target process in order to avoid TOCTOU race
   conditions where the PID that is returned by SECCOMP_IOCTL_NOTIF_RECV
   terminates and is reused by another process. */

static bool
cookieIsValid(int notifyFd, uint64_t id)
{
    return ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_ID_VALID, &id) == 0;
}

/* Access the memory of the target process in order to fetch the
   pathname referred to by the system call argument 'argNum' in
   'req->data.args[]'.  The pathname is returned in 'path',
   a buffer of 'len' bytes allocated by the caller.

   Returns true if the pathname is successfully fetched, and false
   otherwise. For possible causes of failure, see the comments below. */

static bool
getTargetPathname(struct seccomp_notif *req, int notifyFd,
                  int argNum, char *path, size_t len)
{
    int      procMemFd;
    char     procMemPath[PATH_MAX];
    ssize_t  nread;

    snprintf(procMemPath, sizeof(procMemPath), "/proc/%d/mem", req->pid);

    procMemFd = open(procMemPath, O_RDONLY | O_CLOEXEC);
    if (procMemFd == -1)
        return false;
```

```
    /* Check that the process whose info we are accessing is still alive
       and blocked in the system call that caused the notification.
       If the SECCOMP_IOCTL_NOTIF_ID_VALID operation (performed in
       cookieIsValid()) succeeded, we know that the /proc/PID/mem file
       descriptor that we opened corresponded to the process for which we
       received a notification. If that process subsequently terminates,
       then read() on that file descriptor will return 0 (EOF). */

    if (!cookieIsValid(notifyFd, req->id)) {
        close(procMemFd);
        return false;
    }

    /* Read bytes at the location containing the pathname argument */

    nread = pread(procMemFd, path, len, req->data.args[argNum]);

    close(procMemFd);

    if (nread <= 0)
        return false;

    /* Once again check that the notification ID is still valid. The
       case we are particularly concerned about here is that just
       before we fetched the pathname, the target's blocked system
       call was interrupted by a signal handler, and after the handler
       returned, the target carried on execution (past the interrupted
       system call). In that case, we have no guarantees about what we
       are reading, since the target's memory may have been arbitrarily
       changed by subsequent operations. */

    if (!cookieIsValid(notifyFd, req->id)) {
        perror("\tS: notification ID check failed!!!");
        return false;
    }

    /* Even if the target's system call was not interrupted by a signal,
       we have no guarantees about what was in the memory of the target
       process. (The memory may have been modified by another thread, or
       even by an external attacking process.) We therefore treat the
       buffer returned by pread() as untrusted input. The buffer should
       contain a terminating null byte; if not, then we will trigger an
       error for the target process. */

    if (strnlen(path, nread) < nread)
        return true;

    return false;
}

/* Allocate buffers for the seccomp user-space notification request and
   response structures. It is the caller's responsibility to free the
   buffers returned via 'req' and 'resp'. */

static void
allocSeccompNotifBuffers(struct seccomp_notif **req,
                         struct seccomp_notif_resp **resp,
                         struct seccomp_notif_sizes *sizes)
```

```
{
    size_t  resp_size;

    /* Discover the sizes of the structures that are used to receive
       notifications and send notification responses, and allocate
       buffers of those sizes. */

    if (seccomp(SECCOMP_GET_NOTIF_SIZES, 0, sizes) == -1)
        err(EXIT_FAILURE, "seccomp-SECCOMP_GET_NOTIF_SIZES");

    *req = malloc(sizes->seccomp_notif);
    if (*req == NULL)
        err(EXIT_FAILURE, "malloc-seccomp_notif");

    /* When allocating the response buffer, we must allow for the fact
       that the user-space binary may have been built with user-space
       headers where 'struct seccomp_notif_resp' is bigger than the
       response buffer expected by the (older) kernel. Therefore, we
       allocate a buffer that is the maximum of the two sizes. This
       ensures that if the supervisor places bytes into the response
       structure that are past the response size that the kernel expects,
       then the supervisor is not touching an invalid memory location. */

    resp_size = sizes->seccomp_notif_resp;
    if (sizeof(struct seccomp_notif_resp) > resp_size)
        resp_size = sizeof(struct seccomp_notif_resp);

    *resp = malloc(resp_size);
    if (*resp == NULL)
        err(EXIT_FAILURE, "malloc-seccomp_notif_resp");

}

/* Handle notifications that arrive via the SECCOMP_RET_USER_NOTIF file
   descriptor, 'notifyFd'. */

static void
handleNotifications(int notifyFd)
{
    bool                        pathOK;
    char                        path[PATH_MAX];
    struct seccomp_notif        *req;
    struct seccomp_notif_resp   *resp;
    struct seccomp_notif_sizes  sizes;

    allocSeccompNotifBuffers(&req, &resp, &sizes);

    /* Loop handling notifications */

    for (;;) {

        /* Wait for next notification, returning info in '*req' */

        memset(req, 0, sizes.seccomp_notif);
        if (ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_RECV, req) == -1) {
            if (errno == EINTR)
                continue;
            err(EXIT_FAILURE, "\tS: ioctl-SECCOMP_IOCTL_NOTIF_RECV");
        }
```

```
            printf("\tS: got notification (ID %#llx) for PID %d\n",
                   req->id, req->pid);

            /* The only system call that can generate a notification event
               is mkdir(2). Nevertheless, we check that the notified system
               call is indeed mkdir() as kind of future-proofing of this
               code in case the seccomp filter is later modified to
               generate notifications for other system calls. */

            if (req->data.nr != SYS_mkdir) {
                printf("\tS: notification contained unexpected "
                       "system call number; bye!!!\n");
                exit(EXIT_FAILURE);
            }

            pathOK = getTargetPathname(req, notifyFd, 0, path, sizeof(path));

            /* Prepopulate some fields of the response */

            resp->id = req->id;     /* Response includes notification ID */
            resp->flags = 0;
            resp->val = 0;

            /* If getTargetPathname() failed, trigger an EINVAL error
               response (sending this response may yield an error if the
               failure occurred because the notification ID was no longer
               valid); if the directory is in /tmp, then create it on behalf
               of the supervisor; if the pathname starts with '.', tell the
               kernel to let the target process execute the mkdir();
               otherwise, give an error for a directory pathname in any other
               location. */

            if (!pathOK) {
                resp->error = -EINVAL;
                printf("\tS: spoofing error for invalid pathname (%s)\n",
                       strerror(-resp->error));
            } else if (strncmp(path, "/tmp/", strlen("/tmp/")) == 0) {
                printf("\tS: executing: mkdir(\"%s\", %#llo)\n",
                       path, req->data.args[1]);

                if (mkdir(path, req->data.args[1]) == 0) {
                    resp->error = 0;            /* "Success" */
                    resp->val = strlen(path);   /* Used as return value of
                                                   mkdir() in target */
                    printf("\tS: success! spoofed return = %lld\n",
                           resp->val);
                } else {

                    /* If mkdir() failed in the supervisor, pass the error
                       back to the target */

                    resp->error = -errno;
                    printf("\tS: failure! (errno = %d; %s)\n", errno,
                           strerror(errno));
                }
            } else if (strncmp(path, "./", strlen("./")) == 0) {
                resp->error = resp->val = 0;
                resp->flags = SECCOMP_USER_NOTIF_FLAG_CONTINUE;
```

```
            printf("\tS: target can execute system call\n");
        } else {
            resp->error = -EOPNOTSUPP;
            printf("\tS: spoofing error response (%s)\n",
                   strerror(-resp->error));
        }

        /* Send a response to the notification */

        printf("\tS: sending response "
               "(flags = %#x; val = %lld; error = %d)\n",
               resp->flags, resp->val, resp->error);

        if (ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_SEND, resp) == -1) {
            if (errno == ENOENT)
                printf("\tS: response failed with ENOENT; "
                       "perhaps target process's syscall was "
                       "interrupted by a signal?\n");
            else
                perror("ioctl-SECCOMP_IOCTL_NOTIF_SEND");
        }

        /* If the pathname is just "/bye", then the supervisor breaks out
           of the loop and terminates. This allows us to see what happens
           if the target process makes further calls to mkdir(2). */

        if (strcmp(path, "/bye") == 0)
            break;
    }

    free(req);
    free(resp);
    printf("\tS: terminating **********\n");
    exit(EXIT_FAILURE);
}

/* Implementation of the supervisor process:

   (1) obtains the notification file descriptor from 'sockPair[1]'
   (2) handles notifications that arrive on that file descriptor. */

static void
supervisor(int sockPair[2])
{
    int notifyFd;

    notifyFd = recvfd(sockPair[1]);

    if (notifyFd == -1)
        err(EXIT_FAILURE, "recvfd");

    closeSocketPair(sockPair);  /* We no longer need the socket pair */

    handleNotifications(notifyFd);
}

int
main(int argc, char *argv[])
{
```

```
        int               sockPair[2];
        struct sigaction  sa;

        setbuf(stdout, NULL);

        if (argc < 2) {
            fprintf(stderr, "At least one pathname argument is required\n");
            exit(EXIT_FAILURE);
        }

        /* Create a UNIX domain socket that is used to pass the seccomp
           notification file descriptor from the target process to the
           supervisor process. */

        if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockPair) == -1)
            err(EXIT_FAILURE, "socketpair");

        /* Create a child process--the "target"--that installs seccomp
           filtering. The target process writes the seccomp notification
           file descriptor onto 'sockPair[0]' and then calls mkdir(2) for
           each directory in the command-line arguments. */

        (void) targetProcess(sockPair, &argv[optind]);

        /* Catch SIGCHLD when the target terminates, so that the
           supervisor can also terminate. */

        sa.sa_handler = sigchldHandler;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);
        if (sigaction(SIGCHLD, &sa, NULL) == -1)
            err(EXIT_FAILURE, "sigaction");

        supervisor(sockPair);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ioctl(2), *pidfd_getfd*(2), *pidfd_open*(2), *seccomp*(2)

A further example program can be found in the kernel source file *samples/seccomp/user-trap.c*.

NAME

select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO, fd_set − synchronous I/O multiplexing

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/select.h>

typedef /* ... */ fd_set;

int select(int nfds, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           struct timeval *_Nullable restrict timeout);

void FD_CLR(int  fd, fd_set *set);
int  FD_ISSET(int  fd, fd_set *set);
void FD_SET(int  fd, fd_set *set);
void FD_ZERO(fd_set *set);

int pselect(int nfds, fd_set *_Nullable restrict readfds,
            fd_set *_Nullable restrict writefds,
            fd_set *_Nullable restrict exceptfds,
            const struct timespec *_Nullable restrict timeout,
            const sigset_t *_Nullable restrict sigmask);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pselect():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

WARNING: **select**() can monitor only file descriptors numbers that are less than **FD_SETSIZE** (1024)—an unreasonably low limit for many modern applications—and this limitation will not change. All modern applications should instead use *poll*(2) or *epoll*(7), which do not suffer this limitation.

select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., *read*(2), or a sufficiently small *write*(2)) without blocking.

fd_set

A structure type that can represent a set of file descriptors. According to POSIX, the maximum number of file descriptors in an *fd_set* structure is the value of the macro **FD_SETSIZE**.

File descriptor sets

The principal arguments of **select**() are three "sets" of file descriptors (declared with the type *fd_set*), which allow the caller to wait for three classes of events on the specified set of file descriptors. Each of the *fd_set* arguments may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Note well: Upon return, each of the file descriptor sets is modified in place to indicate which file descriptors are currently "ready". Thus, if using **select**() within a loop, the sets *must be reinitialized* before each call.

The contents of a file descriptor set can be manipulated using the following macros:

FD_ZERO()
: This macro clears (removes all file descriptors from) *set*. It should be employed as the first step in initializing a file descriptor set.

FD_SET()
: This macro adds the file descriptor *fd* to *set*. Adding a file descriptor that is already present in the set is a no-op, and does not produce an error.

FD_CLR()
: This macro removes the file descriptor *fd* from *set*. Removing a file descriptor that is not present in the set is a no-op, and does not produce an error.

FD_ISSET()
select() modifies the contents of the sets according to the rules described below. After calling **select**(), the **FD_ISSET**() macro can be used to test if a file descriptor is still present in a set. **FD_ISSET**() returns nonzero if the file descriptor *fd* is present in *set*, and zero if it is not.

Arguments
The arguments of **select**() are as follows:

readfds The file descriptors in this set are watched to see if they are ready for reading. A file descriptor is ready for reading if a read operation will not block; in particular, a file descriptor is also ready on end-of-file.

After **select**() has returned, *readfds* will be cleared of all file descriptors except for those that are ready for reading.

writefds
The file descriptors in this set are watched to see if they are ready for writing. A file descriptor is ready for writing if a write operation will not block. However, even if a file descriptor indicates as writable, a large write may still block.

After **select**() has returned, *writefds* will be cleared of all file descriptors except for those that are ready for writing.

exceptfds
The file descriptors in this set are watched for "exceptional conditions". For examples of some exceptional conditions, see the discussion of **POLLPRI** in *poll*(2).

After **select**() has returned, *exceptfds* will be cleared of all file descriptors except for those for which an exceptional condition has occurred.

nfds This argument should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit (but see BUGS).

timeout The *timeout* argument is a *timeval* structure (shown below) that specifies the interval that **select**() should block waiting for a file descriptor to become ready. The call will block until either:

- a file descriptor becomes ready;
- the call is interrupted by a signal handler; or
- the timeout expires.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.

If both fields of the *timeval* structure are zero, then **select**() returns immediately. (This is useful for polling.)

If *timeout* is specified as NULL, **select**() blocks indefinitely waiting for a file descriptor to become ready.

pselect()
The **pselect**() system call allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

The operation of **select**() and **pselect**() is identical, other than these three differences:

- **select**() uses a timeout that is a *struct timeval* (with seconds and microseconds), while **pselect**() uses a *struct timespec* (with seconds and nanoseconds).
- **select**() may update the *timeout* argument to indicate how much time was left. **pselect**() does not change this argument.
- **select**() has no *sigmask* argument, and behaves as **pselect**() called with NULL *sigmask*.

sigmask is a pointer to a signal mask (see *sigprocmask*(2)); if it is not NULL, then **pselect**() first replaces the current signal mask by the one pointed to by *sigmask*, then does the "select" function, and then restores the original signal mask. (If *sigmask* is NULL, the signal mask is not modified during the **pselect**() call.)

Other than the difference in the precision of the *timeout* argument, the following **pselect**() call:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds,
                timeout, &sigmask);
```

is equivalent to *atomically* executing the following calls:

```
sigset_t origmask;

pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The reason that **pselect**() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of **select**() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, **pselect**() allows one to first block signals, handle the signals that have come in, then call **pselect**() with the desired *sigmask*, avoiding the race.)

The timeout

The *timeout* argument for **select**() is a structure of the following type:

```
struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

The corresponding argument for **pselect**() is a *timespec*(3) structure.

On Linux, **select**() modifies *timeout* to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1 permits either behavior.) This causes problems both when Linux code which reads *timeout* is ported to other operating systems, and when code is ported to Linux that reuses a *struct timeval* for multiple **select**()s in a loop without reinitializing it. Consider *timeout* to be undefined after **select**() returns.

RETURN VALUE

On success, **select**() and **pselect**() return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in *readfds*, *writefds*, *exceptfds*). The return value may be zero if the timeout expired before any file descriptors became ready.

On error, −1 is returned, and *errno* is set to indicate the error; the file descriptor sets are unmodified, and *timeout* becomes undefined.

ERRORS

EBADF
: An invalid file descriptor was given in one of the sets. (Perhaps a file descriptor that was already closed, or one on which an error has occurred.) However, see BUGS.

EINTR
: A signal was caught; see *signal*(7).

EINVAL
: *nfds* is negative or exceeds the **RLIMIT_NOFILE** resource limit (see *getrlimit*(2)).

EINVAL
: The value contained within *timeout* is invalid.

ENOMEM
: Unable to allocate memory for internal tables.

VERSIONS

On some other UNIX systems, **select**() can fail with the error **EAGAIN** if the system fails to allocate kernel-internal resources, rather than **ENOMEM** as Linux does. POSIX specifies this error for *poll*(2), but not for **select**(). Portable programs may wish to check for **EAGAIN** and loop, just as with **EINTR**.

STANDARDS

POSIX.1-2008.

HISTORY

select() POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants). However, note that the System V variant typically sets the timeout variable before returning, but the BSD variant does not.

pselect()

Linux 2.6.16. POSIX.1g, POSIX.1-2001.

Prior to this, it was emulated in glibc (but see BUGS).

fd_set POSIX.1-2001.

NOTES

The following header also provides the *fd_set* type: <*sys/time.h*>.

An *fd_set* is a fixed size buffer. Executing **FD_CLR**() or **FD_SET**() with a value of *fd* that is negative or is equal to or larger than **FD_SETSIZE** will result in undefined behavior. Moreover, POSIX requires *fd* to be a valid file descriptor.

The operation of **select**() and **pselect**() is not affected by the **O_NONBLOCK** flag.

The self-pipe trick

On systems that lack **pselect**(), reliable (and more portable) signal trapping can be achieved using the self-pipe trick. In this technique, a signal handler writes a byte to a pipe whose other end is monitored by **select**() in the main program. (To avoid possibly blocking when writing to a pipe that may be full or reading from a pipe that may be empty, nonblocking I/O is used when reading from and writing to the pipe.)

Emulating usleep(3)

Before the advent of *usleep*(3), some code employed a call to **select**() with all three sets empty, *nfds* zero, and a non-NULL *timeout* as a fairly portable way to sleep with subsecond precision.

Correspondence between select() and poll() notifications

Within the Linux kernel source, we find the following definitions which show the correspondence between the readable, writable, and exceptional condition notifications of **select**() and the event notifications provided by *poll*(2) and *epoll*(7):

```
#define POLLIN_SET  (EPOLLRDNORM | EPOLLRDBAND | EPOLLIN |
                     EPOLLHUP | EPOLLERR)
                    /* Ready for reading */
#define POLLOUT_SET (EPOLLWRBAND | EPOLLWRNORM | EPOLLOUT |
                     EPOLLERR)
                    /* Ready for writing */
#define POLLEX_SET  (EPOLLPRI)
                    /* Exceptional condition */
```

Multithreaded applications

If a file descriptor being monitored by **select**() is closed in another thread, the result is unspecified. On some UNIX systems, **select**() unblocks and returns, with an indication that the file descriptor is ready (a subsequent I/O operation will likely fail with an error, unless another process reopens the file descriptor between the time **select**() returned and the I/O operation is performed). On Linux (and some other systems), closing the file descriptor in another thread has no effect on **select**(). In summary, any application that relies on a particular behavior in this scenario must be considered buggy.

C library/kernel differences

The Linux kernel allows file descriptor sets of arbitrary size, determining the length of the sets to be checked from the value of *nfds*. However, in the glibc implementation, the *fd_set* type is fixed in size. See also BUGS.

The **pselect**() interface described in this page is implemented by glibc. The underlying Linux system call is named **pselect6**(). This system call has somewhat different behavior from the glibc wrapper function.

The Linux **pselect6**() system call modifies its *timeout* argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc **pselect**() function does not modify its *timeout* argument; this is the behavior required

by POSIX.1-2001.

The final argument of the **pselect6**() system call is not a *sigset_t* * pointer, but is instead a structure of the form:

```
struct {
    const kernel_sigset_t *ss;   /* Pointer to signal set */
    size_t ss_len;               /* Size (in bytes) of object
                                    pointed to by 'ss' */
};
```

This allows the system call to obtain both a pointer to the signal set and its size, while allowing for the fact that most architectures support a maximum of 6 arguments to a system call. See *sigprocmask*(2) for a discussion of the difference between the kernel and libc notion of the signal set.

Historical glibc details

glibc 2.0 provided an incorrect version of **pselect**() that did not take a *sigmask* argument.

From glibc 2.1 to glibc 2.2.1, one must define **_GNU_SOURCE** in order to obtain the declaration of **pselect**() from <*sys/select.h*>.

BUGS

POSIX allows an implementation to define an upper limit, advertised via the constant **FD_SETSIZE**, on the range of file descriptors that can be specified in a file descriptor set. The Linux kernel imposes no fixed limit, but the glibc implementation makes *fd_set* a fixed-size type, with **FD_SETSIZE** defined as 1024, and the **FD_***() macros operating according to that limit. To monitor file descriptors greater than 1023, use *poll*(2) or *epoll*(7) instead.

The implementation of the *fd_set* arguments as value-result arguments is a design error that is avoided in *poll*(2) and *epoll*(7).

According to POSIX, **select**() should check all specified file descriptors in the three file descriptor sets, up to the limit *nfds−1*. However, the current implementation ignores any file descriptor in these sets that is greater than the maximum file descriptor number that the process currently has open. According to POSIX, any such file descriptor that is specified in one of the sets should result in the error **EBADF**.

Starting with glibc 2.1, glibc provided an emulation of **pselect**() that was implemented using *sigprocmask*(2) and **select**(). This implementation remained vulnerable to the very race condition that **pselect**() was designed to prevent. Modern versions of glibc use the (race-free) **pselect**() system call on kernels where it is provided.

On Linux, **select**() may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has the wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use **O_NONBLOCK** on sockets that should not block.

On Linux, **select**() also modifies *timeout* if the call is interrupted by a signal handler (i.e., the **EINTR** error return). This is not permitted by POSIX.1. The Linux **pselect**() system call has the same behavior, but the glibc wrapper hides this behavior by internally copying the *timeout* to a local variable and passing that variable to the system call.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>

int
main(void)
{
    int             retval;
    fd_set          rfds;
    struct timeval  tv;

    /* Watch stdin (fd 0) to see when it has input. */
```

```
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);

        /* Wait up to five seconds. */

        tv.tv_sec = 5;
        tv.tv_usec = 0;

        retval = select(1, &rfds, NULL, NULL, &tv);
        /* Don't rely on the value of tv now! */

        if (retval == -1)
            perror("select()");
        else if (retval)
            printf("Data is available now.\n");
            /* FD_ISSET(0, &rfds) will be true. */
        else
            printf("No data within five seconds.\n");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

accept(2), *connect*(2), *poll*(2), *read*(2), *recv*(2), *restart_syscall*(2), *send*(2), *sigprocmask*(2), *write*(2), *timespec*(3), *epoll*(7), *time*(7)

For a tutorial with discussion and examples, see *select_tut*(2).

NAME

select, pselect − synchronous I/O multiplexing

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

See *select*(2)

DESCRIPTION

The **select**() and **pselect**() system calls are used to efficiently monitor multiple file descriptors, to see if any of them is, or becomes, "ready"; that is, to see whether I/O becomes possible, or an "exceptional condition" has occurred on any of the file descriptors.

This page provides background and tutorial information on the use of these system calls. For details of the arguments and semantics of **select**() and **pselect**(), see *select*(2).

Combining signal and data events

pselect() is useful if you are waiting for a signal as well as for file descriptor(s) to become ready for I/O. Programs that receive signals normally use the signal handler only to raise a global flag. The global flag will indicate that the event must be processed in the main loop of the program. A signal will cause the **select**() (or *pselect*()) call to return with *errno* set to **EINTR**. This behavior is essential so that signals can be processed in the main loop of the program, otherwise **select**() would block indefinitely.

Now, somewhere in the main loop will be a conditional to check the global flag. So we must ask: what if a signal arrives after the conditional, but before the **select**() call? The answer is that **select**() would block indefinitely, even though an event is actually pending. This race condition is solved by the **pselect**() call. This call can be used to set the signal mask to a set of signals that are to be received only within the **pselect**() call. For instance, let us say that the event in question was the exit of a child process. Before the start of the main loop, we would block **SIGCHLD** using *sigprocmask*(2). Our **pselect**() call would enable **SIGCHLD** by using an empty signal mask. Our program would look like:

```
static volatile sig_atomic_t got_SIGCHLD = 0;

static void
child_sig_handler(int sig)
{
    got_SIGCHLD = 1;
}

int
main(int argc, char *argv[])
{
    sigset_t sigmask, empty_mask;
    struct sigaction sa;
    fd_set readfds, writefds, exceptfds;
    int r;

    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &sigmask, NULL) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    sa.sa_flags = 0;
    sa.sa_handler = child_sig_handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
```

```
        sigemptyset(&empty_mask);

        for (;;) {          /* main loop */
            /* Initialize readfds, writefds, and exceptfds
               before the pselect() call. (Code omitted.) */

            r = pselect(nfds, &readfds, &writefds, &exceptfds,
                        NULL, &empty_mask);
            if (r == -1 && errno != EINTR) {
                /* Handle error */
            }

            if (got_SIGCHLD) {
                got_SIGCHLD = 0;

                /* Handle signalled event here; e.g., wait() for all
                   terminated children. (Code omitted.) */
            }

            /* main body of program */
        }
    }
```

Practical

So what is the point of **select**()? Can't I just read and write to my file descriptors whenever I want? The point of **select**() is that it watches multiple descriptors at the same time and properly puts the process to sleep if there is no activity. UNIX programmers often find themselves in a position where they have to handle I/O from more than one file descriptor where the data flow may be intermittent. If you were to merely create a sequence of *read*(2) and *write*(2) calls, you would find that one of your calls may block waiting for data from/to a file descriptor, while another file descriptor is unused though ready for I/O. **select**() efficiently copes with this situation.

Select law

Many people who try to use **select**() come across behavior that is difficult to understand and produces nonportable or borderline results. For instance, the above program is carefully written not to block at any point, even though it does not set its file descriptors to nonblocking mode. It is easy to introduce subtle errors that will remove the advantage of using **select**(), so here is a list of essentials to watch for when using **select**().

1. You should always try to use **select**() without a timeout. Your program should have nothing to do if there is no data available. Code that depends on timeouts is not usually portable and is difficult to debug.
2. The value *nfds* must be properly calculated for efficiency as explained above.
3. No file descriptor must be added to any set if you do not intend to check its result after the **select**() call, and respond appropriately. See next rule.
4. After **select**() returns, all file descriptors in all sets should be checked to see if they are ready.
5. The functions *read*(2), *recv*(2), *write*(2), and *send*(2) do *not* necessarily read/write the full amount of data that you have requested. If they do read/write the full amount, it's because you have a low traffic load and a fast stream. This is not always going to be the case. You should cope with the case of your functions managing to send or receive only a single byte.
6. Never read/write only in single bytes at a time unless you are really sure that you have a small amount of data to process. It is extremely inefficient not to read/write as much data as you can buffer each time. The buffers in the example below are 1024 bytes although they could easily be made larger.
7. Calls to *read*(2), *recv*(2), *write*(2), *send*(2), and **select**() can fail with the error **EINTR**, and calls to *read*(2), *recv*(2), *write*(2), and *send*(2) can fail with *errno* set to **EAGAIN** (**EWOULDBLOCK**). These results must be properly managed (not done properly above). If your program is not going

to receive any signals, then it is unlikely you will get **EINTR**. If your program does not set non-blocking I/O, you will not get **EAGAIN**.

8. Never call *read*(2), *recv*(2), *write*(2), or *send*(2) with a buffer length of zero.
9. If the functions *read*(2), *recv*(2), *write*(2), and *send*(2) fail with errors other than those listed in **7.**, or one of the input functions returns 0, indicating end of file, then you should *not* pass that file descriptor to **select**() again. In the example below, I close the file descriptor immediately, and then set it to −1 to prevent it being included in a set.
10. The timeout value must be initialized with each new call to **select**(), since some operating systems modify the structure. **pselect**() however does not modify its timeout structure.
11. Since **select**() modifies its file descriptor sets, if the call is being used in a loop, then the sets must be reinitialized before each call.

RETURN VALUE

See *select*(2).

NOTES

Generally speaking, all operating systems that support sockets also support **select**(). **select**() can be used to solve many problems in a portable and efficient way that naive programmers try to solve in a more complicated manner using threads, forking, IPCs, signals, memory sharing, and so on.

The *poll*(2) system call has the same functionality as **select**(), and is somewhat more efficient when monitoring sparse file descriptor sets. It is nowadays widely available, but historically was less portable than **select**().

The Linux-specific *epoll*(7) API provides an interface that is more efficient than *select*(2) and *poll*(2) when monitoring large numbers of file descriptors.

EXAMPLES

Here is an example that better demonstrates the true utility of **select**(). The listing below is a TCP forwarding program that forwards from one TCP port to another.

```
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>

static int forward_port;

#undef max
#define max(x, y) ((x) > (y) ? (x) : (y))

static int
listen_socket(int listen_port)
{
    int                 lfd;
    int                 yes;
    struct sockaddr_in  addr;

    lfd = socket(AF_INET, SOCK_STREAM, 0);
    if (lfd == -1) {
        perror("socket");
        return -1;
    }

    yes = 1;
```

```
    if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR,
                   &yes, sizeof(yes)) == -1)
    {
        perror("setsockopt");
        close(lfd);
        return -1;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_port = htons(listen_port);
    addr.sin_family = AF_INET;
    if (bind(lfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        perror("bind");
        close(lfd);
        return -1;
    }

    printf("accepting connections on port %d\n", listen_port);
    listen(lfd, 10);
    return lfd;
}

static int
connect_socket(int connect_port, char *address)
{
    int                 cfd;
    struct sockaddr_in  addr;

    cfd = socket(AF_INET, SOCK_STREAM, 0);
    if (cfd == -1) {
        perror("socket");
        return -1;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_port = htons(connect_port);
    addr.sin_family = AF_INET;

    if (!inet_aton(address, (struct in_addr *) &addr.sin_addr.s_addr)) {
        fprintf(stderr, "inet_aton(): bad IP address format\n");
        close(cfd);
        return -1;
    }

    if (connect(cfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        perror("connect()");
        shutdown(cfd, SHUT_RDWR);
        close(cfd);
        return -1;
    }
    return cfd;
}

#define SHUT_FD1 do {                                   \
                     if (fd1 >= 0) {                    \
                         shutdown(fd1, SHUT_RDWR);      \
                         close(fd1);                    \
                         fd1 = -1;                      \
                     }                                  \
```

```
                } while (0)

#define SHUT_FD2 do {                                   \
                     if (fd2 >= 0) {                    \
                         shutdown(fd2, SHUT_RDWR);      \
                         close(fd2);                    \
                         fd2 = -1;                      \
                     }                                  \
                 } while (0)

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int      h;
    int      ready, nfds;
    int      fd1 = -1, fd2 = -1;
    int      buf1_avail = 0, buf1_written = 0;
    int      buf2_avail = 0, buf2_written = 0;
    char     buf1[BUF_SIZE], buf2[BUF_SIZE];
    fd_set   readfds, writefds, exceptfds;
    ssize_t  nbytes;

    if (argc != 4) {
        fprintf(stderr, "Usage\n\tfwd <listen-port> "
                "<forward-to-port> <forward-to-ip-address>\n");
        exit(EXIT_FAILURE);
    }

    signal(SIGPIPE, SIG_IGN);

    forward_port = atoi(argv[2]);

    h = listen_socket(atoi(argv[1]));
    if (h == -1)
        exit(EXIT_FAILURE);

    for (;;) {
        nfds = 0;

        FD_ZERO(&readfds);
        FD_ZERO(&writefds);
        FD_ZERO(&exceptfds);
        FD_SET(h, &readfds);
        nfds = max(nfds, h);

        if (fd1 > 0 && buf1_avail < BUF_SIZE)
            FD_SET(fd1, &readfds);
            /* Note: nfds is updated below, when fd1 is added to
               exceptfds. */
        if (fd2 > 0 && buf2_avail < BUF_SIZE)
            FD_SET(fd2, &readfds);

        if (fd1 > 0 && buf2_avail - buf2_written > 0)
            FD_SET(fd1, &writefds);
        if (fd2 > 0 && buf1_avail - buf1_written > 0)
            FD_SET(fd2, &writefds);
```

```
        if (fd1 > 0) {
            FD_SET(fd1, &exceptfds);
            nfds = max(nfds, fd1);
        }
        if (fd2 > 0) {
            FD_SET(fd2, &exceptfds);
            nfds = max(nfds, fd2);
        }

        ready = select(nfds + 1, &readfds, &writefds, &exceptfds, NULL);

        if (ready == -1 && errno == EINTR)
            continue;

        if (ready == -1) {
            perror("select()");
            exit(EXIT_FAILURE);
        }

        if (FD_ISSET(h, &readfds)) {
            socklen_t addrlen;
            struct sockaddr_in client_addr;
            int fd;

            addrlen = sizeof(client_addr);
            memset(&client_addr, 0, addrlen);
            fd = accept(h, (struct sockaddr *) &client_addr, &addrlen);
            if (fd == -1) {
                perror("accept()");
            } else {
                SHUT_FD1;
                SHUT_FD2;
                buf1_avail = buf1_written = 0;
                buf2_avail = buf2_written = 0;
                fd1 = fd;
                fd2 = connect_socket(forward_port, argv[3]);
                if (fd2 == -1)
                    SHUT_FD1;
                else
                    printf("connect from %s\n",
                           inet_ntoa(client_addr.sin_addr));

                /* Skip any events on the old, closed file
                   descriptors. */

                continue;
            }
        }

        /* NB: read OOB data before normal reads. */

        if (fd1 > 0 && FD_ISSET(fd1, &exceptfds)) {
            char c;

            nbytes = recv(fd1, &c, 1, MSG_OOB);
            if (nbytes < 1)
                SHUT_FD1;
            else
                send(fd2, &c, 1, MSG_OOB);
```

```
        }
        if (fd2 > 0 && FD_ISSET(fd2, &exceptfds)) {
            char c;

            nbytes = recv(fd2, &c, 1, MSG_OOB);
            if (nbytes < 1)
                SHUT_FD2;
            else
                send(fd1, &c, 1, MSG_OOB);
        }
        if (fd1 > 0 && FD_ISSET(fd1, &readfds)) {
            nbytes = read(fd1, buf1 + buf1_avail,
                          BUF_SIZE - buf1_avail);
            if (nbytes < 1)
                SHUT_FD1;
            else
                buf1_avail += nbytes;
        }
        if (fd2 > 0 && FD_ISSET(fd2, &readfds)) {
            nbytes = read(fd2, buf2 + buf2_avail,
                          BUF_SIZE - buf2_avail);
            if (nbytes < 1)
                SHUT_FD2;
            else
                buf2_avail += nbytes;
        }
        if (fd1 > 0 && FD_ISSET(fd1, &writefds) && buf2_avail > 0) {
            nbytes = write(fd1, buf2 + buf2_written,
                           buf2_avail - buf2_written);
            if (nbytes < 1)
                SHUT_FD1;
            else
                buf2_written += nbytes;
        }
        if (fd2 > 0 && FD_ISSET(fd2, &writefds) && buf1_avail > 0) {
            nbytes = write(fd2, buf1 + buf1_written,
                           buf1_avail - buf1_written);
            if (nbytes < 1)
                SHUT_FD2;
            else
                buf1_written += nbytes;
        }

        /* Check if write data has caught read data. */

        if (buf1_written == buf1_avail)
            buf1_written = buf1_avail = 0;
        if (buf2_written == buf2_avail)
            buf2_written = buf2_avail = 0;

        /* One side has closed the connection, keep
           writing to the other side until empty. */

        if (fd1 < 0 && buf1_avail - buf1_written == 0)
            SHUT_FD2;
        if (fd2 < 0 && buf2_avail - buf2_written == 0)
            SHUT_FD1;
    }
    exit(EXIT_SUCCESS);
```

```
}
```

The above program properly forwards most kinds of TCP connections including OOB signal data transmitted by **telnet** servers. It handles the tricky problem of having data flow in both directions simultaneously. You might think it more efficient to use a *fork*(2) call and devote a thread to each stream. This becomes more tricky than you might suspect. Another idea is to set nonblocking I/O using *fcntl*(2). This also has its problems because you end up using inefficient timeouts.

The program does not handle more than one simultaneous connection at a time, although it could easily be extended to do this with a linked list of buffers—one for each connection. At the moment, new connections cause the current connection to be dropped.

SEE ALSO

accept(2), *connect*(2), *poll*(2), *read*(2), *recv*(2), *select*(2), *send*(2), *sigprocmask*(2), *write*(2), *epoll*(7)

NAME

semctl − System V semaphore control operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sem.h>

int semctl(int *semid***, int** *semnum***, int** *op***, ...);**

DESCRIPTION

semctl() performs the control operation specified by *op* on the System V semaphore set identified by *semid*, or on the *semnum*-th semaphore of that set. (The semaphores in a set are numbered starting at 0.)

This function has three or four arguments, depending on *op*. When there are four, the fourth has the type *union semun*. The *calling program* must define this union as follows:

```
union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                (Linux-specific) */
};
```

The *semid_ds* data structure is defined in <*sys/sem.h*> as follows:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions */
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Creation time/time of last
                                  modification via semctl() */
    unsigned long   sem_nsems; /* No. of semaphores in set */
};
```

The fields of the *semid_ds* structure are as follows:

sem_perm This is an *ipc_perm* structure (see below) that specifies the access permissions on the semaphore set.

sem_otime Time of last *semop*(2) system call.

sem_ctime Time of creation of semaphore set or time of last **semctl**() **IPCSET**, **SETVAL**, or **SETALL** operation.

sem_nsems Number of semaphores in the set. Each semaphore of the set is referenced by a nonnegative integer ranging from **0** to *sem_nsems−1*.

The *ipc_perm* structure is defined as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
    key_t          __key; /* Key supplied to semget(2) */
    uid_t          uid;   /* Effective UID of owner */
    gid_t          gid;   /* Effective GID of owner */
    uid_t          cuid;  /* Effective UID of creator */
    gid_t          cgid;  /* Effective GID of creator */
    unsigned short mode;  /* Permissions */
    unsigned short __seq; /* Sequence number */
};
```

The least significant 9 bits of the *mode* field of the *ipc_perm* structure define the access permissions for the shared memory segment. The permission bits are as follows:

0400 Read by user
0200 Write by user
0040 Read by group

0020 Write by group
0004 Read by others
0002 Write by others

In effect, "write" means "alter" for a semaphore set. Bits 0100, 0010, and 0001 (the execute bits) are unused by the system.

Valid values for *op* are:

IPC_STAT
Copy information from the kernel data structure associated with *semid* into the *semid_ds* structure pointed to by *arg.buf*. The argument *semnum* is ignored. The calling process must have read permission on the semaphore set.

IPC_SET
Write the values of some members of the *semid_ds* structure pointed to by *arg.buf* to the kernel data structure associated with this semaphore set, updating also its *sem_ctime* member.

The following members of the structure are updated: *sem_perm.uid*, *sem_perm.gid*, and (the least significant 9 bits of) *sem_perm.mode*.

The effective UID of the calling process must match the owner (*sem_perm.uid*) or creator (*sem_perm.cuid*) of the semaphore set, or the caller must be privileged. The argument *semnum* is ignored.

IPC_RMID
Immediately remove the semaphore set, awakening all processes blocked in *semop*(2) calls on the set (with an error return and *errno* set to **EIDRM**). The effective user ID of the calling process must match the creator or owner of the semaphore set, or the caller must be privileged. The argument *semnum* is ignored.

IPC_INFO (Linux-specific)
Return information about system-wide semaphore limits and parameters in the structure pointed to by *arg.__buf*. This structure is of type *seminfo*, defined in <*sys/sem.h*> if the **_GNU_SOURCE** feature test macro is defined:

```
struct  seminfo {
    int semmap;  /* Number of entries in semaphore
                    map; unused within kernel */
    int semmni;  /* Maximum number of semaphore sets */
    int semmns;  /* Maximum number of semaphores in all
                    semaphore sets */
    int semmnu;  /* System-wide maximum number of undo
                    structures; unused within kernel */
    int semmsl;  /* Maximum number of semaphores in a
                    set */
    int semopm;  /* Maximum number of operations for
                    semop(2) */
    int semume;  /* Maximum number of undo entries per
                    process; unused within kernel */
    int semusz;  /* Size of struct sem_undo */
    int semvmx;  /* Maximum semaphore value */
    int semaem;  /* Max. value that can be recorded for
                    semaphore adjustment (SEM_UNDO) */
};
```

The *semmsl*, *semmns*, *semopm*, and *semmni* settings can be changed via */proc/sys/kernel/sem*; see *proc*(5) for details.

SEM_INFO (Linux-specific)
Return a *seminfo* structure containing the same information as for **IPC_INFO**, except that the following fields are returned with information about system resources consumed by semaphores: the *semusz* field returns the number of semaphore sets that currently exist on the system; and the *semaem* field returns the total number of semaphores in all semaphore sets on the system.

SEM_STAT (Linux-specific)
Return a *semid_ds* structure as for **IPC_STAT**. However, the *semid* argument is not a semaphore identifier, but instead an index into the kernel's internal array that maintains information about all semaphore sets on the system.

SEM_STAT_ANY (Linux-specific, since Linux 4.17)
Return a *semid_ds* structure as for **SEM_STAT**. However, *sem_perm.mode* is not checked for read access for *semid* meaning that any user can employ this operation (just as any user may read */proc/sysvipc/sem* to obtain the same information).

GETALL
Return **semval** (i.e., the current value) for all semaphores of the set into *arg.array*. The argument *semnum* is ignored. The calling process must have read permission on the semaphore set.

GETNCNT
Return the **semncnt** value for the *semnum*−th semaphore of the set (i.e., the number of processes waiting for the semaphore's value to increase). The calling process must have read permission on the semaphore set.

GETPID
Return the **sempid** value for the *semnum*−th semaphore of the set. This is the PID of the process that last performed an operation on that semaphore (but see NOTES). The calling process must have read permission on the semaphore set.

GETVAL
Return **semval** (i.e., the semaphore value) for the *semnum*−th semaphore of the set. The calling process must have read permission on the semaphore set.

GETZCNT
Return the **semzcnt** value for the *semnum*−th semaphore of the set (i.e., the number of processes waiting for the semaphore value to become 0). The calling process must have read permission on the semaphore set.

SETALL
Set the **semval** values for all semaphores of the set using *arg.array*, updating also the *sem_ctime* member of the *semid_ds* structure associated with the set. Undo entries (see *semop*(2)) are cleared for altered semaphores in all processes. If the changes to semaphore values would permit blocked *semop*(2) calls in other processes to proceed, then those processes are woken up. The argument *semnum* is ignored. The calling process must have alter (write) permission on the semaphore set.

SETVAL
Set the semaphore value (**semval**) to *arg.val* for the *semnum*−th semaphore of the set, updating also the *sem_ctime* member of the *semid_ds* structure associated with the set. Undo entries are cleared for altered semaphores in all processes. If the changes to semaphore values would permit blocked *semop*(2) calls in other processes to proceed, then those processes are woken up. The calling process must have alter permission on the semaphore set.

RETURN VALUE
On success, **semctl**() returns a nonnegative value depending on *op* as follows:

GETNCNT
the value of **semncnt**.

GETPID
the value of **sempid**.

GETVAL
the value of **semval**.

GETZCNT
the value of **semzcnt**.

IPC_INFO
the index of the highest used entry in the kernel's internal array recording information about all semaphore sets. (This information can be used with repeated **SEM_STAT** or

SEM_STAT_ANY operations to obtain information about all semaphore sets on the system.)

SEM_INFO
as for **IPC_INFO**.

SEM_STAT
the identifier of the semaphore set whose index was given in *semid*.

SEM_STAT_ANY
as for **SEM_STAT**.

All other *op* values return 0 on success.

On failure, **semctl**() returns −1 and sets *errno* to indicate the error.

ERRORS

EACCES
The argument *op* has one of the values **GETALL**, **GETPID**, **GETVAL**, **GETNCNT**, **GETZCNT**, **IPC_STAT**, **SEM_STAT**, **SEM_STAT_ANY**, **SETALL**, or **SETVAL** and the calling process does not have the required permissions on the semaphore set and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EFAULT
The address pointed to by *arg.buf* or *arg.array* isn't accessible.

EIDRM
The semaphore set was removed.

EINVAL
Invalid value for *op* or *semid*. Or: for a **SEM_STAT** operation, the index value specified in *semid* referred to an array slot that is currently unused.

EPERM
The argument *op* has the value **IPC_SET** or **IPC_RMID** but the effective user ID of the calling process is not the creator (as found in *sem_perm.cuid*) or the owner (as found in *sem_perm.uid*) of the semaphore set, and the process does not have the **CAP_SYS_ADMIN** capability.

ERANGE
The argument *op* has the value **SETALL** or **SETVAL** and the value to which **semval** is to be set (for some semaphore of the set) is less than 0 or greater than the implementation limit **SEMVMX**.

VERSIONS

POSIX.1 specifies the *sem_nsems* field of the *semid_ds* structure as having the type *unsigned short*, and the field is so defined on most other systems. It was also so defined on Linux 2.2 and earlier, but, since Linux 2.4, the field has the type *unsigned long*.

The sempid value

POSIX.1 defines *sempid* as the "process ID of [the] last operation" on a semaphore, and explicitly notes that this value is set by a successful *semop*(2) call, with the implication that no other interface affects the *sempid* value.

While some implementations conform to the behavior specified in POSIX.1, others do not. (The fault here probably lies with POSIX.1 inasmuch as it likely failed to capture the full range of existing implementation behaviors.) Various other implementations also update *sempid* for the other operations that update the value of a semaphore: the **SETVAL** and **SETALL** operations, as well as the semaphore adjustments performed on process termination as a consequence of the use of the **SEM_UNDO** flag (see *semop*(2)).

Linux also updates *sempid* for **SETVAL** operations and semaphore adjustments. However, somewhat inconsistently, up to and including Linux 4.5, the kernel did not update *sempid* for **SETALL** operations. This was rectified in Linux 4.6.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

Various fields in a *struct semid_ds* were typed as *short* under Linux 2.2 and have become *long* under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by an **IPC_64** flag in *op*.)

In some earlier versions of glibc, the *semun* union was defined in <*sys/sem.h*>, but POSIX.1 requires that the caller define this union. On versions of glibc where this union is *not* defined, the macro **_SEM_SEMUN_UNDEFINED** is defined in <*sys/sem.h*>.

NOTES

The **IPC_INFO**, **SEM_STAT**, and **SEM_INFO** operations are used by the *ipcs*(1) program to provide information on allocated resources. In the future these may modified or moved to a */proc* filesystem interface.

The following system limit on semaphore sets affects a **semctl**() call:

SEMVMX
Maximum value for **semval**: implementation dependent (32767).

For greater portability, it is best to always call **semctl**() with four arguments.

EXAMPLES

See *shmop*(2).

SEE ALSO

ipc(2), *semget*(2), *semop*(2), *capabilities*(7), *sem_overview*(7), *sysvipc*(7)

NAME

semget − get a System V semaphore set identifier

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sem.h>

int semget(key_t *key***, int** *nsems***, int** *semflg***);**

DESCRIPTION

The **semget**() system call returns the System V semaphore set identifier associated with the argument *key*. It may be used either to obtain the identifier of a previously created semaphore set (when *semflg* is zero and *key* does not have the value **IPC_PRIVATE**), or to create a new set.

A new set of *nsems* semaphores is created if *key* has the value **IPC_PRIVATE** or if no existing semaphore set is associated with *key* and **IPC_CREAT** is specified in *semflg*.

If *semflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a semaphore set already exists for *key*, then **semget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for *open*(2).)

Upon creation, the least significant 9 bits of the argument *semflg* define the permissions (for owner, group, and others) for the semaphore set. These bits have the same format, and the same meaning, as the *mode* argument of *open*(2) (though the execute permissions are not meaningful for semaphores, and write permissions mean permission to alter semaphore values).

When creating a new semaphore set, **semget**() initializes the set's associated data structure, *semid_ds* (see *semctl*(2)), as follows:

- *sem_perm.cuid* and *sem_perm.uid* are set to the effective user ID of the calling process.
- *sem_perm.cgid* and *sem_perm.gid* are set to the effective group ID of the calling process.
- The least significant 9 bits of *sem_perm.mode* are set to the least significant 9 bits of *semflg*.
- *sem_nsems* is set to the value of *nsems*.
- *sem_otime* is set to 0.
- *sem_ctime* is set to the current time.

The argument *nsems* can be 0 (a don't care) when a semaphore set is not being created. Otherwise, *nsems* must be greater than 0 and less than or equal to the maximum number of semaphores per semaphore set (**SEMMSL**).

If the semaphore set already exists, the permissions are verified.

RETURN VALUE

On success, **semget**() returns the semaphore set identifier (a nonnegative integer). On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: A semaphore set exists for *key*, but the calling process does not have permission to access the set, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EEXIST
: **IPC_CREAT** and **IPC_EXCL** were specified in *semflg*, but a semaphore set already exists for *key*.

EINVAL
: *nsems* is less than 0 or greater than the limit on the number of semaphores per semaphore set (**SEMMSL**).

EINVAL
: A semaphore set corresponding to *key* already exists, but *nsems* is larger than the number of semaphores in that set.

ENOENT
No semaphore set exists for *key* and *semflg* did not specify **IPC_CREAT**.

ENOMEM
A semaphore set has to be created but the system does not have enough memory for the new data structure.

ENOSPC
A semaphore set has to be created but the system limit for the maximum number of semaphore sets (**SEMMNI**), or the system wide maximum number of semaphores (**SEMMNS**), would be exceeded.

STANDARDS
POSIX.1-2008.

HISTORY
SVr4, POSIX.1-2001.

NOTES
IPC_PRIVATE isn't a flag field but a *key_t* type. If this special value is used for *key*, the system call ignores all but the least significant 9 bits of *semflg* and creates a new semaphore set (on success).

Semaphore initialization
The values of the semaphores in a newly created set are indeterminate. (POSIX.1-2001 and POSIX.1-2008 are explicit on this point, although POSIX.1-2008 notes that a future version of the standard may require an implementation to initialize the semaphores to 0.) Although Linux, like many other implementations, initializes the semaphore values to 0, a portable application cannot rely on this: it should explicitly initialize the semaphores to the desired values.

Initialization can be done using *semctl*(2) **SETVAL** or **SETALL** operation. Where multiple peers do not know who will be the first to initialize the set, checking for a nonzero *sem_otime* in the associated data structure retrieved by a *semctl*(2) **IPC_STAT** operation can be used to avoid races.

Semaphore limits
The following limits on semaphore set resources affect the **semget**() call:

SEMMNI
System-wide limit on the number of semaphore sets. Before Linux 3.19, the default value for this limit was 128. Since Linux 3.19, the default value is 32,000. On Linux, this limit can be read and modified via the fourth field of */proc/sys/kernel/sem*.

SEMMSL
Maximum number of semaphores per semaphore ID. Before Linux 3.19, the default value for this limit was 250. Since Linux 3.19, the default value is 32,000. On Linux, this limit can be read and modified via the first field of */proc/sys/kernel/sem*.

SEMMNS
System-wide limit on the number of semaphores: policy dependent (on Linux, this limit can be read and modified via the second field of */proc/sys/kernel/sem*). Note that the number of semaphores system-wide is also limited by the product of **SEMMSL** and **SEMMNI**.

BUGS
The name choice **IPC_PRIVATE** was perhaps unfortunate, **IPC_NEW** would more clearly show its function.

EXAMPLES
The program shown below uses **semget**() to create a new semaphore set or retrieve the ID of an existing set. It generates the *key* for **semget**() using *ftok*(3). The first two command-line arguments are used as the *pathname* and *proj_id* arguments for *ftok*(3). The third command-line argument is an integer that specifies the *nsems* argument for **semget**(). Command-line options can be used to specify the **IPC_CREAT** (*−c*) and **IPC_EXCL** (*−x*) flags for the call to **semget**(). The usage of this program is demonstrated below.

We first create two files that will be used to generate keys using *ftok*(3), create two semaphore sets using those files, and then list the sets using *ipcs*(1):

```
$ touch mykey mykey2
$ ./t_semget -c mykey p 1
```

```
ID = 9
$ ./t_semget -c mykey2 p 2
ID = 10
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x7004136d 9          mtk        600        1
0x70041368 10         mtk        600        2
```

Next, we demonstrate that when *semctl*(2) is given the same *key* (as generated by the same arguments to *ftok*(3)), it returns the ID of the already existing semaphore set:

```
$ ./t_semget -c mykey p 1
ID = 9
```

Finally, we demonstrate the kind of collision that can occur when *ftok*(3) is given different *pathname* arguments that have the same inode number:

```
$ ln mykey link
$ ls -i1 link mykey
2233197 link
2233197 mykey
$ ./t_semget link p 1       # Generates same key as 'mykey'
ID = 9
```

Program source

```
/* t_semget.c

   Licensed under GNU General Public License v2 or later.
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

static void
usage(const char *pname)
{
    fprintf(stderr, "Usage: %s [-cx] pathname proj-id num-sems\n",
            pname);
    fprintf(stderr, "    -c           Use IPC_CREAT flag\n");
    fprintf(stderr, "    -x           Use IPC_EXCL flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int    semid, nsems, flags, opt;
    key_t  key;

    flags = 0;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c': flags |= IPC_CREAT;   break;
        case 'x': flags |= IPC_EXCL;    break;
        default:  usage(argv[0]);
        }
    }
```

```
        if (argc != optind + 3)
            usage(argv[0]);

        key = ftok(argv[optind], argv[optind + 1][0]);
        if (key == -1) {
            perror("ftok");
            exit(EXIT_FAILURE);
        }

        nsems = atoi(argv[optind + 2]);

        semid = semget(key, nsems, flags | 0600);
        if (semid == -1) {
            perror("semget");
            exit(EXIT_FAILURE);
        }

        printf("ID = %d\n", semid);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

semctl(2), *semop*(2), *ftok*(3), *capabilities*(7), *sem_overview*(7), *sysvipc*(7)

NAME

semop, semtimedop − System V semaphore operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
int semtimedop(int semid, struct sembuf *sops, size_t nsops,
               const struct timespec *_Nullable timeout);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

semtimedop():
_GNU_SOURCE

DESCRIPTION

Each semaphore in a System V semaphore set has the following associated values:

```
unsigned short  semval;   /* semaphore value */
unsigned short  semzcnt;  /* # waiting for zero */
unsigned short  semncnt;  /* # waiting for increase */
pid_t           sempid;   /* PID of process that last
                             modified the semaphore value */
```

semop() performs operations on selected semaphores in the set indicated by *semid*. Each of the *nsops* elements in the array pointed to by *sops* is a structure that specifies an operation to be performed on a single semaphore. The elements of this structure are of type *struct sembuf*, containing the following members:

```
unsigned short sem_num;  /* semaphore number */
short          sem_op;   /* semaphore operation */
short          sem_flg;  /* operation flags */
```

Flags recognized in *sem_flg* are **IPC_NOWAIT** and **SEM_UNDO**. If an operation specifies **SEM_UNDO**, it will be automatically undone when the process terminates.

The set of operations contained in *sops* is performed in *array order*, and *atomically*, that is, the operations are performed either as a complete unit, or not at all. The behavior of the system call if not all operations can be performed immediately depends on the presence of the **IPC_NOWAIT** flag in the individual *sem_flg* fields, as noted below.

Each operation is performed on the *sem_num*−th semaphore of the semaphore set, where the first semaphore of the set is numbered 0. There are three types of operation, distinguished by the value of *sem_op*.

If *sem_op* is a positive integer, the operation adds this value to the semaphore value (*semval*). Furthermore, if **SEM_UNDO** is specified for this operation, the system subtracts the value *sem_op* from the semaphore adjustment (*semadj*) value for this semaphore. This operation can always proceed—it never forces a thread to wait. The calling process must have alter permission on the semaphore set.

If *sem_op* is zero, the process must have read permission on the semaphore set. This is a "wait-for-zero" operation: if *semval* is zero, the operation can immediately proceed. Otherwise, if **IPC_NOWAIT** is specified in *sem_flg*, **semop**() fails with *errno* set to **EAGAIN** (and none of the operations in *sops* is performed). Otherwise, *semzcnt* (the count of threads waiting until this semaphore's value becomes zero) is incremented by one and the thread sleeps until one of the following occurs:

- *semval* becomes 0, at which time the value of *semzcnt* is decremented.
- The semaphore set is removed: **semop**() fails, with *errno* set to **EIDRM**.
- The calling thread catches a signal: the value of *semzcnt* is decremented and **semop**() fails, with *errno* set to **EINTR**.

If *sem_op* is less than zero, the process must have alter permission on the semaphore set. If *semval* is greater than or equal to the absolute value of *sem_op*, the operation can proceed immediately: the absolute value of *sem_op* is subtracted from *semval*, and, if **SEM_UNDO** is specified for this operation,

the system adds the absolute value of *sem_op* to the semaphore adjustment (*semadj*) value for this semaphore. If the absolute value of *sem_op* is greater than *semval*, and **IPC_NOWAIT** is specified in *sem_flg*, **semop**() fails, with *errno* set to **EAGAIN** (and none of the operations in *sops* is performed). Otherwise, *semncnt* (the counter of threads waiting for this semaphore's value to increase) is incremented by one and the thread sleeps until one of the following occurs:

- *semval* becomes greater than or equal to the absolute value of *sem_op*: the operation now proceeds, as described above.
- The semaphore set is removed from the system: **semop**() fails, with *errno* set to **EIDRM**.
- The calling thread catches a signal: the value of *semncnt* is decremented and **semop**() fails, with *errno* set to **EINTR**.

On successful completion, the *sempid* value for each semaphore specified in the array pointed to by *sops* is set to the caller's process ID. In addition, the *sem_otime* is set to the current time.

semtimedop()

semtimedop() behaves identically to **semop**() except that in those cases where the calling thread would sleep, the duration of that sleep is limited by the amount of elapsed time specified by the *timespec* structure whose address is passed in the *timeout* argument. (This sleep interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the interval may overrun by a small amount.) If the specified time limit has been reached, **semtimedop**() fails with *errno* set to **EAGAIN** (and none of the operations in *sops* is performed). If the *timeout* argument is NULL, then **semtimedop**() behaves exactly like **semop**().

Note that if **semtimedop**() is interrupted by a signal, causing the call to fail with the error **EINTR**, the contents of *timeout* are left unchanged.

RETURN VALUE

On success, **semop**() and **semtimedop**() return 0. On failure, they return −1, and set *errno* to indicate the error.

ERRORS

E2BIG The argument *nsops* is greater than **SEMOPM**, the maximum number of operations allowed per system call.

EACCES
The calling process does not have the permissions required to perform the specified semaphore operations, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EAGAIN
An operation could not proceed immediately and either **IPC_NOWAIT** was specified in *sem_flg* or the time limit specified in *timeout* expired.

EFAULT
An address specified in either the *sops* or the *timeout* argument isn't accessible.

EFBIG
For some operation the value of *sem_num* is less than 0 or greater than or equal to the number of semaphores in the set.

EIDRM
The semaphore set was removed.

EINTR
While blocked in this system call, the thread caught a signal; see *signal*(7).

EINVAL
The semaphore set doesn't exist, or *semid* is less than zero, or *nsops* has a nonpositive value.

ENOMEM
The *sem_flg* of some operation specified **SEM_UNDO** and the system does not have enough memory to allocate the undo structure.

ERANGE
For some operation *sem_op*+*semval* is greater than **SEMVMX**, the implementation dependent maximum value for *semval*.

STANDARDS

POSIX.1-2008.

VERSIONS

Linux 2.5.52 (backported into Linux 2.4.22), glibc 2.3.3. POSIX.1-2001, SVr4.

NOTES

The *sem_undo* structures of a process aren't inherited by the child produced by *fork*(2), but they are inherited across an *execve*(2) system call.

semop() is never automatically restarted after being interrupted by a signal handler, regardless of the setting of the **SA_RESTART** flag when establishing a signal handler.

A semaphore adjustment (*semadj*) value is a per-process, per-semaphore integer that is the negated sum of all operations performed on a semaphore specifying the **SEM_UNDO** flag. Each process has a list of *semadj* values—one value for each semaphore on which it has operated using **SEM_UNDO**. When a process terminates, each of its per-semaphore *semadj* values is added to the corresponding semaphore, thus undoing the effect of that process's operations on the semaphore (but see BUGS below). When a semaphore's value is directly set using the **SETVAL** or **SETALL** request to *semctl*(2), the corresponding *semadj* values in all processes are cleared. The *clone*(2) **CLONE_SYSVSEM** flag allows more than one process to share a *semadj* list; see *clone*(2) for details.

The *semval*, *sempid*, *semzcnt*, and *semnct* values for a semaphore can all be retrieved using appropriate *semctl*(2) calls.

Semaphore limits

The following limits on semaphore set resources affect the **semop**() call:

SEMOPM
: Maximum number of operations allowed for one **semop**() call. Before Linux 3.19, the default value for this limit was 32. Since Linux 3.19, the default value is 500. On Linux, this limit can be read and modified via the third field of */proc/sys/kernel/sem*. *Note*: this limit should not be raised above 1000, because of the risk of that **semop**() fails due to kernel memory fragmentation when allocating memory to copy the *sops* array.

SEMVMX
: Maximum allowable value for *semval*: implementation dependent (32767).

The implementation has no intrinsic limits for the adjust on exit maximum value (**SEMAEM**), the system wide maximum number of undo structures (**SEMMNU**) and the per-process maximum number of undo entries system parameters.

BUGS

When a process terminates, its set of associated *semadj* structures is used to undo the effect of all of the semaphore operations it performed with the **SEM_UNDO** flag. This raises a difficulty: if one (or more) of these semaphore adjustments would result in an attempt to decrease a semaphore's value below zero, what should an implementation do? One possible approach would be to block until all the semaphore adjustments could be performed. This is however undesirable since it could force process termination to block for arbitrarily long periods. Another possibility is that such semaphore adjustments could be ignored altogether (somewhat analogously to failing when **IPC_NOWAIT** is specified for a semaphore operation). Linux adopts a third approach: decreasing the semaphore value as far as possible (i.e., to zero) and allowing process termination to proceed immediately.

In Linux 2.6.x, x <= 10, there is a bug that in some circumstances prevents a thread that is waiting for a semaphore value to become zero from being woken up when the value does actually become zero. This bug is fixed in Linux 2.6.11.

EXAMPLES

The following code segment uses **semop**() to atomically wait for the value of semaphore 0 to become zero, and then increment the semaphore value by one.

```
struct sembuf sops[2];
int semid;

/* Code to set semid omitted */
```

```
sops[0].sem_num = 0;        /* Operate on semaphore 0 */
sops[0].sem_op = 0;         /* Wait for value to equal 0 */
sops[0].sem_flg = 0;

sops[1].sem_num = 0;        /* Operate on semaphore 0 */
sops[1].sem_op = 1;         /* Increment value by one */
sops[1].sem_flg = 0;

if (semop(semid, sops, 2) == -1) {
    perror("semop");
    exit(EXIT_FAILURE);
}
```

A further example of the use of **semop**() can be found in *shmop*(2).

SEE ALSO

clone(2), *semctl*(2), *semget*(2), *sigaction*(2), *capabilities*(7), *sem_overview*(7), *sysvipc*(7), *time*(7)

NAME

send, sendto, sendmsg − send a message on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

ssize_t send(int *sockfd***, const void** *buf***[.***len***], size_t** *len***, int** *flags***);**
ssize_t sendto(int *sockfd***, const void** *buf***[.***len***], size_t** *len***, int** *flags***,**
const struct sockaddr ***dest_addr*, socklen_t** *addrlen***);**
ssize_t sendmsg(int *sockfd***, const struct msghdr ***msg***, int** *flags***);**

DESCRIPTION

The system calls **send**(), **sendto**(), and **sendmsg**() are used to transmit a message to another socket.

The **send**() call may be used only when the socket is in a *connected* state (so that the intended recipient is known). The only difference between **send**() and *write*(2) is the presence of *flags*. With a zero *flags* argument, **send**() is equivalent to *write*(2). Also, the following call

```
send(sockfd, buf, len, flags);
```

is equivalent to

```
sendto(sockfd, buf, len, flags, NULL, 0);
```

The argument *sockfd* is the file descriptor of the sending socket.

If **sendto**() is used on a connection-mode (**SOCK_STREAM**, **SOCK_SEQPACKET**) socket, the arguments *dest_addr* and *addrlen* are ignored (and the error **EISCONN** may be returned when they are not NULL and 0), and the error **ENOTCONN** is returned when the socket was not actually connected. Otherwise, the address of the target is given by *dest_addr* with *addrlen* specifying its size. For **sendmsg**(), the address of the target is given by *msg.msg_name*, with *msg.msg_namelen* specifying its size.

For **send**() and **sendto**(), the message is found in *buf* and has length *len*. For **sendmsg**(), the message is pointed to by the elements of the array *msg.msg_iov*. The **sendmsg**() call also allows sending ancillary data (also known as control information).

If the message is too long to pass atomically through the underlying protocol, the error **EMSGSIZE** is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send**(). Locally detected errors are indicated by a return value of −1.

When the message does not fit into the send buffer of the socket, **send**() normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode it would fail with the error **EAGAIN** or **EWOULDBLOCK** in this case. The *select*(2) call may be used to determine when it is possible to send more data.

The flags argument

The *flags* argument is the bitwise OR of zero or more of the following flags.

MSG_CONFIRM (since Linux 2.3.15)
: Tell the link layer that forward progress happened: you got a successful reply from the other side. If the link layer doesn't get this it will regularly reprobe the neighbor (e.g., via a unicast ARP). Valid only on **SOCK_DGRAM** and **SOCK_RAW** sockets and currently implemented only for IPv4 and IPv6. See *arp*(7) for details.

MSG_DONTROUTE
: Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is usually used only by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.

MSG_DONTWAIT (since Linux 2.2)
: Enables nonblocking operation; if the operation would block, **EAGAIN** or **EWOULDBLOCK** is returned. This provides similar behavior to setting the **O_NONBLOCK** flag (via the *fcntl*(2) **F_SETFL** operation), but differs in that **MSG_DONTWAIT** is a per-call option,

whereas **O_NONBLOCK** is a setting on the open file description (see *open*(2)), which will affect all threads in the calling process as well as other processes that hold file descriptors referring to the same open file description.

MSG_EOR (since Linux 2.2)
Terminates a record (when this notion is supported, as for sockets of type **SOCK_SEQPACKET**).

MSG_MORE (since Linux 2.4.4)
The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the **TCP_CORK** socket option (see *tcp*(7)), with the difference that this flag can be set on a per-call basis.

Since Linux 2.6, this flag is also supported for UDP sockets, and informs the kernel to package all of the data sent in calls with this flag set into a single datagram which is transmitted only when a call is performed that does not specify this flag. (See also the **UDP_CORK** socket option described in *udp*(7).)

MSG_NOSIGNAL (since Linux 2.2)
Don't generate a **SIGPIPE** signal if the peer on a stream-oriented socket has closed the connection. The **EPIPE** error is still returned. This provides similar behavior to using *sigaction*(2) to ignore **SIGPIPE**, but, whereas **MSG_NOSIGNAL** is a per-call feature, ignoring **SIGPIPE** sets a process attribute that affects all threads in the process.

MSG_OOB
Sends *out-of-band* data on sockets that support this notion (e.g., of type **SOCK_STREAM**); the underlying protocol must also support *out-of-band* data.

MSG_FASTOPEN (since Linux 3.7)
Attempts TCP Fast Open (RFC7413) and sends data in the SYN like a combination of *connect*(2) and *write*(2), by performing an implicit *connect*(2) operation. It blocks until the data is buffered and the handshake has completed. For a non-blocking socket, it returns the number of bytes buffered and sent in the SYN packet. If the cookie is not available locally, it returns **EINPROGRESS**, and sends a SYN with a Fast Open cookie request automatically. The caller needs to write the data again when the socket is connected. On errors, it sets the same *errno* as *connect*(2) if the handshake fails. This flag requires enabling TCP Fast Open client support on sysctl *net.ipv4.tcp_fastopen*.

Refer to **TCP_FASTOPEN_CONNECT** socket option in *tcp*(7) for an alternative approach.

sendmsg()
The definition of the *msghdr* structure employed by **sendmsg**() is as follows:

```
struct msghdr {
    void         *msg_name;       /* Optional address */
    socklen_t     msg_namelen;    /* Size of address */
    struct iovec *msg_iov;        /* Scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* Ancillary data, see below */
    size_t        msg_controllen; /* Ancillary data buffer len */
    int           msg_flags;      /* Flags (unused) */
};
```

The *msg_name* field is used on an unconnected socket to specify the target address for a datagram. It points to a buffer containing the address; the *msg_namelen* field should be set to the size of the address. For a connected socket, these fields should be specified as NULL and 0, respectively.

The *msg_iov* and *msg_iovlen* fields specify scatter-gather locations, as for *writev*(2).

You may send control information (ancillary data) using the *msg_control* and *msg_controllen* members. The maximum control buffer length the kernel can process is limited per socket by the value in */proc/sys/net/core/optmem_max*; see *socket*(7). For further information on the use of ancillary data in various socket domains, see *unix*(7) and *ip*(7).

The *msg_flags* field is ignored.

RETURN VALUE

On success, these calls return the number of bytes sent. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their respective manual pages.

EACCES

(For UNIX domain sockets, which are identified by pathname) Write permission is denied on the destination socket file, or search permission is denied for one of the directories the path prefix. (See *path_resolution*(7).)

(For UDP sockets) An attempt was made to send to a network/broadcast address as though it was a unicast address.

EAGAIN or **EWOULDBLOCK**

The socket is marked nonblocking and the requested operation would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EAGAIN

(Internet domain datagram sockets) The socket referred to by *sockfd* had not previously been bound to an address and, upon attempting to bind it to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of */proc/sys/net/ipv4/ip_local_port_range* in *ip*(7).

EALREADY

Another Fast Open is in progress.

EBADF

sockfd is not a valid open file descriptor.

ECONNRESET

Connection reset by peer.

EDESTADDRREQ

The socket is not connection-mode, and no peer address is set.

EFAULT

An invalid user space address was specified for an argument.

EINTR

A signal occurred before any data was transmitted; see *signal*(7).

EINVAL

Invalid argument passed.

EISCONN

The connection-mode socket was connected already but a recipient was specified. (Now either this error is returned, or the recipient specification is ignored.)

EMSGSIZE

The socket type requires that message be sent atomically, and the size of the message to be sent made this impossible.

ENOBUFS

The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion. (Normally, this does not occur in Linux. Packets are just silently dropped when a device queue overflows.)

ENOMEM

No memory available.

ENOTCONN

The socket is not connected, and no target has been given.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

EOPNOTSUPP
Some bit in the *flags* argument is inappropriate for the socket type.

EPIPE The local end has been shut down on a connection oriented socket. In this case, the process will also receive a **SIGPIPE** unless **MSG_NOSIGNAL** is set.

VERSIONS

According to POSIX.1-2001, the *msg_controllen* field of the *msghdr* structure should be typed as *socklen_t*, and the *msg_iovlen* field should be typed as *int*, but glibc currently types both as *size_t*.

STANDARDS

POSIX.1-2008.

MSG_CONFIRM is a Linux extension.

HISTORY

4.4BSD, SVr4, POSIX.1-2001. (first appeared in 4.2BSD).

POSIX.1-2001 describes only the **MSG_OOB** and **MSG_EOR** flags. POSIX.1-2008 adds a specification of **MSG_NOSIGNAL**.

NOTES

See *sendmmsg*(2) for information about a Linux-specific system call that can be used to transmit multiple datagrams in a single call.

BUGS

Linux may return **EPIPE** instead of **ENOTCONN**.

EXAMPLES

An example of the use of **sendto**() is shown in *getaddrinfo*(3).

SEE ALSO

fcntl(2), *getsockopt*(2), *recv*(2), *select*(2), *sendfile*(2), *sendmmsg*(2), *shutdown*(2), *socket*(2), *write*(2), *cmsg*(3), *ip*(7), *ipv6*(7), *socket*(7), *tcp*(7), *udp*(7), *unix*(7)

NAME

sendfile − transfer data between file descriptors

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sendfile.h>

ssize_t sendfile(int *out_fd***, int** *in_fd***, off_t *_Nullable** *offset***,**
size_t *count***);**

DESCRIPTION

sendfile() copies data between one file descriptor and another. Because this copying is done within the kernel, **sendfile**() is more efficient than the combination of *read*(2) and *write*(2), which would require transferring data to and from user space.

in_fd should be a file descriptor opened for reading and *out_fd* should be a descriptor opened for writing.

If *offset* is not NULL, then it points to a variable holding the file offset from which **sendfile**() will start reading data from *in_fd*. When **sendfile**() returns, this variable will be set to the offset of the byte following the last byte that was read. If *offset* is not NULL, then **sendfile**() does not modify the file offset of *in_fd*; otherwise the file offset is adjusted to reflect the number of bytes read from *in_fd*.

If *offset* is NULL, then data will be read from *in_fd* starting at the file offset, and the file offset will be updated by the call.

count is the number of bytes to copy between the file descriptors.

The *in_fd* argument must correspond to a file which supports *mmap*(2)-like operations (i.e., it cannot be a socket). Except since Linux 5.12 and if *out_fd* is a pipe, in which case **sendfile**() desugars to a *splice*(2) and its restrictions apply.

Before Linux 2.6.33, *out_fd* must refer to a socket. Since Linux 2.6.33 it can be any file. If it's seekable, then **sendfile**() changes the file offset appropriately.

RETURN VALUE

If the transfer was successful, the number of bytes written to *out_fd* is returned. Note that a successful call to **sendfile**() may write fewer bytes than requested; the caller should be prepared to retry the call if there were unsent bytes. See also NOTES.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
Nonblocking I/O has been selected using **O_NONBLOCK** and the write would block.

EBADF
The input file was not opened for reading or the output file was not opened for writing.

EFAULT
Bad address.

EINVAL
Descriptor is not valid or locked, or an *mmap*(2)-like operation is not available for *in_fd*, or *count* is negative.

EINVAL
out_fd has the **O_APPEND** flag set. This is not currently supported by **sendfile**().

EIO Unspecified error while reading from *in_fd*.

ENOMEM
Insufficient memory to read from *in_fd*.

EOVERFLOW
count is too large, the operation would result in exceeding the maximum size of either the input file or the output file.

ESPIPE
offset is not NULL but the input file is not seekable.

VERSIONS

Other UNIX systems implement **sendfile**() with different semantics and prototypes. It should not be used in portable programs.

STANDARDS

None.

HISTORY

Linux 2.2, glibc 2.1.

In Linux 2.4 and earlier, *out_fd* could also refer to a regular file; this possibility went away in the Linux 2.6.x kernel series, but was restored in Linux 2.6.33.

The original Linux **sendfile**() system call was not designed to handle large file offsets. Consequently, Linux 2.4 added **sendfile64**(), with a wider type for the *offset* argument. The glibc **sendfile**() wrapper function transparently deals with the kernel differences.

NOTES

sendfile() will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

If you plan to use **sendfile**() for sending files to a TCP socket, but need to send some header data in front of the file contents, you will find it useful to employ the **TCP_CORK** option, described in *tcp*(7), to minimize the number of packets and to tune performance.

Applications may wish to fall back to *read*(2) and *write*(2) in the case where **sendfile**() fails with **EINVAL** or **ENOSYS**.

If *out_fd* refers to a socket or pipe with zero-copy support, callers must ensure the transferred portions of the file referred to by *in_fd* remain unmodified until the reader on the other end of *out_fd* has consumed the transferred data.

The Linux-specific *splice*(2) call supports transferring data between arbitrary file descriptors provided one (or both) of them is a pipe.

SEE ALSO

copy_file_range(2), *mmap*(2), *open*(2), *socket*(2), *splice*(2)

NAME

sendmmsg − send multiple messages on a socket

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/socket.h>

int sendmmsg(int sockfd, struct mmsghdr *msgvec, unsigned int vlen,
             int flags);
```

DESCRIPTION

The **sendmmsg**() system call is an extension of *sendmsg*(2) that allows the caller to transmit multiple messages on a socket using a single system call. (This has performance benefits for some applications.)

The *sockfd* argument is the file descriptor of the socket on which data is to be transmitted.

The *msgvec* argument is a pointer to an array of *mmsghdr* structures. The size of this array is specified in *vlen*.

The *mmsghdr* structure is defined in <*sys/socket.h*> as:

```
struct mmsghdr {
    struct msghdr msg_hdr;  /* Message header */
    unsigned int  msg_len;  /* Number of bytes transmitted */
};
```

The *msg_hdr* field is a *msghdr* structure, as described in *sendmsg*(2). The *msg_len* field is used to return the number of bytes sent from the message in *msg_hdr* (i.e., the same as the return value from a single *sendmsg*(2) call).

The *flags* argument contains flags ORed together. The flags are the same as for *sendmsg*(2).

A blocking **sendmmsg**() call blocks until *vlen* messages have been sent. A nonblocking call sends as many messages as possible (up to the limit specified by *vlen*) and returns immediately.

On return from **sendmmsg**(), the *msg_len* fields of successive elements of *msgvec* are updated to contain the number of bytes transmitted from the corresponding *msg_hdr*. The return value of the call indicates the number of elements of *msgvec* that have been updated.

RETURN VALUE

On success, **sendmmsg**() returns the number of messages sent from *msgvec*; if this is less than *vlen*, the caller can retry with a further **sendmmsg**() call to send the remaining messages.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Errors are as for *sendmsg*(2). An error is returned only if no datagrams could be sent. See also BUGS.

STANDARDS

Linux.

HISTORY

Linux 3.0, glibc 2.14.

NOTES

The value specified in *vlen* is capped to **UIO_MAXIOV** (1024).

BUGS

If an error occurs after at least one message has been sent, the call succeeds, and returns the number of messages sent. The error code is lost. The caller can retry the transmission, starting at the first failed message, but there is no guarantee that, if an error is returned, it will be the same as the one that was lost on the previous call.

EXAMPLES

The example below uses **sendmmsg**() to send *onetwo* and *three* in two distinct UDP datagrams using one system call. The contents of the first datagram originates from a pair of buffers.

```
#define _GNU_SOURCE
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>

int
main(void)
{
    int                 retval;
    int                 sockfd;
    struct iovec        msg1[2], msg2;
    struct mmsghdr      msg[2];
    struct sockaddr_in  addr;

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    addr.sin_port = htons(1234);
    if (connect(sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        perror("connect()");
        exit(EXIT_FAILURE);
    }

    memset(msg1, 0, sizeof(msg1));
    msg1[0].iov_base = "one";
    msg1[0].iov_len = 3;
    msg1[1].iov_base = "two";
    msg1[1].iov_len = 3;

    memset(&msg2, 0, sizeof(msg2));
    msg2.iov_base = "three";
    msg2.iov_len = 5;

    memset(msg, 0, sizeof(msg));
    msg[0].msg_hdr.msg_iov = msg1;
    msg[0].msg_hdr.msg_iovlen = 2;

    msg[1].msg_hdr.msg_iov = &msg2;
    msg[1].msg_hdr.msg_iovlen = 1;

    retval = sendmmsg(sockfd, msg, 2, 0);
    if (retval == -1)
        perror("sendmmsg()");
    else
        printf("%d messages sent\n", retval);

    exit(0);
}
```

SEE ALSO

recvmmsg(2), *sendmsg*(2), *socket*(2), *socket*(7)

NAME

set_mempolicy − set default NUMA memory policy for a thread and its children

LIBRARY

NUMA (Non-Uniform Memory Access) policy library (*libnuma*, *−lnuma*)

SYNOPSIS

#include <numaif.h>

long set_mempolicy(int *mode***, const unsigned long ***_nodemask_**,**
unsigned long *maxnode***);**

DESCRIPTION

set_mempolicy() sets the NUMA memory policy of the calling thread, which consists of a policy mode and zero or more nodes, to the values specified by the *mode*, *nodemask*, and *maxnode* arguments.

A NUMA machine has different memory controllers with different distances to specific CPUs. The memory policy defines from which node memory is allocated for the thread.

This system call defines the default policy for the thread. The thread policy governs allocation of pages in the process's address space outside of memory ranges controlled by a more specific policy set by *mbind*(2). The thread default policy also controls allocation of any pages for memory-mapped files mapped using the *mmap*(2) call with the **MAP_PRIVATE** flag and that are only read (loaded) from by the thread and of memory-mapped files mapped using the *mmap*(2) call with the **MAP_SHARED** flag, regardless of the access type. The policy is applied only when a new page is allocated for the thread. For anonymous memory this is when the page is first touched by the thread.

The *mode* argument must specify one of **MPOL_DEFAULT**, **MPOL_BIND**, **MPOL_INTERLEAVE**, **MPOL_PREFERRED**, or **MPOL_LOCAL** (which are described in detail below). All modes except **MPOL_DEFAULT** require the caller to specify the node or nodes to which the mode applies, via the *nodemask* argument.

The *mode* argument may also include an optional *mode flag*. The supported *mode flags* are:

MPOL_F_NUMA_BALANCING (since Linux 5.12)
When *mode* is **MPOL_BIND**, enable the kernel NUMA balancing for the task if it is supported by the kernel. If the flag isn't supported by the kernel, or is used with *mode* other than **MPOL_BIND**, −1 is returned and *errno* is set to **EINVAL**.

MPOL_F_RELATIVE_NODES (since Linux 2.6.26)
A nonempty *nodemask* specifies node IDs that are relative to the set of node IDs allowed by the process's current cpuset.

MPOL_F_STATIC_NODES (since Linux 2.6.26)
A nonempty *nodemask* specifies physical node IDs. Linux will not remap the *nodemask* when the process moves to a different cpuset context, nor when the set of nodes allowed by the process's current cpuset context changes.

nodemask points to a bit mask of node IDs that contains up to *maxnode* bits. The bit mask size is rounded to the next multiple of *sizeof(unsigned long)*, but the kernel will use bits only up to *maxnode*. A NULL value of *nodemask* or a *maxnode* value of zero specifies the empty set of nodes. If the value of *maxnode* is zero, the *nodemask* argument is ignored.

Where a *nodemask* is required, it must contain at least one node that is on-line, allowed by the process's current cpuset context, (unless the **MPOL_F_STATIC_NODES** mode flag is specified), and contains memory. If the **MPOL_F_STATIC_NODES** is set in *mode* and a required *nodemask* contains no nodes that are allowed by the process's current cpuset context, the memory policy reverts to *local allocation*. This effectively overrides the specified policy until the process's cpuset context includes one or more of the nodes specified by *nodemask*.

The *mode* argument must include one of the following values:

MPOL_DEFAULT
This mode specifies that any nondefault thread memory policy be removed, so that the memory policy "falls back" to the system default policy. The system default policy is "local allocation"—that is, allocate memory on the node of the CPU that triggered the allocation. *nodemask* must be specified as NULL. If the "local node" contains no free memory, the system will attempt to allocate memory from a "near by" node.

MPOL_BIND
This mode defines a strict policy that restricts memory allocation to the nodes specified in *nodemask*. If *nodemask* specifies more than one node, page allocations will come from the node with the lowest numeric node ID first, until that node contains no free memory. Allocations will then come from the node with the next highest node ID specified in *nodemask* and so forth, until none of the specified nodes contain free memory. Pages will not be allocated from any node not specified in the *nodemask*.

MPOL_INTERLEAVE
This mode interleaves page allocations across the nodes specified in *nodemask* in numeric node ID order. This optimizes for bandwidth instead of latency by spreading out pages and memory accesses to those pages across multiple nodes. However, accesses to a single page will still be limited to the memory bandwidth of a single node.

MPOL_PREFERRED
This mode sets the preferred node for allocation. The kernel will try to allocate pages from this node first and fall back to "near by" nodes if the preferred node is low on free memory. If *nodemask* specifies more than one node ID, the first node in the mask will be selected as the preferred node. If the *nodemask* and *maxnode* arguments specify the empty set, then the policy specifies "local allocation" (like the system default policy discussed above).

MPOL_LOCAL (since Linux 3.8)
This mode specifies "local allocation"; the memory is allocated on the node of the CPU that triggered the allocation (the "local node"). The *nodemask* and *maxnode* arguments must specify the empty set. If the "local node" is low on free memory, the kernel will try to allocate memory from other nodes. The kernel will allocate memory from the "local node" whenever memory for this node is available. If the "local node" is not allowed by the process's current cpuset context, the kernel will try to allocate memory from other nodes. The kernel will allocate memory from the "local node" whenever it becomes allowed by the process's current cpuset context.

The thread memory policy is preserved across an *execve*(2), and is inherited by child threads created using *fork*(2) or *clone*(2).

RETURN VALUE
On success, **set_mempolicy**() returns 0; on error, −1 is returned and *errno* is set to indicate the error.

ERRORS
EFAULT
Part of all of the memory range specified by *nodemask* and *maxnode* points outside your accessible address space.

EINVAL
mode is invalid. Or, *mode* is **MPOL_DEFAULT** and *nodemask* is nonempty, or *mode* is **MPOL_BIND** or **MPOL_INTERLEAVE** and *nodemask* is empty. Or, *maxnode* specifies more than a page worth of bits. Or, *nodemask* specifies one or more node IDs that are greater than the maximum supported node ID. Or, none of the node IDs specified by *nodemask* are on-line and allowed by the process's current cpuset context, or none of the specified nodes contain memory. Or, the *mode* argument specified both **MPOL_F_STATIC_NODES** and **MPOL_F_RELATIVE_NODES**. Or, the **MPOL_F_NUMA_BALANCING** isn't supported by the kernel, or is used with *mode* other than **MPOL_BIND**.

ENOMEM
Insufficient kernel memory was available.

STANDARDS
Linux.

HISTORY
Linux 2.6.7.

NOTES
Memory policy is not remembered if the page is swapped out. When such a page is paged back in, it will use the policy of the thread or memory range that is in effect at the time the page is allocated.

For information on library support, see *numa*(7).

SEE ALSO

get_mempolicy(2), *getcpu*(2), *mbind*(2), *mmap*(2), *numa*(3), *cpuset*(7), *numa*(7), *numactl*(8)

NAME

get_thread_area, set_thread_area − manipulate thread-local storage information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

#if defined __i386__ || defined __x86_64__
# include <asm/ldt.h>         /* Definition of struct user_desc */

int syscall(SYS_get_thread_area, struct user_desc *u_info);
int syscall(SYS_set_thread_area, struct user_desc *u_info);

#elif defined __m68k__

int syscall(SYS_get_thread_area);
int syscall(SYS_set_thread_area, unsigned long tp);

#elif defined __mips__ || defined __csky__

int syscall(SYS_set_thread_area, unsigned long addr);

#endif
```

Note: glibc provides no wrappers for these system calls, necessitating the use of *syscall*(2).

DESCRIPTION

These calls provide architecture-specific support for a thread-local storage implementation. At the moment, **set_thread_area**() is available on m68k, MIPS, C-SKY, and x86 (both 32-bit and 64-bit variants); **get_thread_area**() is available on m68k and x86.

On m68k, MIPS and C-SKY, **set_thread_area**() allows storing an arbitrary pointer (provided in the **tp** argument on m68k and in the **addr** argument on MIPS and C-SKY) in the kernel data structure associated with the calling thread; this pointer can later be retrieved using **get_thread_area**() (see also NOTES for information regarding obtaining the thread pointer on MIPS).

On x86, Linux dedicates three global descriptor table (GDT) entries for thread-local storage. For more information about the GDT, see the Intel Software Developer's Manual or the AMD Architecture Programming Manual.

Both of these system calls take an argument that is a pointer to a structure of the following type:

```
struct user_desc {
    unsigned int  entry_number;
    unsigned int  base_addr;
    unsigned int  limit;
    unsigned int  seg_32bit:1;
    unsigned int  contents:2;
    unsigned int  read_exec_only:1;
    unsigned int  limit_in_pages:1;
    unsigned int  seg_not_present:1;
    unsigned int  useable:1;
#ifdef __x86_64__
    unsigned int  lm:1;
#endif
};
```

get_thread_area() reads the GDT entry indicated by *u_info->entry_number* and fills in the rest of the fields in *u_info*.

set_thread_area() sets a TLS entry in the GDT.

The TLS array entry set by **set_thread_area**() corresponds to the value of *u_info->entry_number* passed in by the user. If this value is in bounds, **set_thread_area**() writes the TLS descriptor pointed to by *u_info* into the thread's TLS array.

When **set_thread_area**() is passed an *entry_number* of −1, it searches for a free TLS entry. If

set_thread_area() finds a free TLS entry, the value of *u_info->entry_number* is set upon return to show which entry was changed.

A *user_desc* is considered "empty" if *read_exec_only* and *seg_not_present* are set to 1 and all of the other fields are 0. If an "empty" descriptor is passed to **set_thread_area**(), the corresponding TLS entry will be cleared. See BUGS for additional details.

Since Linux 3.19, **set_thread_area**() cannot be used to write non-present segments, 16-bit segments, or code segments, although clearing a segment is still acceptable.

RETURN VALUE

On x86, these system calls return 0 on success, and −1 on failure, with *errno* set to indicate the error.

On C-SKY, MIPS and m68k, **set_thread_area**() always returns 0. On m68k, **get_thread_area**() returns the thread area pointer value (previously set via *set_thread_area*())

ERRORS

EFAULT
u_info is an invalid pointer.

EINVAL
u_info->entry_number is out of bounds.

ENOSYS
get_thread_area() or **set_thread_area**() was invoked as a 64-bit system call.

ESRCH
(**set_thread_area**()) A free TLS entry could not be located.

STANDARDS

Linux.

HISTORY

set_thread_area()
Linux 2.5.29.

get_thread_area()
Linux 2.5.32.

NOTES

These system calls are generally intended for use only by threading libraries.

arch_prctl(2) can interfere with **set_thread_area**() on x86. See *arch_prctl*(2) for more details. This is not normally a problem, as *arch_prctl*(2) is normally used only by 64-bit programs.

On MIPS, the current value of the thread area pointer can be obtained using the instruction:

```
rdhwr dest, $29
```

This instruction traps and is handled by kernel.

BUGS

On 64-bit kernels before Linux 3.19, one of the padding bits in *user_desc*, if set, would prevent the descriptor from being considered empty (see *modify_ldt*(2)). As a result, the only reliable way to clear a TLS entry is to use *memset*(3) to zero the entire *user_desc* structure, including padding bits, and then to set the *read_exec_only* and *seg_not_present* bits. On Linux 3.19, a *user_desc* consisting entirely of zeros except for *entry_number* will also be interpreted as a request to clear a TLS entry, but this behaved differently on older kernels.

Prior to Linux 3.19, the DS and ES segment registers must not reference TLS entries.

SEE ALSO

arch_prctl(2), *modify_ldt*(2), *ptrace*(2) (**PTRACE_GET_THREAD_AREA** and **PTRACE_SET_THREAD_AREA**)

NAME

set_tid_address − set pointer to thread ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

pid_t syscall(SYS_set_tid_address, int *****tidptr);

Note: glibc provides no wrapper for **set_tid_address**(), necessitating the use of *syscall*(2).

DESCRIPTION

For each thread, the kernel maintains two attributes (addresses) called *set_child_tid* and *clear_child_tid*. These two attributes contain the value NULL by default.

set_child_tid
: If a thread is started using *clone*(2) with the **CLONE_CHILD_SETTID** flag, *set_child_tid* is set to the value passed in the *ctid* argument of that system call.

  When *set_child_tid* is set, the very first thing the new thread does is to write its thread ID at this address.

clear_child_tid
: If a thread is started using *clone*(2) with the **CLONE_CHILD_CLEARTID** flag, *clear_child_tid* is set to the value passed in the *ctid* argument of that system call.

The system call **set_tid_address**() sets the *clear_child_tid* value for the calling thread to *tidptr*.

When a thread whose *clear_child_tid* is not NULL terminates, then, if the thread is sharing memory with other threads, then 0 is written at the address specified in *clear_child_tid* and the kernel performs the following operation:

```
futex(clear_child_tid, FUTEX_WAKE, 1, NULL, NULL, 0);
```

The effect of this operation is to wake a single thread that is performing a futex wait on the memory location. Errors from the futex wake operation are ignored.

RETURN VALUE

set_tid_address() always returns the caller's thread ID.

ERRORS

set_tid_address() always succeeds.

STANDARDS

Linux.

HISTORY

Linux 2.5.48.

Details as given here are valid since Linux 2.5.49.

SEE ALSO

clone(2), *futex*(2), *gettid*(2)

NAME

seteuid, setegid − set effective user or group ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

seteuid(), **setegid**():
_POSIX_C_SOURCE >= 200112L
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

seteuid() sets the effective user ID of the calling process. Unprivileged processes may only set the effective user ID to the real user ID, the effective user ID or the saved set-user-ID.

Precisely the same holds for **setegid**() with "group" instead of "user".

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **seteuid**() can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **seteuid**().

ERRORS

EINVAL
The target user or group ID is not valid in this user namespace.

EPERM
In the case of **seteuid**(): the calling process is not privileged (does not have the **CAP_SETUID** capability in its user namespace) and *euid* does not match the current real user ID, current effective user ID, or current saved set-user-ID.

In the case of **setegid**(): the calling process is not privileged (does not have the **CAP_SETGID** capability in its user namespace) and *egid* does not match the current real group ID, current effective group ID, or current saved set-group-ID.

VERSIONS

Setting the effective user (group) ID to the saved set-user-ID (saved set-group-ID) is possible since Linux 1.1.37 (1.1.38). On an arbitrary system one should check **_POSIX_SAVED_IDS**.

Under glibc 2.0, **seteuid(***euid***)** is equivalent to **setreuid(−1,** *euid***)** and hence may change the saved set-user-ID. Under glibc 2.1 and later, it is equivalent to **setresuid(−1,** *euid***, −1)** and hence does not change the saved set-user-ID. Analogous remarks hold for **setegid**(), with the difference that the change in implementation from **setregid(−1,** *egid***)** to **setresgid(−1,** *egid***, −1)** occurred in glibc 2.2 or 2.3 (depending on the hardware architecture).

According to POSIX.1, **seteuid**() (**setegid**()) need not permit *euid* (*egid*) to be the same value as the current effective user (group) ID, and some implementations do not permit this.

C library/kernel differences

On Linux, **seteuid**() and **setegid**() are implemented as library functions that call, respectively, *setresuid*(2) and *setresgid*(2).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

SEE ALSO

geteuid(2), *setresuid*(2), *setreuid*(2), *setuid*(2), *capabilities*(7), *credentials*(7), *user_namespaces*(7)

NAME

setfsgid − set group identity used for filesystem checks

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/fsuid.h>

[[deprecated]] int setfsgid(gid_t *fsgid***);**

DESCRIPTION

On Linux, a process has both a filesystem group ID and an effective group ID. The (Linux-specific) filesystem group ID is used for permissions checking when accessing filesystem objects, while the effective group ID is used for some other kinds of permissions checks (see *credentials*(7)).

Normally, the value of the process's filesystem group ID is the same as the value of its effective group ID. This is so, because whenever a process's effective group ID is changed, the kernel also changes the filesystem group ID to be the same as the new value of the effective group ID. A process can cause the value of its filesystem group ID to diverge from its effective group ID by using **setfsgid**() to change its filesystem group ID to the value given in *fsgid*.

setfsgid() will succeed only if the caller is the superuser or if *fsgid* matches either the caller's real group ID, effective group ID, saved set-group-ID, or current the filesystem user ID.

RETURN VALUE

On both success and failure, this call returns the previous filesystem group ID of the caller.

STANDARDS

Linux.

HISTORY

Linux 1.2.

C library/kernel differences

In glibc 2.15 and earlier, when the wrapper for this system call determines that the argument can't be passed to the kernel without integer truncation (because the kernel is old and does not support 32-bit group IDs), it will return −1 and set *errno* to **EINVAL** without attempting the system call.

NOTES

The filesystem group ID concept and the **setfsgid**() system call were invented for historical reasons that are no longer applicable on modern Linux kernels. See *setfsuid*(2) for a discussion of why the use of both *setfsuid*(2) and **setfsgid**() is nowadays unneeded.

The original Linux **setfsgid**() system call supported only 16-bit group IDs. Subsequently, Linux 2.4 added **setfsgid32**() supporting 32-bit IDs. The glibc **setfsgid**() wrapper function transparently deals with the variation across kernel versions.

BUGS

No error indications of any kind are returned to the caller, and the fact that both successful and unsuccessful calls return the same value makes it impossible to directly determine whether the call succeeded or failed. Instead, the caller must resort to looking at the return value from a further call such as *setfsgid(−1)* (which will always fail), in order to determine if a preceding call to **setfsgid**() changed the filesystem group ID. At the very least, **EPERM** should be returned when the call fails (because the caller lacks the **CAP_SETGID** capability).

SEE ALSO

kill(2), *setfsuid*(2), *capabilities*(7), *credentials*(7)

NAME

setfsuid − set user identity used for filesystem checks

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/fsuid.h>

[[deprecated]] int setfsuid(uid_t *fsuid***);**

DESCRIPTION

On Linux, a process has both a filesystem user ID and an effective user ID. The (Linux-specific) filesystem user ID is used for permissions checking when accessing filesystem objects, while the effective user ID is used for various other kinds of permissions checks (see *credentials*(7)).

Normally, the value of the process's filesystem user ID is the same as the value of its effective user ID. This is so, because whenever a process's effective user ID is changed, the kernel also changes the filesystem user ID to be the same as the new value of the effective user ID. A process can cause the value of its filesystem user ID to diverge from its effective user ID by using **setfsuid**() to change its filesystem user ID to the value given in *fsuid*.

Explicit calls to **setfsuid**() and *setfsgid*(2) are (were) usually used only by programs such as the Linux NFS server that need to change what user and group ID is used for file access without a corresponding change in the real and effective user and group IDs. A change in the normal user IDs for a program such as the NFS server is (was) a security hole that can expose it to unwanted signals. (However, this issue is historical; see below.)

setfsuid() will succeed only if the caller is the superuser or if *fsuid* matches either the caller's real user ID, effective user ID, saved set-user-ID, or current filesystem user ID.

RETURN VALUE

On both success and failure, this call returns the previous filesystem user ID of the caller.

STANDARDS

Linux.

HISTORY

Linux 1.2.

At the time when this system call was introduced, one process could send a signal to another process with the same effective user ID. This meant that if a privileged process changed its effective user ID for the purpose of file permission checking, then it could become vulnerable to receiving signals sent by another (unprivileged) process with the same user ID. The filesystem user ID attribute was thus added to allow a process to change its user ID for the purposes of file permission checking without at the same time becoming vulnerable to receiving unwanted signals. Since Linux 2.0, signal permission handling is different (see *kill*(2)), with the result that a process can change its effective user ID without being vulnerable to receiving signals from unwanted processes. Thus, **setfsuid**() is nowadays unneeded and should be avoided in new applications (likewise for *setfsgid*(2)).

The original Linux **setfsuid**() system call supported only 16-bit user IDs. Subsequently, Linux 2.4 added **setfsuid32**() supporting 32-bit IDs. The glibc **setfsuid**() wrapper function transparently deals with the variation across kernel versions.

C library/kernel differences

In glibc 2.15 and earlier, when the wrapper for this system call determines that the argument can't be passed to the kernel without integer truncation (because the kernel is old and does not support 32-bit user IDs), it will return −1 and set *errno* to **EINVAL** without attempting the system call.

BUGS

No error indications of any kind are returned to the caller, and the fact that both successful and unsuccessful calls return the same value makes it impossible to directly determine whether the call succeeded or failed. Instead, the caller must resort to looking at the return value from a further call such as *setfsuid(−1)* (which will always fail), in order to determine if a preceding call to **setfsuid**() changed the filesystem user ID. At the very least, **EPERM** should be returned when the call fails (because the caller lacks the **CAP_SETUID** capability).

SEE ALSO

kill(2), *setfsgid*(2), *capabilities*(7), *credentials*(7)

NAME

setgid − set group identity

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int setgid(gid_t *gid***);**

DESCRIPTION

setgid() sets the effective group ID of the calling process. If the calling process is privileged (more precisely: has the **CAP_SETGID** capability in its user namespace), the real GID and saved set-group-ID are also set.

Under Linux, **setgid**() is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-group-ID program that is not set-user-ID-root to drop all of its group privileges, do some un-privileged work, and then reengage the original effective group ID in a secure manner.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL

The group ID specified in *gid* is not valid in this user namespace.

EPERM

The calling process is not privileged (does not have the **CAP_SETGID** capability in its user namespace), and *gid* does not match the real group ID or saved set-group-ID of the calling process.

VERSIONS

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for *setgid*()) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see *nptl*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

The original Linux **setgid**() system call supported only 16-bit group IDs. Subsequently, Linux 2.4 added **setgid32**() supporting 32-bit IDs. The glibc **setgid**() wrapper function transparently deals with the variation across kernel versions.

SEE ALSO

getgid(2), *setegid*(2), *setregid*(2), *capabilities*(7), *credentials*(7), *user_namespaces*(7)

NAME

setns − reassociate thread with a namespace

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sched.h>

int setns(int fd, int nstype);
```

DESCRIPTION

The **setns**() system call allows the calling thread to move into different namespaces. The *fd* argument is one of the following:

- a file descriptor referring to one of the magic links in a */proc/*pid*/ns/* directory (or a bind mount to such a link);
- a PID file descriptor (see *pidfd_open*(2)).

The *nstype* argument is interpreted differently in each case.

fd refers to a */proc/*pid*/ns/* link

If *fd* refers to a */proc/*pid*/ns/* link, then **setns**() reassociates the calling thread with the namespace associated with that link, subject to any constraints imposed by the *nstype* argument. In this usage, each call to **setns**() changes just one of the caller's namespace memberships.

The *nstype* argument specifies which type of namespace the calling thread may be reassociated with. This argument can have *one* of the following values:

0 Allow any type of namespace to be joined.

CLONE_NEWCGROUP (since Linux 4.6)
fd must refer to a cgroup namespace.

CLONE_NEWIPC (since Linux 3.0)
fd must refer to an IPC namespace.

CLONE_NEWNET (since Linux 3.0)
fd must refer to a network namespace.

CLONE_NEWNS (since Linux 3.8)
fd must refer to a mount namespace.

CLONE_NEWPID (since Linux 3.8)
fd must refer to a descendant PID namespace.

CLONE_NEWTIME (since Linux 5.8)
fd must refer to a time namespace.

CLONE_NEWUSER (since Linux 3.8)
fd must refer to a user namespace.

CLONE_NEWUTS (since Linux 3.0)
fd must refer to a UTS namespace.

Specifying *nstype* as 0 suffices if the caller knows (or does not care) what type of namespace is referred to by *fd*. Specifying a nonzero value for *nstype* is useful if the caller does not know what type of namespace is referred to by *fd* and wants to ensure that the namespace is of a particular type. (The caller might not know the type of the namespace referred to by *fd* if the file descriptor was opened by another process and, for example, passed to the caller via a UNIX domain socket.)

fd is a PID file descriptor

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from *pidfd_open*(2) or *clone*(2). In this usage, **setns**() atomically moves the calling thread into one or more of the same namespaces as the thread referred to by *fd*.

The *nstype* argument is a bit mask specified by ORing together *one or more* of the **CLONE_NEW*** namespace constants listed above. The caller is moved into each of the target thread's namespaces that is specified in *nstype*; the caller's memberships in the remaining namespaces are left unchanged.

For example, the following code would move the caller into the same user, network, and UTS namespaces as PID 1234, but would leave the caller's other namespace memberships unchanged:

```
int fd = pidfd_open(1234, 0);
setns(fd, CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWUTS);
```

Details for specific namespace types

Note the following details and restrictions when reassociating with specific namespace types:

User namespaces

A process reassociating itself with a user namespace must have the **CAP_SYS_ADMIN** capability in the target user namespace. (This necessarily implies that it is only possible to join a descendant user namespace.) Upon successfully joining a user namespace, a process is granted all capabilities in that namespace, regardless of its user and group IDs.

A multithreaded process may not change user namespace with **setns**().

It is not permitted to use **setns**() to reenter the caller's current user namespace. This prevents a caller that has dropped capabilities from regaining those capabilities via a call to **setns**().

For security reasons, a process can't join a new user namespace if it is sharing filesystem-related attributes (the attributes whose sharing is controlled by the *clone*(2) **CLONE_FS** flag) with another process.

For further details on user namespaces, see *user_namespaces*(7).

Mount namespaces

Changing the mount namespace requires that the caller possess both **CAP_SYS_CHROOT** and **CAP_SYS_ADMIN** capabilities in its own user namespace and **CAP_SYS_ADMIN** in the user namespace that owns the target mount namespace.

A process can't join a new mount namespace if it is sharing filesystem-related attributes (the attributes whose sharing is controlled by the *clone*(2) **CLONE_FS** flag) with another process.

See *user_namespaces*(7) for details on the interaction of user namespaces and mount namespaces.

PID namespaces

In order to reassociate itself with a new PID namespace, the caller must have the **CAP_SYS_ADMIN** capability both in its own user namespace and in the user namespace that owns the target PID namespace.

Reassociating the PID namespace has somewhat different from other namespace types. Reassociating the calling thread with a PID namespace changes only the PID namespace that subsequently created child processes of the caller will be placed in; it does not change the PID namespace of the caller itself.

Reassociating with a PID namespace is allowed only if the target PID namespace is a descendant (child, grandchild, etc.) of, or is the same as, the current PID namespace of the caller.

For further details on PID namespaces, see *pid_namespaces*(7).

Cgroup namespaces

In order to reassociate itself with a new cgroup namespace, the caller must have the **CAP_SYS_ADMIN** capability both in its own user namespace and in the user namespace that owns the target cgroup namespace.

Using **setns**() to change the caller's cgroup namespace does not change the caller's cgroup memberships.

Network, IPC, time, and UTS namespaces

In order to reassociate itself with a new network, IPC, time, or UTS namespace, the caller must have the **CAP_SYS_ADMIN** capability both in its own user namespace and in the user namespace that owns the target namespace.

RETURN VALUE

On success, **setns**() returns 0. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not a valid file descriptor.

EINVAL
fd refers to a namespace whose type does not match that specified in *nstype*.

EINVAL
There is problem with reassociating the thread with the specified namespace.

EINVAL
The caller tried to join an ancestor (parent, grandparent, and so on) PID namespace.

EINVAL
The caller attempted to join the user namespace in which it is already a member.

EINVAL
The caller shares filesystem (**CLONE_FS**) state (in particular, the root directory) with other processes and tried to join a new user namespace.

EINVAL
The caller is multithreaded and tried to join a new user namespace.

EINVAL
fd is a PID file descriptor and *nstype* is invalid (e.g., it is 0).

ENOMEM
Cannot allocate sufficient memory to change the specified namespace.

EPERM
The calling thread did not have the required capability for this operation.

ESRCH
fd is a PID file descriptor but the process it refers to no longer exists (i.e., it has terminated and been waited on).

STANDARDS

Linux.

VERSIONS

Linux 3.0, glibc 2.14.

NOTES

For further information on the */proc/*pid*/ns/* magic links, see *namespaces*(7).

Not all of the attributes that can be shared when a new thread is created using *clone*(2) can be changed using **setns**().

EXAMPLES

The program below takes two or more arguments. The first argument specifies the pathname of a namespace file in an existing */proc/*pid*/ns/* directory. The remaining arguments specify a command and its arguments. The program opens the namespace file, joins that namespace using **setns**(), and executes the specified command inside that namespace.

The following shell session demonstrates the use of this program (compiled as a binary named *ns_exec*) in conjunction with the **CLONE_NEWUTS** example program in the *clone*(2) man page (complied as a binary named *newuts*).

We begin by executing the example program in *clone*(2) in the background. That program creates a child in a separate UTS namespace. The child changes the hostname in its namespace, and then both processes display the hostnames in their UTS namespaces, so that we can see that they are different.

```
$ su                    # Need privilege for namespace operations
Password:
# ./newuts bizarro &
[1] 3549
clone() returned 3550
uts.nodename in child:  bizarro
uts.nodename in parent: antero
```

```
# uname -n              # Verify hostname in the shell
antero
```

We then run the program shown below, using it to execute a shell. Inside that shell, we verify that the hostname is the one set by the child created by the first program:

```
# ./ns_exec /proc/3550/ns/uts /bin/bash
# uname -n              # Executed in shell started by ns_exec
bizarro
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int fd;

    if (argc < 3) {
        fprintf(stderr, "%s /proc/PID/ns/FILE cmd args...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Get file descriptor for namespace; the file descriptor is opened
       with O_CLOEXEC so as to ensure that it is not inherited by the
       program that is later executed. */

    fd = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (fd == -1)
        err(EXIT_FAILURE, "open");

    if (setns(fd, 0) == -1)       /* Join that namespace */
        err(EXIT_FAILURE, "setns");

    execvp(argv[2], &argv[2]);    /* Execute a command in namespace */
    err(EXIT_FAILURE, "execvp");
}
```

SEE ALSO

nsenter(1), *clone*(2), *fork*(2), *unshare*(2), *vfork*(2), *namespaces*(7), *unix*(7)

NAME

setpgid, getpgid, setpgrp, getpgrp − set/get process group

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);

pid_t getpgrp(void);                    /* POSIX.1 version */
[[deprecated]] pid_t getpgrp(pid_t pid);    /* BSD version */

int setpgrp(void);                      /* System V version */
[[deprecated]] int setpgrp(pid_t pid, pid_t pgid);  /* BSD version */
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpgid():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

setpgrp() (POSIX.1):
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

setpgrp() (BSD), **getpgrp**() (BSD):
[These are available only before glibc 2.19]
_BSD_SOURCE &&
! (_POSIX_SOURCE || _POSIX_C_SOURCE || _XOPEN_SOURCE
|| _GNU_SOURCE || _SVID_SOURCE)

DESCRIPTION

All of these interfaces are available on Linux, and are used for getting and setting the process group ID (PGID) of a process. The preferred, POSIX.1-specified ways of doing this are: *getpgrp*(void), for retrieving the calling process's PGID; and **setpgid**(), for setting a process's PGID.

setpgid() sets the PGID of the process specified by *pid* to *pgid*. If *pid* is zero, then the process ID of the calling process is used. If *pgid* is zero, then the PGID of the process specified by *pid* is made the same as its process ID. If **setpgid**() is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session (see *setsid*(2) and *credentials*(7)). In this case, the *pgid* specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

The POSIX.1 version of **getpgrp**(), which takes no arguments, returns the PGID of the calling process.

getpgid() returns the PGID of the process specified by *pid*. If *pid* is zero, the process ID of the calling process is used. (Retrieving the PGID of a process other than the caller is rarely necessary, and the POSIX.1 **getpgrp**() is preferred for that task.)

The System V-style **setpgrp**(), which takes no arguments, is equivalent to *setpgid(0, 0)*.

The BSD-specific **setpgrp**() call, which takes arguments *pid* and *pgid*, is a wrapper function that calls

```
setpgid(pid, pgid)
```

Since glibc 2.19, the BSD-specific **setpgrp**() function is no longer exposed by <*unistd.h*>; calls should be replaced with the **setpgid**() call shown above.

The BSD-specific **getpgrp**() call, which takes a single *pid* argument, is a wrapper function that calls

```
getpgid(pid)
```

Since glibc 2.19, the BSD-specific **getpgrp**() function is no longer exposed by <*unistd.h*>; calls should be replaced with calls to the POSIX.1 **getpgrp**() which takes no arguments (if the intent is to obtain the caller's PGID), or with the **getpgid**() call shown above.

RETURN VALUE

On success, **setpgid**() and **setpgrp**() return zero. On error, −1 is returned, and *errno* is set to indicate the error.

The POSIX.1 **getpgrp**() always returns the PGID of the caller.

getpgid(), and the BSD-specific **getpgrp**() return a process group on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: An attempt was made to change the process group ID of one of the children of the calling process and the child had already performed an *execve*(2) (**setpgid**(), *setpgrp*())

EINVAL
: *pgid* is less than 0 (**setpgid**(), *setpgrp*())

EPERM
: An attempt was made to move a process into a process group in a different session, or to change the process group ID of one of the children of the calling process and the child was in a different session, or to change the process group ID of a session leader (**setpgid**(), *setpgrp*())

EPERM
: The target process group does not exist. (**setpgid**(), *setpgrp*())

ESRCH
: For **getpgid**(): *pid* does not match any process. For **setpgid**(): *pid* is not the calling process and not a child of the calling process.

STANDARDS

getpgid()
setpgid()
getpgrp() (no args)
setpgrp() (no args)
: POSIX.1-2008 (but see HISTORY).

setpgrp() (2 args)
getpgrp() (1 arg)
: None.

HISTORY

getpgid()
setpgid()
getpgrp() (no args)
: POSIX.1-2001.

setpgrp() (no args)
: POSIX.1-2001. POSIX.1-2008 marks it as obsolete.

setpgrp() (2 args)
getpgrp() (1 arg)
: 4.2BSD.

NOTES

A child created via *fork*(2) inherits its parent's process group ID. The PGID is preserved across an *execve*(2).

Each process group is a member of a session and each process is a member of the session of which its process group is a member. (See *credentials*(7).)

A session can have a controlling terminal. At any time, one (and only one) of the process groups in the session can be the foreground process group for the terminal; the remaining process groups are in the background. If a signal is generated from the terminal (e.g., typing the interrupt key to generate **SIGINT**), that signal is sent to the foreground process group. (See *termios*(3) for a description of the characters that generate signals.) Only the foreground process group may *read*(2) from the terminal; if a background process group tries to *read*(2) from the terminal, then the group is sent a **SIGTTIN** signal, which suspends it. The *tcgetpgrp*(3) and *tcsetpgrp*(3) functions are used to get/set the foreground process group of the controlling terminal.

The **setpgid**() and **getpgrp**() calls are used by programs such as *bash*(1) to create process groups in order to implement shell job control.

If the termination of a process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in the newly orphaned process group. An orphaned process group is one in which the parent of every member of process group is either itself also a member of the process group or is a member of a process group in a different session (see also *credentials*(7)).

SEE ALSO

getuid(2), *setsid*(2), *tcgetpgrp*(3), *tcsetpgrp*(3), *termios*(3), *credentials*(7)

NAME

setresuid, setresgid − set real, effective, and saved user or group ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>

int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

DESCRIPTION

setresuid() sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

An unprivileged process may change its real UID, effective UID, and saved set-user-ID, each to one of: the current real UID, the current effective UID, or the current saved set-user-ID.

A privileged process (on Linux, one having the **CAP_SETUID** capability) may set its real UID, effective UID, and saved set-user-ID to arbitrary values.

If one of the arguments equals −1, the corresponding value is not changed.

Regardless of what changes are made to the real UID, effective UID, and saved set-user-ID, the filesystem UID is always set to the same value as the (possibly new) effective UID.

Completely analogously, **setresgid**() sets the real GID, effective GID, and saved set-group-ID of the calling process (and always modifies the filesystem GID to be the same as the effective GID), with the same restrictions for unprivileged processes.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **setresuid**() can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **setresuid**().

ERRORS

EAGAIN
: The call would change the caller's real UID (i.e., *ruid* does not match the caller's real UID), but there was a temporary failure allocating the necessary kernel data structures.

EAGAIN
: *ruid* does not match the caller's real UID and this call would bring the number of processes belonging to the real user ID *ruid* over the caller's **RLIMIT_NPROC** resource limit. Since Linux 3.1, this error case no longer occurs (but robust applications should check for this error); see the description of **EAGAIN** in *execve*(2).

EINVAL
: One or more of the target user or group IDs is not valid in this user namespace.

EPERM
: The calling process is not privileged (did not have the necessary capability in its user namespace) and tried to change the IDs to values that are not permitted. For **setresuid**(), the necessary capability is **CAP_SETUID**; for **setresgid**(), it is **CAP_SETGID**.

VERSIONS

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including those for **setresuid**() and *setresgid*()) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see *nptl*(7).

STANDARDS

None.

HISTORY

Linux 2.1.44, glibc 2.3.2. HP-UX, FreeBSD.

The original Linux **setresuid**() and **setresgid**() system calls supported only 16-bit user and group IDs. Subsequently, Linux 2.4 added **setresuid32**() and **setresgid32**(), supporting 32-bit IDs. The glibc **setresuid**() and **setresgid**() wrapper functions transparently deal with the variations across kernel versions.

SEE ALSO

getresuid(2), *getuid*(2), *setfsgid*(2), *setfsuid*(2), *setreuid*(2), *setuid*(2), *capabilities*(7), *credentials*(7), *user_namespaces*(7)

NAME

setreuid, setregid − set real and/or effective user or group ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int setreuid(uid_t *ruid***, uid_t** *euid***);**
int setregid(gid_t *rgid***, gid_t** *egid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setreuid(), **setregid**():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

setreuid() sets real and effective user IDs of the calling process.

Supplying a value of −1 for either the real or effective user ID forces the system to leave that ID unchanged.

Unprivileged processes may only set the effective user ID to the real user ID, the effective user ID, or the saved set-user-ID.

Unprivileged users may only set the real user ID to the real user ID or the effective user ID.

If the real user ID is set (i.e., *ruid* is not −1) or the effective user ID is set to a value not equal to the previous real user ID, the saved set-user-ID will be set to the new effective user ID.

Completely analogously, **setregid**() sets real and effective group ID's of the calling process, and all of the above holds with "group" instead of "user".

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **setreuid**() can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **setreuid**().

ERRORS

EAGAIN
The call would change the caller's real UID (i.e., *ruid* does not match the caller's real UID), but there was a temporary failure allocating the necessary kernel data structures.

EAGAIN
ruid does not match the caller's real UID and this call would bring the number of processes belonging to the real user ID *ruid* over the caller's **RLIMIT_NPROC** resource limit. Since Linux 3.1, this error case no longer occurs (but robust applications should check for this error); see the description of **EAGAIN** in *execve*(2).

EINVAL
One or more of the target user or group IDs is not valid in this user namespace.

EPERM
The calling process is not privileged (on Linux, does not have the necessary capability in its user namespace: **CAP_SETUID** in the case of **setreuid**(), or **CAP_SETGID** in the case of *setregid*()) and a change other than (i) swapping the effective user (group) ID with the real user (group) ID, or (ii) setting one to the value of the other or (iii) setting the effective user (group) ID to the value of the saved set-user-ID (saved set-group-ID) was specified.

VERSIONS

POSIX.1 does not specify all of the UID changes that Linux permits for an unprivileged process. For **setreuid**(), the effective user ID can be made the same as the real user ID or the saved set-user-ID, and it is unspecified whether unprivileged processes may set the real user ID to the real user ID, the effective user ID, or the saved set-user-ID. For **setregid**(), the real group ID can be changed to the value of the saved set-group-ID, and the effective group ID can be changed to the value of the real group ID or

the saved set-group-ID. The precise details of what ID changes are permitted vary across implementations.

POSIX.1 makes no specification about the effect of these calls on the saved set-user-ID and saved set-group-ID.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD (first appeared in 4.2BSD).

Setting the effective user (group) ID to the saved set-user-ID (saved set-group-ID) is possible since Linux 1.1.37 (1.1.38).

The original Linux **setreuid**() and **setregid**() system calls supported only 16-bit user and group IDs. Subsequently, Linux 2.4 added **setreuid32**() and **setregid32**(), supporting 32-bit IDs. The glibc **setreuid**() and **setregid**() wrapper functions transparently deal with the variations across kernel versions.

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including those for **setreuid**() and *setregid*()) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see *nptl*(7).

SEE ALSO

getgid(2), *getuid*(2), *seteuid*(2), *setgid*(2), *setresuid*(2), *setuid*(2), *capabilities*(7), *credentials*(7), *user_namespaces*(7)

NAME

setsid − creates a session and sets the process group ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

pid_t setsid(void);

DESCRIPTION

setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session (i.e., its session ID is made the same as its process ID). The calling process also becomes the process group leader of a new process group in the session (i.e., its process group ID is made the same as its process ID).

The calling process will be the only process in the new process group and in the new session.

Initially, the new session has no controlling terminal. For details of how a session acquires a controlling terminal, see *credentials*(7).

RETURN VALUE

On success, the (new) session ID of the calling process is returned. On error, *(pid_t) −1* is returned, and *errno* is set to indicate the error.

ERRORS

EPERM

The process group ID of any process equals the PID of the calling process. Thus, in particular, **setsid**() fails if the calling process is already a process group leader.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

NOTES

A child created via *fork*(2) inherits its parent's session ID. The session ID is preserved across an *execve*(2).

A process group leader is a process whose process group ID equals its PID. Disallowing a process group leader from calling **setsid**() prevents the possibility that a process group leader places itself in a new session while other processes in the process group remain in the original session; such a scenario would break the strict two-level hierarchy of sessions and process groups. In order to be sure that **setsid**() will succeed, call *fork*(2) and have the parent *_exit*(2), while the child (which by definition can't be a process group leader) calls **setsid**().

If a session has a controlling terminal, and the **CLOCAL** flag for that terminal is not set, and a terminal hangup occurs, then the session leader is sent a **SIGHUP** signal.

If a process that is a session leader terminates, then a **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal.

SEE ALSO

setsid(1), *getsid*(2), *setpgid*(2), *setpgrp*(2), *tcgetsid*(3), *credentials*(7), *sched*(7)

NAME

setuid − set user identity

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int setuid(uid_t *uid***);**

DESCRIPTION

setuid() sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the **CAP_SETUID** capability in its user namespace), the real UID and saved set-user-ID are also set.

Under Linux, **setuid**() is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some un-privileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken: **setuid**() checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to *uid*. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a set-user-ID-root program wishing to temporarily drop root privileges, assume the identity of an unprivileged user, and then regain root privileges afterward cannot use **setuid**(). You can accomplish this with *seteuid*(2).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note: there are cases where **setuid**() can fail even when the caller is UID 0; it is a grave security error to omit checking for a failure return from **setuid**().

ERRORS

EAGAIN

The call would change the caller's real UID (i.e., *uid* does not match the caller's real UID), but there was a temporary failure allocating the necessary kernel data structures.

EAGAIN

uid does not match the real user ID of the caller and this call would bring the number of processes belonging to the real user ID *uid* over the caller's **RLIMIT_NPROC** resource limit. Since Linux 3.1, this error case no longer occurs (but robust applications should check for this error); see the description of **EAGAIN** in *execve*(2).

EINVAL

The user ID specified in *uid* is not valid in this user namespace.

EPERM

The user is not privileged (Linux: does not have the **CAP_SETUID** capability in its user namespace) and *uid* does not match the real UID or saved set-user-ID of the calling process.

VERSIONS

C library/kernel differences

At the kernel level, user IDs and group IDs are a per-thread attribute. However, POSIX requires that all threads in a process share the same credentials. The NPTL threading implementation handles the POSIX requirements by providing wrapper functions for the various system calls that change process UIDs and GIDs. These wrapper functions (including the one for *setuid*()) employ a signal-based technique to ensure that when one thread changes credentials, all of the other threads in the process also change their credentials. For details, see *nptl*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

Not quite compatible with the 4.4BSD call, which sets all of the real, saved, and effective user IDs.

The original Linux **setuid**() system call supported only 16-bit user IDs. Subsequently, Linux 2.4 added **setuid32**() supporting 32-bit IDs. The glibc **setuid**() wrapper function transparently deals with the variation across kernel versions.

NOTES

Linux has the concept of the filesystem user ID, normally equal to the effective user ID. The **setuid**() call also sets the filesystem user ID of the calling process. See *setfsuid*(2).

If *uid* is different from the old effective UID, the process will be forbidden from leaving core dumps.

SEE ALSO

getuid(2), *seteuid*(2), *setfsuid*(2), *setreuid*(2), *capabilities*(7), *credentials*(7), *user_namespaces*(7)

NAME

setup − setup devices and filesystems, mount root filesystem

LIBRARY

Standard C library (*libc*, −*lc*)

SYNOPSIS

#include <unistd.h>

[[deprecated]] int setup(void);

DESCRIPTION

setup() is called once from within *linux/init/main.c*. It calls initialization functions for devices and filesystems configured into the kernel and then mounts the root filesystem.

No user process may call **setup**(). Any user process, even a process with superuser permission, will receive **EPERM**.

RETURN VALUE

setup() always returns −1 for a user process.

ERRORS

EPERM

Always, for a user process.

STANDARDS

Linux.

VERSIONS

Removed in Linux 2.1.121.

The calling sequence varied: at some times **setup**() has had a single argument *void *BIOS* and at other times a single argument *int magic*.

NAME

setxattr, lsetxattr, fsetxattr − set an extended attribute value

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/xattr.h>

int setxattr(const char *path, const char *name,
        const void value[.size], size_t size, int flags);
int lsetxattr(const char *path, const char *name,
        const void value[.size], size_t size, int flags);
int fsetxattr(int fd, const char *name,
        const void value[.size], size_t size, int flags);
```

DESCRIPTION

Extended attributes are *name*:*value* pairs associated with inodes (files, directories, symbolic links, etc.). They are extensions to the normal attributes which are associated with all inodes in the system (i.e., the *stat*(2) data). A complete overview of extended attributes concepts can be found in *xattr*(7).

setxattr() sets the *value* of the extended attribute identified by *name* and associated with the given *path* in the filesystem. The *size* argument specifies the size (in bytes) of *value*; a zero-length value is permitted.

lsetxattr() is identical to **setxattr**(), except in the case of a symbolic link, where the extended attribute is set on the link itself, not the file that it refers to.

fsetxattr() is identical to **setxattr**(), only the extended attribute is set on the open file referred to by *fd* (as returned by *open*(2)) in place of *path*.

An extended attribute name is a null-terminated string. The *name* includes a namespace prefix; there may be several, disjoint namespaces associated with an individual inode. The *value* of an extended attribute is a chunk of arbitrary textual or binary data of specified length.

By default (i.e., *flags* is zero), the extended attribute will be created if it does not exist, or the value will be replaced if the attribute already exists. To modify these semantics, one of the following values can be specified in *flags*:

XATTR_CREATE
: Perform a pure create, which fails if the named attribute exists already.

XATTR_REPLACE
: Perform a pure replace operation, which fails if the named attribute does not already exist.

RETURN VALUE

On success, zero is returned. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EDQUOT
: Disk quota limits meant that there is insufficient space remaining to store the extended attribute.

EEXIST
: **XATTR_CREATE** was specified, and the attribute exists already.

ENODATA
: **XATTR_REPLACE** was specified, and the attribute does not exist.

ENOSPC
: There is insufficient space remaining to store the extended attribute.

ENOTSUP
: The namespace prefix of *name* is not valid.

ENOTSUP
: Extended attributes are not supported by the filesystem, or are disabled,

EPERM
The file is marked immutable or append-only. (See *ioctl_iflags*(2).)

In addition, the errors documented in *stat*(2) can also occur.

ERANGE
The size of *name* or *value* exceeds a filesystem-specific limit.

STANDARDS
Linux.

HISTORY
Linux 2.4, glibc 2.3.

SEE ALSO
getfattr(1), *setfattr*(1), *getxattr*(2), *listxattr*(2), *open*(2), *removexattr*(2), *stat*(2), *symlink*(7), *xattr*(7)

NAME

sgetmask, ssetmask − manipulation of signal mask (obsolete)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

[[deprecated]] long syscall(SYS_sgetmask, void);
[[deprecated]] long syscall(SYS_ssetmask, long newmask);
```

DESCRIPTION

These system calls are obsolete. *Do not use them*; use *sigprocmask*(2) instead.

sgetmask() returns the signal mask of the calling process.

ssetmask() sets the signal mask of the calling process to the value given in *newmask*. The previous signal mask is returned.

The signal masks dealt with by these two system calls are plain bit masks (unlike the *sigset_t* used by *sigprocmask*(2)); use *sigmask*(3) to create and inspect these masks.

RETURN VALUE

sgetmask() always successfully returns the signal mask. **ssetmask**() always succeeds, and returns the previous signal mask.

ERRORS

These system calls always succeed.

STANDARDS

Linux.

HISTORY

Since Linux 3.16, support for these system calls is optional, depending on whether the kernel was built with the **CONFIG_SGETMASK_SYSCALL** option.

NOTES

These system calls are unaware of signal numbers greater than 31 (i.e., real-time signals).

These system calls do not exist on x86-64.

It is not possible to block **SIGSTOP** or **SIGKILL**.

SEE ALSO

sigprocmask(2), *signal*(7)

NAME

shmctl − System V shared memory control

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/shm.h>

int shmctl(int *shmid***, int** *op***, struct shmid_ds *****buf***);**

DESCRIPTION

shmctl() performs the control operation specified by *op* on the System V shared memory segment whose identifier is given in *shmid*.

The *buf* argument is a pointer to a *shmid_ds* structure, defined in <*sys/shm.h*> as follows:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Creation time/time of last
                                    modification via shmctl() */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

The fields of the *shmid_ds* structure are as follows:

shm_perm This is an *ipc_perm* structure (see below) that specifies the access permissions on the shared memory segment.

shm_segsz Size in bytes of the shared memory segment.

shm_atime Time of the last *shmat*(2) system call that attached this segment.

shm_dtime Time of the last *shmdt*(2) system call that detached tgis segment.

shm_ctime Time of creation of segment or time of the last **shmctl**() **IPC_SET** operation.

shm_cpid ID of the process that created the shared memory segment.

shm_lpid ID of the last process that executed a *shmat*(2) or *shmdt*(2) system call on this segment.

shm_nattch Number of processes that have this segment attached.

The *ipc_perm* structure is defined as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
    gid_t          cgid;     /* Effective GID of creator */
    unsigned short mode;     /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
    unsigned short __seq;    /* Sequence number */
};
```

The least significant 9 bits of the *mode* field of the *ipc_perm* structure define the access permissions for the shared memory segment. The permission bits are as follows:

0400 Read by user
0200 Write by user
0040 Read by group
0020 Write by group

0004 Read by others
0002 Write by others

Bits 0100, 0010, and 0001 (the execute bits) are unused by the system. (It is not necessary to have execute permission on a segment in order to perform a *shmat*(2) call with the **SHM_EXEC** flag.)

Valid values for *op* are:

IPC_STAT
Copy information from the kernel data structure associated with *shmid* into the *shmid_ds* structure pointed to by *buf*. The caller must have read permission on the shared memory segment.

IPC_SET
Write the values of some members of the *shmid_ds* structure pointed to by *buf* to the kernel data structure associated with this shared memory segment, updating also its *shm_ctime* member.

The following fields are updated: *shm_perm.uid*, *shm_perm.gid*, and (the least significant 9 bits of) *shm_perm.mode*.

The effective UID of the calling process must match the owner (*shm_perm.uid*) or creator (*shm_perm.cuid*) of the shared memory segment, or the caller must be privileged.

IPC_RMID
Mark the segment to be destroyed. The segment will actually be destroyed only after the last process detaches it (i.e., when the *shm_nattch* member of the associated structure *shmid_ds* is zero). The caller must be the owner or creator of the segment, or be privileged. The *buf* argument is ignored.

If a segment has been marked for destruction, then the (nonstandard) **SHM_DEST** flag of the *shm_perm.mode* field in the associated data structure retrieved by **IPC_STAT** will be set.

The caller *must* ensure that a segment is eventually destroyed; otherwise its pages that were faulted in will remain in memory or swap.

See also the description of */proc/sys/kernel/shm_rmid_forced* in *proc*(5).

IPC_INFO (Linux-specific)
Return information about system-wide shared memory limits and parameters in the structure pointed to by *buf*. This structure is of type *shminfo* (thus, a cast is required), defined in <*sys/shm.h*> if the **_GNU_SOURCE** feature test macro is defined:

```
struct shminfo {
    unsigned long shmmax; /* Maximum segment size */
    unsigned long shmmin; /* Minimum segment size;
                             always 1 */
    unsigned long shmmni; /* Maximum number of segments */
    unsigned long shmseg; /* Maximum number of segments
                             that a process can attach;
                             unused within kernel */
    unsigned long shmall; /* Maximum number of pages of
                             shared memory, system-wide */
};
```

The *shmmni*, *shmmax*, and *shmall* settings can be changed via */proc* files of the same name; see *proc*(5) for details.

SHM_INFO (Linux-specific)
Return a *shm_info* structure whose fields contain information about system resources consumed by shared memory. This structure is defined in <*sys/shm.h*> if the **_GNU_SOURCE** feature test macro is defined:

```
struct shm_info {
    int           used_ids; /* # of currently existing
                               segments */
    unsigned long shm_tot;  /* Total number of shared
                               memory pages */
```

```
            unsigned long shm_rss;  /* # of resident shared
                                       memory pages */
            unsigned long shm_swp;  /* # of swapped shared
                                       memory pages */
            unsigned long swap_attempts;
                                    /* Unused since Linux 2.4 */
            unsigned long swap_successes;
                                    /* Unused since Linux 2.4 */
        };
```

SHM_STAT (Linux-specific)
: Return a *shmid_ds* structure as for **IPC_STAT**. However, the *shmid* argument is not a segment identifier, but instead an index into the kernel's internal array that maintains information about all shared memory segments on the system.

SHM_STAT_ANY (Linux-specific, since Linux 4.17)
: Return a *shmid_ds* structure as for **SHM_STAT**. However, *shm_perm.mode* is not checked for read access for *shmid*, meaning that any user can employ this operation (just as any user may read */proc/sysvipc/shm* to obtain the same information).

The caller can prevent or allow swapping of a shared memory segment with the following *op* values:

SHM_LOCK (Linux-specific)
: Prevent swapping of the shared memory segment. The caller must fault in any pages that are required to be present after locking is enabled. If a segment has been locked, then the (non-standard) **SHM_LOCKED** flag of the *shm_perm.mode* field in the associated data structure retrieved by **IPC_STAT** will be set.

SHM_UNLOCK (Linux-specific)
: Unlock the segment, allowing it to be swapped out.

Before Linux 2.6.10, only a privileged process could employ **SHM_LOCK** and **SHM_UNLOCK**. Since Linux 2.6.10, an unprivileged process can employ these operations if its effective UID matches the owner or creator UID of the segment, and (for **SHM_LOCK**) the amount of memory to be locked falls within the **RLIMIT_MEMLOCK** resource limit (see *setrlimit*(2)).

RETURN VALUE

A successful **IPC_INFO** or **SHM_INFO** operation returns the index of the highest used entry in the kernel's internal array recording information about all shared memory segments. (This information can be used with repeated **SHM_STAT** or **SHM_STAT_ANY** operations to obtain information about all shared memory segments on the system.) A successful **SHM_STAT** operation returns the identifier of the shared memory segment whose index was given in *shmid*. Other operations return 0 on success.

On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: **IPC_STAT** or **SHM_STAT** is requested and *shm_perm.mode* does not allow read access for *shmid*, and the calling process does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EFAULT
: The argument *op* has value **IPC_SET** or **IPC_STAT** but the address pointed to by *buf* isn't accessible.

EIDRM
: *shmid* points to a removed identifier.

EINVAL
: *shmid* is not a valid identifier, or *op* is not a valid operation. Or: for a **SHM_STAT** or **SHM_STAT_ANY** operation, the index value specified in *shmid* referred to an array slot that is currently unused.

ENOMEM
: (Since Linux 2.6.9), **SHM_LOCK** was specified and the size of the to-be-locked segment would mean that the total bytes in locked shared memory segments would exceed the limit for

the real user ID of the calling process. This limit is defined by the **RLIMIT_MEMLOCK** soft resource limit (see *setrlimit*(2)).

EOVERFLOW
IPC_STAT is attempted, and the GID or UID value is too large to be stored in the structure pointed to by *buf*.

EPERM
IPC_SET or **IPC_RMID** is attempted, and the effective user ID of the calling process is not that of the creator (found in *shm_perm.cuid*), or the owner (found in *shm_perm.uid*), and the process was not privileged (Linux: did not have the **CAP_SYS_ADMIN** capability).

Or (before Linux 2.6.9), **SHM_LOCK** or **SHM_UNLOCK** was specified, but the process was not privileged (Linux: did not have the **CAP_IPC_LOCK** capability). (Since Linux 2.6.9, this error can also occur if the **RLIMIT_MEMLOCK** is 0 and the caller is not privileged.)

VERSIONS

Linux permits a process to attach (**shmat**(2)) a shared memory segment that has already been marked for deletion using *shmctl(IPC_RMID)*. This feature is not available on other UNIX implementations; portable applications should avoid relying on it.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

Various fields in a *struct shmid_ds* were typed as *short* under Linux 2.2 and have become *long* under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by an **IPC_64** flag in *op*.)

NOTES

The **IPC_INFO**, **SHM_STAT**, and **SHM_INFO** operations are used by the *ipcs*(1) program to provide information on allocated resources. In the future, these may modified or moved to a */proc* filesystem interface.

SEE ALSO

mlock(2), *setrlimit*(2), *shmget*(2), *shmop*(2), *capabilities*(7), *sysvipc*(7)

NAME

shmget − allocates a System V shared memory segment

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/shm.h>

int shmget(key_t *key***, size_t** *size***, int** *shmflg***);**

DESCRIPTION

shmget() returns the identifier of the System V shared memory segment associated with the value of the argument *key*. It may be used either to obtain the identifier of a previously created shared memory segment (when *shmflg* is zero and *key* does not have the value **IPC_PRIVATE**), or to create a new set.

A new shared memory segment, with size equal to the value of *size* rounded up to a multiple of **PAGE_SIZE**, is created if *key* has the value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no shared memory segment corresponding to *key* exists, and **IPC_CREAT** is specified in *shmflg*.

If *shmflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a shared memory segment already exists for *key*, then **shmget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for *open*(2).)

The value *shmflg* is composed of:

IPC_CREAT
: Create a new segment. If this flag is not used, then **shmget**() will find the segment associated with *key* and check to see if the user has permission to access the segment.

IPC_EXCL
: This flag is used with **IPC_CREAT** to ensure that this call creates the segment. If the segment already exists, the call fails.

SHM_HUGETLB (since Linux 2.6)
: Allocate the segment using "huge" pages. See the Linux kernel source file *Documentation/admin−guide/mm/hugetlbpage.rst* for further information.

SHM_HUGE_2MB
SHM_HUGE_1GB (since Linux 3.8)
: Used in conjunction with **SHM_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that support multiple hugetlb page sizes.

  More generally, the desired huge page size can be configured by encoding the base-2 logarithm of the desired page size in the six bits at the offset **SHM_HUGE_SHIFT**. Thus, the above two constants are defined as:

```
#define SHM_HUGE_2MB    (21 << SHM_HUGE_SHIFT)
#define SHM_HUGE_1GB    (30 << SHM_HUGE_SHIFT)
```

  For some additional details, see the discussion of the similarly named constants in *mmap*(2).

SHM_NORESERVE (since Linux 2.6.15)
: This flag serves the same purpose as the *mmap*(2) **MAP_NORESERVE** flag. Do not reserve swap space for this segment. When swap space is reserved, one has the guarantee that it is possible to modify the segment. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in *proc*(5).

In addition to the above flags, the least significant 9 bits of *shmflg* specify the permissions granted to the owner, group, and others. These bits have the same format, and the same meaning, as the *mode* argument of *open*(2). Presently, execute permissions are not used by the system.

When a new shared memory segment is created, its contents are initialized to zero values, and its associated data structure, *shmid_ds* (see *shmctl*(2)), is initialized as follows:

- *shm_perm.cuid* and *shm_perm.uid* are set to the effective user ID of the calling process.

- *shm_perm.cgid* and *shm_perm.gid* are set to the effective group ID of the calling process.
- The least significant 9 bits of *shm_perm.mode* are set to the least significant 9 bit of *shmflg*.
- *shm_segsz* is set to the value of *size*.
- *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set to 0.
- *shm_ctime* is set to the current time.

If the shared memory segment already exists, the permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

On success, a valid shared memory identifier is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
The user does not have permission to access the shared memory segment, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EEXIST
IPC_CREAT and **IPC_EXCL** were specified in *shmflg*, but a shared memory segment already exists for *key*.

EINVAL
A new segment was to be created and *size* is less than **SHMMIN** or greater than **SHMMAX**.

EINVAL
A segment for the given *key* exists, but *size* is greater than the size of that segment.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
No segment exists for the given *key*, and **IPC_CREAT** was not specified.

ENOMEM
No memory could be allocated for segment overhead.

ENOSPC
All possible shared memory IDs have been taken (**SHMMNI**), or allocating a segment of the requested *size* would cause the system to exceed the system-wide limit on shared memory (**SHMALL**).

EPERM
The **SHM_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability) and is not a member of the *sysctl_hugetlb_shm_group* group; see the description of */proc/sys/vm/sysctl_hugetlb_shm_group* in *proc*(5).

STANDARDS

POSIX.1-2008.

SHM_HUGETLB and **SHM_NORESERVE** are Linux extensions.

HISTORY

POSIX.1-2001, SVr4.

NOTES

IPC_PRIVATE isn't a flag field but a *key_t* type. If this special value is used for *key*, the system call ignores all but the least significant 9 bits of *shmflg* and creates a new shared memory segment.

Shared memory limits

The following limits on shared memory segment resources affect the **shmget**() call:

SHMALL
System-wide limit on the total amount of shared memory, measured in units of the system page size.

On Linux, this limit can be read and modified via */proc/sys/kernel/shmall*. Since Linux 3.16, the default value for this limit is:

```
ULONG_MAX - 2^24
```

The effect of this value (which is suitable for both 32-bit and 64-bit systems) is to impose no limitation on allocations. This value, rather than **ULONG_MAX**, was chosen as the default to prevent some cases where historical applications simply raised the existing limit without first checking its current value. Such applications would cause the value to overflow if the limit was set at **ULONG_MAX**.

From Linux 2.4 up to Linux 3.15, the default value for this limit was:

```
SHMMAX / PAGE_SIZE * (SHMMNI / 16)
```

If **SHMMAX** and **SHMMNI** were not modified, then multiplying the result of this formula by the page size (to get a value in bytes) yielded a value of 8 GB as the limit on the total memory used by all shared memory segments.

SHMMAX
Maximum size in bytes for a shared memory segment.

On Linux, this limit can be read and modified via */proc/sys/kernel/shmmax*. Since Linux 3.16, the default value for this limit is:

```
ULONG_MAX - 2^24
```

The effect of this value (which is suitable for both 32-bit and 64-bit systems) is to impose no limitation on allocations. See the description of **SHMALL** for a discussion of why this default value (rather than **ULONG_MAX**) is used.

From Linux 2.2 up to Linux 3.15, the default value of this limit was 0x2000000 (32 MiB).

Because it is not possible to map just part of a shared memory segment, the amount of virtual memory places another limit on the maximum size of a usable segment: for example, on i386 the largest segments that can be mapped have a size of around 2.8 GB, and on x86-64 the limit is around 127 TB.

SHMMIN
Minimum size in bytes for a shared memory segment: implementation dependent (currently 1 byte, though **PAGE_SIZE** is the effective minimum size).

SHMMNI
System-wide limit on the number of shared memory segments. In Linux 2.2, the default value for this limit was 128; since Linux 2.4, the default value is 4096.

On Linux, this limit can be read and modified via */proc/sys/kernel/shmmni*.

The implementation has no specific limits for the per-process maximum number of shared memory segments (**SHMSEG**).

Linux notes
Until Linux 2.3.30, Linux would return **EIDRM** for a **shmget**() on a shared memory segment scheduled for deletion.

BUGS
The name choice **IPC_PRIVATE** was perhaps unfortunate, **IPC_NEW** would more clearly show its function.

EXAMPLES
See *shmop*(2).

SEE ALSO
memfd_create(2), *shmat*(2), *shmctl*(2), *shmdt*(2), *ftok*(3), *capabilities*(7), *shm_overview*(7), *sysvipc*(7)

NAME

shmat, shmdt − System V shared memory operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/shm.h>

void *shmat(int shmid, const void *_Nullable shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

DESCRIPTION

shmat()

shmat() attaches the System V shared memory segment identified by *shmid* to the address space of the calling process. The attaching address is specified by *shmaddr* with one of the following criteria:

- If *shmaddr* is NULL, the system chooses a suitable (unused) page-aligned address to attach the segment.
- If *shmaddr* isn't NULL and **SHM_RND** is specified in *shmflg*, the attach occurs at the address equal to *shmaddr* rounded down to the nearest multiple of **SHMLBA**.
- Otherwise, *shmaddr* must be a page-aligned address at which the attach occurs.

In addition to **SHM_RND**, the following flags may be specified in the *shmflg* bit-mask argument:

SHM_EXEC (Linux-specific; since Linux 2.6.9)
: Allow the contents of the segment to be executed. The caller must have execute permission on the segment.

SHM_RDONLY
: Attach the segment for read-only access. The process must have read permission for the segment. If this flag is not specified, the segment is attached for read and write access, and the process must have read and write permission for the segment. There is no notion of a write-only shared memory segment.

SHM_REMAP (Linux-specific)
: This flag specifies that the mapping of the segment should replace any existing mapping in the range starting at *shmaddr* and continuing for the size of the segment. (Normally, an **EINVAL** error would result if a mapping already exists in this address range.) In this case, *shmaddr* must not be NULL.

The *brk*(2) value of the calling process is not altered by the attach. The segment will automatically be detached at process exit. The same segment may be attached as a read and as a read-write one, and more than once, in the process's address space.

A successful **shmat**() call updates the members of the *shmid_ds* structure (see *shmctl*(2)) associated with the shared memory segment as follows:

- *shm_atime* is set to the current time.
- *shm_lpid* is set to the process-ID of the calling process.
- *shm_nattch* is incremented by one.

shmdt()

shmdt() detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. The to-be-detached segment must be currently attached with *shmaddr* equal to the value returned by the attaching **shmat**() call.

On a successful **shmdt**() call, the system updates the members of the *shmid_ds* structure associated with the shared memory segment as follows:

- *shm_dtime* is set to the current time.
- *shm_lpid* is set to the process-ID of the calling process.
- *shm_nattch* is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted.

RETURN VALUE

On success, **shmat**() returns the address of the attached shared memory segment; on error, *(void *) −1* is returned, and *errno* is set to indicate the error.

On success, **shmdt**() returns 0; on error −1 is returned, and *errno* is set to indicate the error.

ERRORS

shmat() can fail with one of the following errors:

EACCES
The calling process does not have the required permissions for the requested attach type, and does not have the **CAP_IPC_OWNER** capability in the user namespace that governs its IPC namespace.

EIDRM
shmid points to a removed identifier.

EINVAL
Invalid *shmid* value, unaligned (i.e., not page-aligned and **SHM_RND** was not specified) or invalid *shmaddr* value, or can't attach segment at *shmaddr*, or **SHM_REMAP** was specified and *shmaddr* was NULL.

ENOMEM
Could not allocate memory for the descriptor or for the page tables.

shmdt() can fail with one of the following errors:

EINVAL
There is no shared memory segment attached at *shmaddr*; or, *shmaddr* is not aligned on a page boundary.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

In SVID 3 (or perhaps earlier), the type of the *shmaddr* argument was changed from *char* * into *const void* *, and the returned type of **shmat**() from *char* * into *void* *.

NOTES

After a *fork*(2), the child inherits the attached shared memory segments.

After an *execve*(2), all attached shared memory segments are detached from the process.

Upon *_exit*(2), all attached shared memory segments are detached from the process.

Using **shmat**() with *shmaddr* equal to NULL is the preferred, portable way of attaching a shared memory segment. Be aware that the shared memory segment attached in this way may be attached at different addresses in different processes. Therefore, any pointers maintained within the shared memory must be made relative (typically to the starting address of the segment), rather than absolute.

On Linux, it is possible to attach a shared memory segment even if it is already marked to be deleted. However, POSIX.1 does not specify this behavior and many other implementations do not support it.

The following system parameter affects **shmat**():

SHMLBA
Segment low boundary address multiple. When explicitly specifying an attach address in a call to **shmat**(), the caller should ensure that the address is a multiple of this value. This is necessary on some architectures, in order either to ensure good CPU cache performance or to ensure that different attaches of the same segment have consistent views within the CPU cache. **SHMLBA** is normally some multiple of the system page size. (On many Linux architectures, **SHMLBA** is the same as the system page size.)

The implementation places no intrinsic per-process limit on the number of shared memory segments (**SHMSEG**).

EXAMPLES

The two programs shown below exchange a string using a shared memory segment. Further details about the programs are given below. First, we show a shell session demonstrating their use.

In one terminal window, we run the "reader" program, which creates a System V shared memory segment and a System V semaphore set. The program prints out the IDs of the created objects, and then waits for the semaphore to change value.

```
$ ./svshm_string_read
shmid = 1114194; semid = 15
```

In another terminal window, we run the "writer" program. The "writer" program takes three command-line arguments: the IDs of the shared memory segment and semaphore set created by the "reader", and a string. It attaches the existing shared memory segment, copies the string to the shared memory, and modifies the semaphore value.

```
$ ./svshm_string_write 1114194 15 'Hello, world'
```

Returning to the terminal where the "reader" is running, we see that the program has ceased waiting on the semaphore and has printed the string that was copied into the shared memory segment by the writer:

```
Hello, world
```

Program source: svshm_string.h

The following header file is included by the "reader" and "writer" programs:

```
/* svshm_string.h

   Licensed under GNU General Public License v2 or later.
*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

union semun {                   /* Used in calls to semctl() */
    int                 val;
    struct semid_ds *   buf;
    unsigned short *    array;
#if defined(__linux__)
    struct seminfo *    __buf;
#endif
};

#define MEM_SIZE 4096
```

Program source: svshm_string_read.c

The "reader" program creates a shared memory segment and a semaphore set containing one semaphore. It then attaches the shared memory object into its address space and initializes the semaphore value to 1. Finally, the program waits for the semaphore value to become 0, and afterwards prints the string that has been copied into the shared memory segment by the "writer".

```
/* svshm_string_read.c

   Licensed under GNU General Public License v2 or later.
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
```

```
#include "svshm_string.h"

int
main(void)
{
    int            semid, shmid;
    char           *addr;
    union semun    arg, dummy;
    struct sembuf  sop;

    /* Create shared memory and semaphore set containing one
       semaphore. */

    shmid = shmget(IPC_PRIVATE, MEM_SIZE, IPC_CREAT | 0600);
    if (shmid == -1)
        errExit("shmget");

    semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
    if (semid == -1)
        errExit("semget");

    /* Attach shared memory into our address space. */

    addr = shmat(shmid, NULL, SHM_RDONLY);
    if (addr == (void *) -1)
        errExit("shmat");

    /* Initialize semaphore 0 in set with value 1. */

    arg.val = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1)
        errExit("semctl");

    printf("shmid = %d; semid = %d\n", shmid, semid);

    /* Wait for semaphore value to become 0. */

    sop.sem_num = 0;
    sop.sem_op = 0;
    sop.sem_flg = 0;

    if (semop(semid, &sop, 1) == -1)
        errExit("semop");

    /* Print the string from shared memory. */

    printf("%s\n", addr);

    /* Remove shared memory and semaphore set. */

    if (shmctl(shmid, IPC_RMID, NULL) == -1)
        errExit("shmctl");
    if (semctl(semid, 0, IPC_RMID, dummy) == -1)
        errExit("semctl");

    exit(EXIT_SUCCESS);
}
```

Program source: svshm_string_write.c

The writer program takes three command-line arguments: the IDs of the shared memory segment and semaphore set that have already been created by the "reader", and a string. It attaches the shared memory segment into its address space, and then decrements the semaphore value to 0 in order to inform the "reader" that it can now examine the contents of the shared memory.

```
/* svshm_string_write.c

   Licensed under GNU General Public License v2 or later.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "svshm_string.h"

int
main(int argc, char *argv[])
{
    int            semid, shmid;
    char           *addr;
    size_t         len;
    struct sembuf  sop;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s shmid semid string\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    len = strlen(argv[3]) + 1;  /* +1 to include trailing '\0' */
    if (len > MEM_SIZE) {
        fprintf(stderr, "String is too big!\n");
        exit(EXIT_FAILURE);
    }

    /* Get object IDs from command-line. */

    shmid = atoi(argv[1]);
    semid = atoi(argv[2]);

    /* Attach shared memory into our address space and copy string
       (including trailing null byte) into memory. */

    addr = shmat(shmid, NULL, 0);
    if (addr == (void *) -1)
        errExit("shmat");

    memcpy(addr, argv[3], len);

    /* Decrement semaphore to 0. */

    sop.sem_num = 0;
    sop.sem_op = -1;
    sop.sem_flg = 0;

    if (semop(semid, &sop, 1) == -1)
        errExit("semop");
```

```
            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

brk(2), *mmap*(2), *shmctl*(2), *shmget*(2), *capabilities*(7), *shm_overview*(7), *sysvipc*(7)

NAME

shutdown − shut down part of a full-duplex connection

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

int shutdown(int *sockfd***, int** *how***);**

DESCRIPTION

The **shutdown**() call causes all or part of a full-duplex connection on the socket associated with *sockfd* to be shut down. If *how* is **SHUT_RD**, further receptions will be disallowed. If *how* is **SHUT_WR**, further transmissions will be disallowed. If *how* is **SHUT_RDWR**, further receptions and transmissions will be disallowed.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
sockfd is not a valid file descriptor.

EINVAL
An invalid value was specified in *how* (but see BUGS).

ENOTCONN
The specified socket is not connected.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.4BSD (first appeared in 4.2BSD).

NOTES

The constants **SHUT_RD**, **SHUT_WR**, **SHUT_RDWR** have the value 0, 1, 2, respectively, and are defined in <*sys/socket.h*> since glibc-2.1.91.

BUGS

Checks for the validity of *how* are done in domain-specific code, and before Linux 3.7 not all domains performed these checks. Most notably, UNIX domain sockets simply ignored invalid values. This problem was fixed for UNIX domain sockets in Linux 3.7.

SEE ALSO

close(2), *connect*(2), *socket*(2), *socket*(7)

NAME

sigaction, rt_sigaction − examine and change a signal action

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int sigaction(int signum,
              const struct sigaction *_Nullable restrict act,
              struct sigaction *_Nullable restrict oldact);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigaction():
_POSIX_C_SOURCE

siginfo_t:
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal. (See *signal*(7) for an overview of signals.)

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like:

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

On some architectures a union is involved: do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* field is not intended for application use. (POSIX does not specify a *sa_restorer* field.) Some further details of the purpose of this field can be found in *sigreturn*(2).

sa_handler specifies the action to be associated with *signum* and can be one of the following:

- **SIG_DFL** for the default action.
- **SIG_IGN** to ignore this signal.
- A pointer to a signal handling function. This function receives the signal number as its only argument.

If **SA_SIGINFO** is specified in *sa_flags*, then *sa_sigaction* (instead of *sa_handler*) specifies the signal-handling function for *signum*. This function receives three arguments, as described below.

sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** flag is used.

sa_flags specifies a set of flags which modify the behavior of the signal. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
: If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see *wait*(2)). This flag is meaningful only when establishing a handler for **SIGCHLD**.

SA_NOCLDWAIT (since Linux 2.6)
If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. See also *waitpid*(2). This flag is meaningful only when establishing a handler for **SIGCHLD**, or when setting that signal's disposition to **SIG_DFL**.

If the **SA_NOCLDWAIT** flag is set when establishing a handler for **SIGCHLD**, POSIX.1 leaves it unspecified whether a **SIGCHLD** signal is generated when a child process terminates. On Linux, a **SIGCHLD** signal is generated in this case; on some other implementations, it is not.

SA_NODEFER
Do not add the signal to the thread's signal mask while the handler is executing, unless the signal is specified in *act.sa_mask*. Consequently, a further instance of the signal may be delivered to the thread while it is executing the handler. This flag is meaningful only when establishing a signal handler.

SA_NOMASK is an obsolete, nonstandard synonym for this flag.

SA_ONSTACK
Call the signal handler on an alternate signal stack provided by *sigaltstack*(2). If an alternate stack is not available, the default stack will be used. This flag is meaningful only when establishing a signal handler.

SA_RESETHAND
Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler.

SA_ONESHOT is an obsolete, nonstandard synonym for this flag.

SA_RESTART
Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is meaningful only when establishing a signal handler. See *signal*(7) for a discussion of system call restarting.

SA_RESTORER
Not intended for application use. This flag is used by C libraries to indicate that the *sa_restorer* field contains the address of a "signal trampoline". See *sigreturn*(2) for more details.

SA_SIGINFO (since Linux 2.2)
The signal handler takes three arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler*. This flag is meaningful only when establishing a signal handler.

SA_UNSUPPORTED (since Linux 5.11)
Used to dynamically probe for flag bit support.

If an attempt to register a handler succeeds with this flag set in *act->sa_flags* alongside other flags that are potentially unsupported by the kernel, and an immediately subsequent **sigaction**() call specifying the same signal number and with a non-NULL *oldact* argument yields **SA_UNSUPPORTED** *clear* in *oldact->sa_flags*, then *oldact->sa_flags* may be used as a bitmask describing which of the potentially unsupported flags are, in fact, supported. See the section "Dynamically probing for flag bit support" below for more details.

SA_EXPOSE_TAGBITS (since Linux 5.11)
Normally, when delivering a signal, an architecture-specific set of tag bits are cleared from the *si_addr* field of *siginfo_t*. If this flag is set, an architecture-specific subset of the tag bits will be preserved in *si_addr*.

Programs that need to be compatible with Linux versions older than 5.11 must use **SA_UNSUPPORTED** to probe for support.

The siginfo_t argument to a SA_SIGINFO handler

When the **SA_SIGINFO** flag is specified in *act.sa_flags*, the signal handler address is passed via the *act.sa_sigaction* field. This handler takes three arguments, as follows:

```
void
handler(int sig, siginfo_t *info, void *ucontext)
{
    ...
```

```
}
```

These three arguments are as follows

sig The number of the signal that caused invocation of the handler.

info A pointer to a *siginfo_t*, which is a structure containing further information about the signal, as described below.

ucontext

This is a pointer to a *ucontext_t* structure, cast to *void* *. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see *sigreturn*(2). Further information about the *ucontext_t* structure can be found in *getcontext*(3) and *signal*(7). Commonly, the handler function doesn't make any use of the third argument.

The *siginfo_t* data type is a structure with the following fields:

```
siginfo_t {
    int      si_signo;     /* Signal number */
    int      si_errno;     /* An errno value */
    int      si_code;      /* Signal code */
    int      si_trapno;    /* Trap number that caused
                              hardware-generated signal
                              (unused on most architectures) */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    clock_t  si_utime;     /* User time consumed */
    clock_t  si_stime;     /* System time consumed */
    union sigval si_value; /* Signal value */
    int      si_int;       /* POSIX.1b signal */
    void    *si_ptr;       /* POSIX.1b signal */
    int      si_overrun;   /* Timer overrun count;
                              POSIX.1b timers */
    int      si_timerid;   /* Timer ID; POSIX.1b timers */
    void    *si_addr;      /* Memory location which caused fault */
    long     si_band;      /* Band event (was int in
                              glibc 2.3.2 and earlier) */
    int      si_fd;        /* File descriptor */
    short    si_addr_lsb;  /* Least significant bit of address
                              (since Linux 2.6.32) */
    void    *si_lower;     /* Lower bound when address violation
                              occurred (since Linux 3.19) */
    void    *si_upper;     /* Upper bound when address violation
                              occurred (since Linux 3.19) */
    int      si_pkey;      /* Protection key on PTE that caused
                              fault (since Linux 4.6) */
    void    *si_call_addr; /* Address of system call instruction
                              (since Linux 3.5) */
    int      si_syscall;   /* Number of attempted system call
                              (since Linux 3.5) */
    unsigned int si_arch;  /* Architecture of attempted system call
                              (since Linux 3.5) */
}
```

si_signo, *si_errno* and *si_code* are defined for all signals. (*si_errno* is generally unused on Linux.) The rest of the struct may be a union, so that one should read only the fields that are meaningful for the given signal:

- Signals sent with *kill*(2) and *sigqueue*(3) fill in *si_pid* and *si_uid*. In addition, signals sent with *sigqueue*(3) fill in *si_int* and *si_ptr* with the values specified by the sender of the signal; see *sigqueue*(3) for more details.

- Signals sent by POSIX.1b timers (since Linux 2.6) fill in *si_overrun* and *si_timerid*. The *si_timerid* field is an internal ID used by the kernel to identify the timer; it is not the same as the timer ID returned by *timer_create*(2). The *si_overrun* field is the timer overrun count; this is the same information as is obtained by a call to *timer_getoverrun*(2). These fields are nonstandard Linux extensions.
- Signals sent for message queue notification (see the description of **SIGEV_SIGNAL** in *mq_notify*(3)) fill in *si_int*/*si_ptr*, with the *sigev_value* supplied to *mq_notify*(3); *si_pid*, with the process ID of the message sender; and *si_uid*, with the real user ID of the message sender.
- **SIGCHLD** fills in *si_pid*, *si_uid*, *si_status*, *si_utime*, and *si_stime*, providing information about the child. The *si_pid* field is the process ID of the child; *si_uid* is the child's real user ID. The *si_status* field contains the exit status of the child (if *si_code* is **CLD_EXITED**), or the signal number that caused the process to change state. The *si_utime* and *si_stime* contain the user and system CPU time used by the child process; these fields do not include the times used by waited-for children (unlike *getrusage*(2) and *times*(2)). Up to Linux 2.6, and since Linux 2.6.27, these fields report CPU time in units of *sysconf(_SC_CLK_TCK)*. In Linux 2.6 kernels before Linux 2.6.27, a bug meant that these fields reported time in units of the (configurable) system jiffy (see *time*(7)).
- **SIGILL**, **SIGFPE**, **SIGSEGV**, **SIGBUS**, and **SIGTRAP** fill in *si_addr* with the address of the fault. On some architectures, these signals also fill in the *si_trapno* field.

  Some suberrors of **SIGBUS**, in particular **BUS_MCEERR_AO** and **BUS_MCEERR_AR**, also fill in *si_addr_lsb*. This field indicates the least significant bit of the reported address and therefore the extent of the corruption. For example, if a full page was corrupted, *si_addr_lsb* contains *log2(sysconf(_SC_PAGESIZE))*. When **SIGTRAP** is delivered in response to a *ptrace*(2) event (PTRACE_EVENT_foo), *si_addr* is not populated, but *si_pid* and *si_uid* are populated with the respective process ID and user ID responsible for delivering the trap. In the case of *seccomp*(2), the tracee will be shown as delivering the event. **BUS_MCEERR_*** and *si_addr_lsb* are Linux-specific extensions.

  The **SEGV_BNDERR** suberror of **SIGSEGV** populates *si_lower* and *si_upper*.

  The **SEGV_PKUERR** suberror of **SIGSEGV** populates *si_pkey*.
- **SIGIO**/**SIGPOLL** (the two names are synonyms on Linux) fills in *si_band* and *si_fd*. The *si_band* event is a bit mask containing the same values as are filled in the *revents* field by *poll*(2). The *si_fd* field indicates the file descriptor for which the I/O event occurred; for further details, see the description of **F_SETSIG** in *fcntl*(2).
- **SIGSYS**, generated (since Linux 3.5) when a seccomp filter returns **SECCOMP_RET_TRAP**, fills in *si_call_addr*, *si_syscall*, *si_arch*, *si_errno*, and other fields as described in *seccomp*(2).

The si_code field

The *si_code* field inside the *siginfo_t* argument that is passed to a **SA_SIGINFO** signal handler is a value (not a bit mask) indicating why this signal was sent. For a *ptrace*(2) event, *si_code* will contain **SIGTRAP** and have the ptrace event in the high byte:

```
(SIGTRAP | PTRACE_EVENT_foo << 8).
```

For a non-**ptrace**(2) event, the values that can appear in *si_code* are described in the remainder of this section. Since glibc 2.20, the definitions of most of these symbols are obtained from <*signal.h*> by defining feature test macros (before including *any* header file) as follows:

- **_XOPEN_SOURCE** with the value 500 or greater;
- **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**; or
- **_POSIX_C_SOURCE** with the value 200809L or greater.

For the **TRAP_*** constants, the symbol definitions are provided only in the first two cases. Before glibc 2.20, no feature test macros were required to obtain these symbols.

For a regular signal, the following list shows the values which can be placed in *si_code* for any signal, along with the reason that the signal was generated.

SI_USER
: *kill*(2).

SI_KERNEL
Sent by the kernel.

SI_QUEUE
sigqueue(3).

SI_TIMER
POSIX timer expired.

SI_MESGQ (since Linux 2.6.6)
POSIX message queue state changed; see *mq_notify*(3).

SI_ASYNCIO
AIO completed.

SI_SIGIO
Queued **SIGIO** (only up to Linux 2.2; from Linux 2.4 onward **SIGIO**/**SIGPOLL** fills in *si_code* as described below).

SI_TKILL (since Linux 2.4.19)
tkill(2) or *tgkill*(2).

The following values can be placed in *si_code* for a **SIGILL** signal:

ILL_ILLOPC
Illegal opcode.

ILL_ILLOPN
Illegal operand.

ILL_ILLADR
Illegal addressing mode.

ILL_ILLTRP
Illegal trap.

ILL_PRVOPC
Privileged opcode.

ILL_PRVREG
Privileged register.

ILL_COPROC
Coprocessor error.

ILL_BADSTK
Internal stack error.

The following values can be placed in *si_code* for a **SIGFPE** signal:

FPE_INTDIV
Integer divide by zero.

FPE_INTOVF
Integer overflow.

FPE_FLTDIV
Floating-point divide by zero.

FPE_FLTOVF
Floating-point overflow.

FPE_FLTUND
Floating-point underflow.

FPE_FLTRES
Floating-point inexact result.

FPE_FLTINV
Floating-point invalid operation.

FPE_FLTSUB
Subscript out of range.

The following values can be placed in *si_code* for a **SIGSEGV** signal:

SEGV_MAPERR
Address not mapped to object.

SEGV_ACCERR
Invalid permissions for mapped object.

SEGV_BNDERR (since Linux 3.19)
Failed address bound checks.

SEGV_PKUERR (since Linux 4.6)
Access was denied by memory protection keys. See *pkeys*(7). The protection key which applied to this access is available via *si_pkey*.

The following values can be placed in *si_code* for a **SIGBUS** signal:

BUS_ADRALN
Invalid address alignment.

BUS_ADRERR
Nonexistent physical address.

BUS_OBJERR
Object-specific hardware error.

BUS_MCEERR_AR (since Linux 2.6.32)
Hardware memory error consumed on a machine check; action required.

BUS_MCEERR_AO (since Linux 2.6.32)
Hardware memory error detected in process but not consumed; action optional.

The following values can be placed in *si_code* for a **SIGTRAP** signal:

TRAP_BRKPT
Process breakpoint.

TRAP_TRACE
Process trace trap.

TRAP_BRANCH (since Linux 2.4, IA64 only)
Process taken branch trap.

TRAP_HWBKPT (since Linux 2.4, IA64 only)
Hardware breakpoint/watchpoint.

The following values can be placed in *si_code* for a **SIGCHLD** signal:

CLD_EXITED
Child has exited.

CLD_KILLED
Child was killed.

CLD_DUMPED
Child terminated abnormally.

CLD_TRAPPED
Traced child has trapped.

CLD_STOPPED
Child has stopped.

CLD_CONTINUED (since Linux 2.6.9)
Stopped child has continued.

The following values can be placed in *si_code* for a **SIGIO**/**SIGPOLL** signal:

POLL_IN
Data input available.

POLL_OUT
Output buffers available.

POLL_MSG
Input message available.

POLL_ERR
I/O error.

POLL_PRI
High priority input available.

POLL_HUP
Device disconnected.

The following value can be placed in *si_code* for a **SIGSYS** signal:

SYS_SECCOMP (since Linux 3.5)
Triggered by a *seccomp*(2) filter rule.

Dynamically probing for flag bit support

The **sigaction**() call on Linux accepts unknown bits set in *act->sa_flags* without error. The behavior of the kernel starting with Linux 5.11 is that a second **sigaction**() will clear unknown bits from *oldact->sa_flags*. However, historically, a second **sigaction**() call would typically leave those bits set in *oldact->sa_flags*.

This means that support for new flags cannot be detected simply by testing for a flag in *sa_flags*, and a program must test that **SA_UNSUPPORTED** has been cleared before relying on the contents of *sa_flags*.

Since the behavior of the signal handler cannot be guaranteed unless the check passes, it is wise to either block the affected signal while registering the handler and performing the check in this case, or where this is not possible, for example if the signal is synchronous, to issue the second **sigaction**() in the signal handler itself.

In kernels that do not support a specific flag, the kernel's behavior is as if the flag was not set, even if the flag was set in *act->sa_flags*.

The flags **SA_NOCLDSTOP**, **SA_NOCLDWAIT**, **SA_SIGINFO**, **SA_ONSTACK**, **SA_RESTART**, **SA_NODEFER**, **SA_RESETHAND**, and, if defined by the architecture, **SA_RESTORER** may not be reliably probed for using this mechanism, because they were introduced before Linux 5.11. However, in general, programs may assume that these flags are supported, since they have all been supported since Linux 2.6, which was released in the year 2003.

See EXAMPLES below for a demonstration of the use of **SA_UNSUPPORTED**.

RETURN VALUE

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
act or *oldact* points to memory which is not a valid part of the process address space.

EINVAL
An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught or ignored.

VERSIONS

C library/kernel differences

The glibc wrapper function for **sigaction**() gives an error (**EINVAL**) on attempts to change the disposition of the two real-time signals used internally by the NPTL threading implementation. See *nptl*(7) for details.

On architectures where the signal trampoline resides in the C library, the glibc wrapper function for **sigaction**() places the address of the trampoline code in the *act.sa_restorer* field and sets the **SA_RESTORER** flag in the *act.sa_flags* field. See *sigreturn*(2).

The original Linux system call was named **sigaction**(). However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* type supported by that system call was no longer fit for

purpose. Consequently, a new system call, **rt_sigaction**(), was added to support an enlarged *sigset_t* type. The new system call takes a fourth argument, *size_t sigsetsize*, which specifies the size in bytes of the signal sets in *act.sa_mask* and *oldact.sa_mask*. This argument is currently required to have the value *sizeof(sigset_t)* (or the error **EINVAL** results). The glibc **sigaction**() wrapper function hides these details from us, transparently calling **rt_sigaction**() when the kernel provides it.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

POSIX.1-1990 disallowed setting the action for **SIGCHLD** to **SIG_IGN**. POSIX.1-2001 and later allow this possibility, so that ignoring **SIGCHLD** can be used to prevent the creation of zombies (see *wait*(2)). Nevertheless, the historical BSD and System V behaviors for ignoring **SIGCHLD** differ, so that the only completely portable method of ensuring that terminated children do not become zombies is to catch the **SIGCHLD** signal and perform a *wait*(2) or similar.

POSIX.1-1990 specified only **SA_NOCLDSTOP**. POSIX.1-2001 added **SA_NOCLDWAIT**, **SA_NODEFER**, **SA_ONSTACK**, **SA_RESETHAND**, **SA_RESTART**, and **SA_SIGINFO** as XSI extensions. POSIX.1-2008 moved **SA_NODEFER**, **SA_RESETHAND**, **SA_RESTART**, and **SA_SIGINFO** to the base specifications. Use of these latter values in *sa_flags* may be less portable in applications intended for older UNIX implementations.

The **SA_RESETHAND** flag is compatible with the SVr4 flag of the same name.

The **SA_NODEFER** flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and later. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any *sa_mask* settings).

NOTES

A child created via *fork*(2) inherits a copy of its parent's signal dispositions. During an *execve*(2), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by *kill*(2) or *raise*(3). Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by −1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

sigaction() can be called with a NULL second argument to query the current signal handler. It can also be used to check whether a given signal is valid for the current machine by calling it with NULL second and third arguments.

It is not possible to block **SIGKILL** or **SIGSTOP** (by specifying them in *sa_mask*). Attempts to do so are silently ignored.

See *sigsetops*(3) for details on manipulating signal sets.

See *signal-safety*(7) for a list of the async-signal-safe functions that can be safely called inside from inside a signal handler.

Undocumented

Before the introduction of **SA_SIGINFO**, it was also possible to get some additional information about the signal. This was done by providing an *sa_handler* signal handler with a second argument of type *struct sigcontext*, which is the same structure as the one that is passed in the *uc_mcontext* field of the *ucontext* structure that is passed (via a pointer) in the third argument of the *sa_sigaction* handler. See the relevant Linux kernel sources for details. This use is obsolete now.

BUGS

When delivering a signal with a **SA_SIGINFO** handler, the kernel does not always provide meaningful values for all of the fields of the *siginfo_t* that are relevant for that signal.

Up to and including Linux 2.6.13, specifying **SA_NODEFER** in *sa_flags* prevents not only the delivered signal from being masked during execution of the handler, but also the signals specified in *sa_mask*. This bug was fixed in Linux 2.6.14.

EXAMPLES

See *mprotect*(2).

Probing for flag support

The following example program exits with status **EXIT_SUCCESS** if **SA_EXPOSE_TAGBITS** is determined to be supported, and **EXIT_FAILURE** otherwise.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
handler(int signo, siginfo_t *info, void *context)
{
    struct sigaction oldact;

    if (sigaction(SIGSEGV, NULL, &oldact) == -1
        || (oldact.sa_flags & SA_UNSUPPORTED)
        || !(oldact.sa_flags & SA_EXPOSE_TAGBITS))
    {
        _exit(EXIT_FAILURE);
    }
    _exit(EXIT_SUCCESS);
}

int
main(void)
{
    struct sigaction act = { 0 };

    act.sa_flags = SA_SIGINFO | SA_UNSUPPORTED | SA_EXPOSE_TAGBITS;
    act.sa_sigaction = &handler;
    if (sigaction(SIGSEGV, &act, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    raise(SIGSEGV);
}
```

SEE ALSO

kill(1), *kill*(2), *pause*(2), *pidfd_send_signal*(2), *restart_syscall*(2), *seccomp*(2), *sigaltstack*(2), *signal*(2), *signalfd*(2), *sigpending*(2), *sigprocmask*(2), *sigreturn*(2), *sigsuspend*(2), *wait*(2), *killpg*(3), *raise*(3), *siginterrupt*(3), *sigqueue*(3), *sigsetops*(3), *sigvec*(3), *core*(5), *signal*(7)

NAME

sigaltstack − set and/or get signal stack context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

int sigaltstack(const stack_t *_Nullable restrict *ss*,
stack_t *_Nullable restrict *old_ss*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigaltstack():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

sigaltstack() allows a thread to define a new alternate signal stack and/or retrieve the state of an existing alternate signal stack. An alternate signal stack is used during the execution of a signal handler if the establishment of that handler (see *sigaction*(2)) requested it.

The normal sequence of events for using an alternate signal stack is the following:

1. Allocate an area of memory to be used for the alternate signal stack.
2. Use **sigaltstack**() to inform the system of the existence and location of the alternate signal stack.
3. When establishing a signal handler using *sigaction*(2), inform the system that the signal handler should be executed on the alternate signal stack by specifying the **SA_ONSTACK** flag.

The *ss* argument is used to specify a new alternate signal stack, while the *old_ss* argument is used to retrieve information about the currently established signal stack. If we are interested in performing just one of these tasks, then the other argument can be specified as NULL.

The *stack_t* type used to type the arguments of this function is defined as follows:

```
typedef struct {
    void  *ss_sp;     /* Base address of stack */
    int    ss_flags;  /* Flags */
    size_t ss_size;   /* Number of bytes in stack */
} stack_t;
```

To establish a new alternate signal stack, the fields of this structure are set as follows:

ss.ss_flags
: This field contains either 0, or the following flag:

  SS_AUTODISARM (since Linux 4.7)
  : Clear the alternate signal stack settings on entry to the signal handler. When the signal handler returns, the previous alternate signal stack settings are restored.

    This flag was added in order to make it safe to switch away from the signal handler with *swapcontext*(3). Without this flag, a subsequently handled signal will corrupt the state of the switched-away signal handler. On kernels where this flag is not supported, **sigaltstack**() fails with the error **EINVAL** when this flag is supplied.

ss.ss_sp
: This field specifies the starting address of the stack. When a signal handler is invoked on the alternate stack, the kernel automatically aligns the address given in *ss.ss_sp* to a suitable address boundary for the underlying hardware architecture.

ss.ss_size
: This field specifies the size of the stack. The constant **SIGSTKSZ** is defined to be large enough to cover the usual size requirements for an alternate signal stack, and the constant **MINSIGSTKSZ** defines the minimum size required to execute a signal handler.

To disable an existing stack, specify *ss.ss_flags* as **SS_DISABLE**. In this case, the kernel ignores any other flags in *ss.ss_flags* and the remaining fields in *ss*.

If *old_ss* is not NULL, then it is used to return information about the alternate signal stack which was in effect prior to the call to **sigaltstack**(). The *old_ss.ss_sp* and *old_ss.ss_size* fields return the starting address and size of that stack. The *old_ss.ss_flags* may return either of the following values:

SS_ONSTACK
The thread is currently executing on the alternate signal stack. (Note that it is not possible to change the alternate signal stack if the thread is currently executing on it.)

SS_DISABLE
The alternate signal stack is currently disabled.

Alternatively, this value is returned if the thread is currently executing on an alternate signal stack that was established using the **SS_AUTODISARM** flag. In this case, it is safe to switch away from the signal handler with *swapcontext*(3). It is also possible to set up a different alternative signal stack using a further call to **sigaltstack**().

SS_AUTODISARM
The alternate signal stack has been marked to be autodisarmed as described above.

By specifying *ss* as NULL, and *old_ss* as a non-NULL value, one can obtain the current settings for the alternate signal stack without changing them.

RETURN VALUE

sigaltstack() returns 0 on success, or −1 on failure with *errno* set to indicate the error.

ERRORS

EFAULT
Either *ss* or *old_ss* is not NULL and points to an area outside of the process's address space.

EINVAL
ss is not NULL and the *ss_flags* field contains an invalid flag.

ENOMEM
The specified size of the new alternate signal stack *ss.ss_size* was less than **MINSIGSTKSZ**.

EPERM
An attempt was made to change the alternate signal stack while it was active (i.e., the thread was already executing on the current alternate signal stack).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigaltstack()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

SS_AUTODISARM is a Linux extension.

HISTORY

POSIX.1-2001, SUSv2, SVr4.

NOTES

The most common usage of an alternate signal stack is to handle the **SIGSEGV** signal that is generated if the space available for the standard stack is exhausted: in this case, a signal handler for **SIGSEGV** cannot be invoked on the standard stack; if we wish to handle it, we must use an alternate signal stack.

Establishing an alternate signal stack is useful if a thread expects that it may exhaust its standard stack. This may occur, for example, because the stack grows so large that it encounters the upwardly growing heap, or it reaches a limit established by a call to **setrlimit(RLIMIT_STACK, &rlim)**. If the standard stack is exhausted, the kernel sends the thread a **SIGSEGV** signal. In these circumstances the only way to catch this signal is on an alternate signal stack.

On most hardware architectures supported by Linux, stacks grow downward. **sigaltstack**() automatically takes account of the direction of stack growth.

Functions called from a signal handler executing on an alternate signal stack will also use the alternate signal stack. (This also applies to any handlers invoked for other signals while the thread is executing on the alternate signal stack.) Unlike the standard stack, the system does not automatically extend the

alternate signal stack. Exceeding the allocated size of the alternate signal stack will lead to unpredictable results.

A successful call to *execve*(2) removes any existing alternate signal stack. A child process created via *fork*(2) inherits a copy of its parent's alternate signal stack settings. The same is also true for a child process created using *clone*(2), unless the clone flags include **CLONE_VM** and do not include **CLONE_VFORK**, in which case any alternate signal stack that was established in the parent is disabled in the child process.

sigaltstack() supersedes the older **sigstack**() call. For backward compatibility, glibc also provides **sigstack**(). All new applications should be written using **sigaltstack**().

History

4.2BSD had a **sigstack**() system call. It used a slightly different struct, and had the major disadvantage that the caller had to know the direction of stack growth.

BUGS

In Linux 2.2 and earlier, the only flag that could be specified in *ss.sa_flags* was **SS_DISABLE**. In the lead up to the release of the Linux 2.4 kernel, a change was made to allow **sigaltstack**() to allow *ss.ss_flags==SS_ONSTACK* with the same meaning as *ss.ss_flags==0* (i.e., the inclusion of **SS_ONSTACK** in *ss.ss_flags* is a no-op). On other implementations, and according to POSIX.1, **SS_ONSTACK** appears only as a reported flag in *old_ss.ss_flags*. On Linux, there is no need ever to specify **SS_ONSTACK** in *ss.ss_flags*, and indeed doing so should be avoided on portability grounds: various other systems give an error if **SS_ONSTACK** is specified in *ss.ss_flags*.

EXAMPLES

The following code segment demonstrates the use of **sigaltstack**() (and *sigaction*(2)) to install an alternate signal stack that is employed by a handler for the **SIGSEGV** signal:

```
stack_t ss;

ss.ss_sp = malloc(SIGSTKSZ);
if (ss.ss_sp == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

ss.ss_size = SIGSTKSZ;
ss.ss_flags = 0;
if (sigaltstack(&ss, NULL) == -1) {
    perror("sigaltstack");
    exit(EXIT_FAILURE);
}

sa.sa_flags = SA_ONSTACK;
sa.sa_handler = handler();      /* Address of a signal handler */
sigemptyset(&sa.sa_mask);
if (sigaction(SIGSEGV, &sa, NULL) == -1) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

SEE ALSO

execve(2), *setrlimit*(2), *sigaction*(2), *siglongjmp*(3), *sigsetjmp*(3), *signal*(7)

NAME

signal − ANSI C signal handling

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int *signum***, sighandler_t** *handler***);**

DESCRIPTION

WARNING: the behavior of **signal**() varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use**: use *sigaction*(2) instead. See *Portability* below.

signal() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

If the signal *signum* is delivered to the process, then one of the following happens:

* If the disposition is set to **SIG_IGN**, then the signal is ignored.
* If the disposition is set to **SIG_DFL**, then the default action associated with the signal (see *signal*(7)) occurs.
* If the disposition is set to a function, then first either the disposition is reset to **SIG_DFL**, or the signal is blocked (see *Portability* below), and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.

RETURN VALUE

signal() returns the previous value of the signal handler. On failure, it returns **SIG_ERR**, and *errno* is set to indicate the error.

ERRORS

EINVAL

signum is invalid.

VERSIONS

The use of *sighandler_t* is a GNU extension, exposed if **_GNU_SOURCE** is defined; glibc also defines (the BSD-derived) *sig_t* if **_BSD_SOURCE** (glibc 2.19 and earlier) or **_DEFAULT_SOURCE** (glibc 2.19 and later) is defined. Without use of such a type, the declaration of **signal**() is the somewhat harder to read:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

Portability

The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**

POSIX.1 solved the portability mess by specifying *sigaction*(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**().

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

In the original UNIX systems, when a handler that was established using **signal**() was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG_DFL**, and the system did not block delivery of further instances of the signal. This is equivalent to calling *sigaction*(2) with the following flags:

```
sa.sa_flags = SA_RESETHAND | SA_NODEFER;
```

System V also provides these semantics for **signal**(). This was bad because the signal might be

delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler.

BSD improved on this situation, but unfortunately also changed the semantics of the existing **signal**() interface while doing so. On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see *signal*(7)). The BSD semantics are equivalent to calling *sigaction*(2) with the following flags:

```
sa.sa_flags = SA_RESTART;
```

The situation on Linux is as follows:

- The kernel's **signal**() system call provides System V semantics.
- By default, in glibc 2 and later, the **signal**() wrapper function does not invoke the kernel system call. Instead, it calls *sigaction*(2) using flags that supply BSD semantics. This default behavior is provided as long as a suitable feature test macro is defined: **_BSD_SOURCE** on glibc 2.19 and earlier or **_DEFAULT_SOURCE** in glibc 2.19 and later. (By default, these macros are defined; see *feature_test_macros*(7) for details.) If such a feature test macro is not defined, then **signal**() provides System V semantics.

NOTES

The effects of **signal**() in a multithreaded process are unspecified.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by *kill*(2) or *raise*(3). Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by −1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

See *sigaction*(2) for details on what happens when the disposition **SIGCHLD** is set to **SIG_IGN**.

See *signal-safety*(7) for a list of the async-signal-safe functions that can be safely called from inside a signal handler.

SEE ALSO

kill(1), *alarm*(2), *kill*(2), *pause*(2), *sigaction*(2), *signalfd*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *bsd_signal*(3), *killpg*(3), *raise*(3), *siginterrupt*(3), *sigqueue*(3), *sigsetops*(3), *sigvec*(3), *sysv_signal*(3), *signal*(7)

NAME

signalfd − create a file descriptor for accepting signals

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/signalfd.h>

int signalfd(int *fd***, const sigset_t ****mask***, int** *flags***);**

DESCRIPTION

signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternative to the use of a signal handler or *sigwaitinfo*(2), and has the advantage that the file descriptor may be monitored by *select*(2), *poll*(2), and *epoll*(7).

The *mask* argument specifies the set of signals that the caller wishes to accept via the file descriptor. This argument is a signal set whose contents can be initialized using the macros described in *sigsetops*(3). Normally, the set of signals to be received via the file descriptor should be blocked using *sigprocmask*(2), to prevent the signals being handled according to their default dispositions. It is not possible to receive **SIGKILL** or **SIGSTOP** signals via a signalfd file descriptor; these signals are silently ignored if specified in *mask*.

If the *fd* argument is −1, then the call creates a new file descriptor and associates the signal set specified in *mask* with that file descriptor. If *fd* is not −1, then it must specify a valid existing signalfd file descriptor, and *mask* is used to replace the signal set associated with that file descriptor.

Starting with Linux 2.6.27, the following values may be bitwise ORed in *flags* to change the behavior of **signalfd**():

SFD_NONBLOCK
: Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

SFD_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

Up to Linux 2.6.26, the *flags* argument is unused, and must be specified as zero.

signalfd() returns a file descriptor that supports the following operations:

read(2)
: If one or more of the signals specified in *mask* is pending for the process, then the buffer supplied to *read*(2) is used to return one or more *signalfd_siginfo* structures (see below) that describe the signals. The *read*(2) returns information for as many signals as are pending and will fit in the supplied buffer. The buffer must be at least *sizeof(struct signalfd_siginfo)* bytes. The return value of the *read*(2) is the total number of bytes read.

  As a consequence of the *read*(2), the signals are consumed, so that they are no longer pending for the process (i.e., will not be caught by signal handlers, and cannot be accepted using *sigwaitinfo*(2)).

  If none of the signals in *mask* is pending for the process, then the *read*(2) either blocks until one of the signals in *mask* is generated for the process, or fails with the error **EAGAIN** if the file descriptor has been made nonblocking.

poll(2)
select(2)
(and similar)
: The file descriptor is readable (the *select*(2) *readfds* argument; the *poll*(2) **POLLIN** flag) if one or more of the signals in *mask* is pending for the process.

  The signalfd file descriptor also supports the other file-descriptor multiplexing APIs: *pselect*(2), *ppoll*(2), and *epoll*(7).

close(2)
: When the file descriptor is no longer required it should be closed. When all file descriptors associated with the same signalfd object have been closed, the resources for object are freed by

the kernel.

The signalfd_siginfo structure

The format of the *signalfd_siginfo* structure(s) returned by *read*(2)s from a signalfd file descriptor is as follows:

```
struct signalfd_siginfo {
    uint32_t ssi_signo;    /* Signal number */
    int32_t  ssi_errno;    /* Error number (unused) */
    int32_t  ssi_code;     /* Signal code */
    uint32_t ssi_pid;      /* PID of sender */
    uint32_t ssi_uid;      /* Real UID of sender */
    int32_t  ssi_fd;       /* File descriptor (SIGIO) */
    uint32_t ssi_tid;      /* Kernel timer ID (POSIX timers)
    uint32_t ssi_band;     /* Band event (SIGIO) */
    uint32_t ssi_overrun;  /* POSIX timer overrun count */
    uint32_t ssi_trapno;   /* Trap number that caused signal */
    int32_t  ssi_status;   /* Exit status or signal (SIGCHLD) */
    int32_t  ssi_int;      /* Integer sent by sigqueue(3) */
    uint64_t ssi_ptr;      /* Pointer sent by sigqueue(3) */
    uint64_t ssi_utime;    /* User CPU time consumed (SIGCHLD) */
    uint64_t ssi_stime;    /* System CPU time consumed
                              (SIGCHLD) */
    uint64_t ssi_addr;     /* Address that generated signal
                              (for hardware-generated signals) */
    uint16_t ssi_addr_lsb; /* Least significant bit of address
                              (SIGBUS; since Linux 2.6.37) */
    uint8_t  pad[X];       /* Pad size to 128 bytes (allow for
                              additional fields in the future) */
};
```

Each of the fields in this structure is analogous to the similarly named field in the *siginfo_t* structure. The *siginfo_t* structure is described in *sigaction*(2). Not all fields in the returned *signalfd_siginfo* structure will be valid for a specific signal; the set of valid fields can be determined from the value returned in the *ssi_code* field. This field is the analog of the *siginfo_t si_code* field; see *sigaction*(2) for details.

fork(2) semantics

After a *fork*(2), the child inherits a copy of the signalfd file descriptor. A *read*(2) from the file descriptor in the child will return information about signals queued to the child.

Semantics of file descriptor passing

As with other file descriptors, signalfd file descriptors can be passed to another process via a UNIX domain socket (see *unix*(7)). In the receiving process, a *read*(2) from the received file descriptor will return information about signals queued to that process.

execve(2) semantics

Just like any other file descriptor, a signalfd file descriptor remains open across an *execve*(2), unless it has been marked for close-on-exec (see *fcntl*(2)). Any signals that were available for reading before the *execve*(2) remain available to the newly loaded program. (This is analogous to traditional signal semantics, where a blocked signal that is pending remains pending across an *execve*(2).)

Thread semantics

The semantics of signalfd file descriptors in a multithreaded program mirror the standard semantics for signals. In other words, when a thread reads from a signalfd file descriptor, it will read the signals that are directed to the thread itself and the signals that are directed to the process (i.e., the entire thread group). (A thread will not be able to read signals that are directed to other threads in the process.)

epoll(7) semantics

If a process adds (via *epoll_ctl*(2)) a signalfd file descriptor to an *epoll*(7) instance, then *epoll_wait*(2) returns events only for signals sent to that process. In particular, if the process then uses *fork*(2) to create a child process, then the child will be able to *read*(2) signals that are sent to it using the signalfd file descriptor, but *epoll_wait*(2) will **not** indicate that the signalfd file descriptor is ready. In this scenario,

a possible workaround is that after the *fork*(2), the child process can close the signalfd file descriptor that it inherited from the parent process and then create another signalfd file descriptor and add it to the epoll instance. Alternatively, the parent and the child could delay creating their (separate) signalfd file descriptors and adding them to the epoll instance until after the call to *fork*(2).

RETURN VALUE

On success, **signalfd**() returns a signalfd file descriptor; this is either a new file descriptor (if *fd* was −1), or *fd* if *fd* was a valid signalfd file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
The *fd* file descriptor is not a valid file descriptor.

EINVAL
fd is not a valid signalfd file descriptor.

EINVAL
flags is invalid; or, in Linux 2.6.26 or earlier, *flags* is nonzero.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
Could not mount (internal) anonymous inode device.

ENOMEM
There was insufficient memory to create a new signalfd file descriptor.

VERSIONS

C library/kernel differences

The underlying Linux system call requires an additional argument, *size_t sizemask*, which specifies the size of the *mask* argument. The glibc **signalfd**() wrapper function does not include this argument, since it provides the required value for the underlying system call.

There are two underlying Linux system calls: **signalfd**() and the more recent **signalfd4**(). The former system call does not implement a *flags* argument. The latter system call implements the *flags* values described above. Starting with glibc 2.9, the **signalfd**() wrapper function will use **signalfd4**() where it is available.

STANDARDS

Linux.

HISTORY

signalfd()
Linux 2.6.22, glibc 2.8.

signalfd4()
Linux 2.6.27.

NOTES

A process can create multiple signalfd file descriptors. This makes it possible to accept different signals on different file descriptors. (This may be useful if monitoring the file descriptors using *select*(2), *poll*(2), or *epoll*(7): the arrival of different signals will make different file descriptors ready.) If a signal appears in the *mask* of more than one of the file descriptors, then occurrences of that signal can be read (once) from any one of the file descriptors.

Attempts to include **SIGKILL** and **SIGSTOP** in *mask* are silently ignored.

The signal mask employed by a signalfd file descriptor can be viewed via the entry for the corresponding file descriptor in the process's */proc/*pid*/fdinfo* directory. See *proc*(5) for further details.

Limitations

The signalfd mechanism can't be used to receive signals that are synchronously generated, such as the **SIGSEGV** signal that results from accessing an invalid memory address or the **SIGFPE** signal that results from an arithmetic error. Such signals can be caught only via signal handler.

As described above, in normal usage one blocks the signals that will be accepted via **signalfd**(). If spawning a child process to execute a helper program (that does not need the signalfd file descriptor), then, after the call to *fork*(2), you will normally want to unblock those signals before calling *execve*(2), so that the helper program can see any signals that it expects to see. Be aware, however, that this won't be possible in the case of a helper program spawned behind the scenes by any library function that the program may call. In such cases, one must fall back to using a traditional signal handler that writes to a file descriptor monitored by *select*(2), *poll*(2), or *epoll*(7).

BUGS

Before Linux 2.6.25, the *ssi_ptr* and *ssi_int* fields are not filled in with the data accompanying a signal sent by *sigqueue*(3).

EXAMPLES

The program below accepts the signals **SIGINT** and **SIGQUIT** via a signalfd file descriptor. The program terminates after accepting a **SIGQUIT** signal. The following shell session demonstrates the use of the program:

```
$ ./signalfd_demo
^C                    # Control-C generates SIGINT
Got SIGINT
^C
Got SIGINT
^\                     # Control-\ generates SIGQUIT
Got SIGQUIT
$
```

Program source

```
#include <err.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/signalfd.h>
#include <unistd.h>

int
main(void)
{
    int                     sfd;
    ssize_t                 s;
    sigset_t                mask;
    struct signalfd_siginfo fdsi;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);

    /* Block signals so that they aren't handled
       according to their default dispositions. */

    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        err(EXIT_FAILURE, "sigprocmask");

    sfd = signalfd(-1, &mask, 0);
    if (sfd == -1)
        err(EXIT_FAILURE, "signalfd");

    for (;;) {
        s = read(sfd, &fdsi, sizeof(fdsi));
        if (s != sizeof(fdsi))
            err(EXIT_FAILURE, "read");
```

```
            if (fdsi.ssi_signo == SIGINT) {
                printf("Got SIGINT\n");
            } else if (fdsi.ssi_signo == SIGQUIT) {
                printf("Got SIGQUIT\n");
                exit(EXIT_SUCCESS);
            } else {
                printf("Read unexpected signal\n");
            }
        }
    }
```

SEE ALSO

eventfd(2), *poll*(2), *read*(2), *select*(2), *sigaction*(2), *sigprocmask*(2), *sigwaitinfo*(2), *timerfd_create*(2), *sigsetops*(3), *sigwait*(3), *epoll*(7), *signal*(7)

NAME

sigpending, rt_sigpending − examine pending signals

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigpending():
_POSIX_C_SOURCE

DESCRIPTION

sigpending() returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in *set*.

RETURN VALUE

sigpending() returns 0 on success. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
set points to memory which is not a valid part of the process address space.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

C library/kernel differences

The original Linux system call was named **sigpending**(). However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* argument supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigpending**(), was added to support an enlarged *sigset_t* type. The new system call takes a second argument, *size_t sigsetsize*, which specifies the size in bytes of the signal set in *set*. The glibc **sigpending**() wrapper function hides these details from us, transparently calling **rt_sigpending**() when the kernel provides it.

NOTES

See *sigsetops*(3) for details on manipulating signal sets.

If a signal is both blocked and has a disposition of "ignored", it is *not* added to the mask of pending signals when generated.

The set of signals that is pending for a thread is the union of the set of signals that is pending for that thread and the set of signals that is pending for the process as a whole; see *signal*(7).

A child created via *fork*(2) initially has an empty pending signal set; the pending signal set is preserved across an *execve*(2).

BUGS

Up to and including glibc 2.2.1, there is a bug in the wrapper function for **sigpending**() which means that information about pending real-time signals is not correctly returned.

SEE ALSO

kill(2), *sigaction*(2), *signal*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetops*(3), *signal*(7)

NAME

sigprocmask, rt_sigprocmask − examine and change blocked signals

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

/* Prototype for the glibc wrapper function */
int sigprocmask(int how, const sigset_t *_Nullable restrict set,
                sigset_t *_Nullable restrict oldset);

#include <signal.h>          /* Definition of SIG_* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

/* Prototype for the underlying system call */
int syscall(SYS_rt_sigprocmask, int how,
                const kernel_sigset_t *_Nullable set,
                kernel_sigset_t *_Nullable oldset,
                size_t sigsetsize);

/* Prototype for the legacy system call */
[[deprecated]] int syscall(SYS_sigprocmask, int how,
                const old_kernel_sigset_t *_Nullable set,
                old_kernel_sigset_t *_Nullable oldset);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigprocmask():
_POSIX_C_SOURCE

DESCRIPTION

sigprocmask() is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller (see also *signal*(7) for more details).

The behavior of the call is dependent on the value of *how*, as follows.

SIG_BLOCK
The set of blocked signals is the union of the current set and the *set* argument.

SIG_UNBLOCK
The signals in *set* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK
The set of blocked signals is set to the argument *set*.

If *oldset* is non-NULL, the previous value of the signal mask is stored in *oldset*.

If *set* is NULL, then the signal mask is unchanged (i.e., *how* is ignored), but the current value of the signal mask is nevertheless returned in *oldset* (if it is not NULL).

A set of functions for modifying and inspecting variables of type *sigset_t* ("signal sets") is described in *sigsetops*(3).

The use of **sigprocmask**() is unspecified in a multithreaded process; see *pthread_sigmask*(3).

RETURN VALUE

sigprocmask() returns 0 on success. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
The *set* or *oldset* argument points outside the process's allocated address space.

EINVAL
Either the value specified in *how* was invalid or the kernel does not support the size passed in *sigsetsize.*

VERSIONS

C library/kernel differences

The kernel's definition of *sigset_t* differs in size from that used by the C library. In this manual page, the former is referred to as *kernel_sigset_t* (it is nevertheless named *sigset_t* in the kernel sources).

The glibc wrapper function for **sigprocmask**() silently ignores attempts to block the two real-time signals that are used internally by the NPTL threading implementation. See *nptl*(7) for details.

The original Linux system call was named **sigprocmask**(). However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* (referred to as *old_kernel_sigset_t* in this manual page) type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigprocmask**(), was added to support an enlarged *sigset_t* type (referred to as *kernel_sigset_t* in this manual page). The new system call takes a fourth argument, *size_t sigsetsize*, which specifies the size in bytes of the signal sets in *set* and *oldset*. This argument is currently required to have a fixed architecture specific value (equal to *sizeof(kernel_sigset_t)*).

The glibc **sigprocmask**() wrapper function hides these details from us, transparently calling **rt_sigprocmask**() when the kernel provides it.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

It is not possible to block **SIGKILL** or **SIGSTOP**. Attempts to do so are silently ignored.

Each of the threads in a process has its own signal mask.

A child created via *fork*(2) inherits a copy of its parent's signal mask; the signal mask is preserved across *execve*(2).

If **SIGBUS**, **SIGFPE**, **SIGILL**, or **SIGSEGV** are generated while they are blocked, the result is undefined, unless the signal was generated by *kill*(2), *sigqueue*(3), or *raise*(3).

See *sigsetops*(3) for details on manipulating signal sets.

Note that it is permissible (although not very useful) to specify both *set* and *oldset* as NULL.

SEE ALSO

kill(2), *pause*(2), *sigaction*(2), *signal*(2), *sigpending*(2), *sigsuspend*(2), *pthread_sigmask*(3), *sigqueue*(3), *sigsetops*(3), *signal*(7)

NAME

sigreturn, rt_sigreturn − return from signal handler and cleanup stack frame

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

int sigreturn(...);

DESCRIPTION

If the Linux kernel determines that an unblocked signal is pending for a process, then, at the next transition back to user mode in that process (e.g., upon return from a system call or when the process is rescheduled onto the CPU), it creates a new frame on the user-space stack where it saves various pieces of process context (processor status word, registers, signal mask, and signal stack settings).

The kernel also arranges that, during the transition back to user mode, the signal handler is called, and that, upon return from the handler, control passes to a piece of user-space code commonly called the "signal trampoline". The signal trampoline code in turn calls **sigreturn**().

This **sigreturn**() call undoes everything that was done—changing the process's signal mask, switching signal stacks (see **sigaltstack**(2))—in order to invoke the signal handler. Using the information that was earlier saved on the user-space stack **sigreturn**() restores the process's signal mask, switches stacks, and restores the process's context (processor flags and registers, including the stack pointer and instruction pointer), so that the process resumes execution at the point where it was interrupted by the signal.

RETURN VALUE

sigreturn() never returns.

VERSIONS

Many UNIX-type systems have a **sigreturn**() system call or near equivalent. However, this call is not specified in POSIX, and details of its behavior vary across systems.

STANDARDS

None.

NOTES

sigreturn() exists only to allow the implementation of signal handlers. It should **never** be called directly. (Indeed, a simple **sigreturn**() wrapper in the GNU C library simply returns −1, with *errno* set to **ENOSYS**.) Details of the arguments (if any) passed to **sigreturn**() vary depending on the architecture. (On some architectures, such as x86-64, **sigreturn**() takes no arguments, since all of the information that it requires is available in the stack frame that was previously created by the kernel on the user-space stack.)

Once upon a time, UNIX systems placed the signal trampoline code onto the user stack. Nowadays, pages of the user stack are protected so as to disallow code execution. Thus, on contemporary Linux systems, depending on the architecture, the signal trampoline code lives either in the *vdso*(7) or in the C library. In the latter case, the C library's *sigaction*(2) wrapper function informs the kernel of the location of the trampoline code by placing its address in the *sa_restorer* field of the *sigaction* structure, and sets the **SA_RESTORER** flag in the *sa_flags* field.

The saved process context information is placed in a *ucontext_t* structure (see <*sys/ucontext.h*>). That structure is visible within the signal handler as the third argument of a handler established via *sigaction*(2) with the **SA_SIGINFO** flag.

On some other UNIX systems, the operation of the signal trampoline differs a little. In particular, on some systems, upon transitioning back to user mode, the kernel passes control to the trampoline (rather than the signal handler), and the trampoline code calls the signal handler (and then calls **sigreturn**() once the handler returns).

C library/kernel differences

The original Linux system call was named **sigreturn**(). However, with the addition of real-time signals in Linux 2.2, a new system call, **rt_sigreturn**() was added to support an enlarged *sigset_t* type. The GNU C library hides these details from us, transparently employing **rt_sigreturn**() when the kernel provides it.

SEE ALSO

kill(2), *restart_syscall*(2), *sigaltstack*(2), *signal*(2), *getcontext*(3), *signal*(7), *vdso*(7)

NAME

sigsuspend, rt_sigsuspend − wait for a signal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

int sigsuspend(const sigset_t **mask*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigsuspend():
_POSIX_C_SOURCE

DESCRIPTION

sigsuspend() temporarily replaces the signal mask of the calling thread with the mask given by *mask* and then suspends the thread until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

If the signal terminates the process, then **sigsuspend**() does not return. If the signal is caught, then **sigsuspend**() returns after the signal handler returns, and the signal mask is restored to the state before the call to **sigsuspend**().

It is not possible to block **SIGKILL** or **SIGSTOP**; specifying these signals in *mask*, has no effect on the thread's signal mask.

RETURN VALUE

sigsuspend() always returns −1, with *errno* set to indicate the error (normally, **EINTR**).

ERRORS

EFAULT
mask points to memory which is not a valid part of the process address space.

EINTR
The call was interrupted by a signal; *signal*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

C library/kernel differences

The original Linux system call was named **sigsuspend**(). However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigsuspend**(), was added to support an enlarged *sigset_t* type. The new system call takes a second argument, *size_t sigsetsize*, which specifies the size in bytes of the signal set in *mask*. This argument is currently required to have the value *sizeof(sigset_t)* (or the error **EINVAL** results). The glibc **sigsuspend**() wrapper function hides these details from us, transparently calling **rt_sigsuspend**() when the kernel provides it.

NOTES

Normally, **sigsuspend**() is used in conjunction with *sigprocmask*(2) in order to prevent delivery of a signal during the execution of a critical code section. The caller first blocks the signals with *sigprocmask*(2). When the critical code has completed, the caller then waits for the signals by calling **sigsuspend**() with the signal mask that was returned by *sigprocmask*(2) (in the *oldset* argument).

See *sigsetops*(3) for details on manipulating signal sets.

SEE ALSO

kill(2), *pause*(2), *sigaction*(2), *signal*(2), *sigprocmask*(2), *sigwaitinfo*(2), *sigsetops*(3), *sigwait*(3), *signal*(7)

NAME

sigwaitinfo, sigtimedwait, rt_sigtimedwait − synchronously wait for queued signals

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *_Nullable restrict info);
int sigtimedwait(const sigset_t *restrict set,
                 siginfo_t *_Nullable restrict info,
                 const struct timespec *restrict timeout);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigwaitinfo(), **sigtimedwait**():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

sigwaitinfo() suspends execution of the calling thread until one of the signals in *set* is pending (If one of the signals in *set* is already pending for the calling thread, **sigwaitinfo**() will return immediately.)

sigwaitinfo() removes the signal from the set of pending signals and returns the signal number as its function result. If the *info* argument is not NULL, then the buffer that it points to is used to return a structure of type *siginfo_t* (see *sigaction*(2)) containing information about the signal.

If multiple signals in *set* are pending for the caller, the signal that is retrieved by **sigwaitinfo**() is determined according to the usual ordering rules; see *signal*(7) for further details.

sigtimedwait() operates in exactly the same way as **sigwaitinfo**() except that it has an additional argument, *timeout*, which specifies the interval for which the thread is suspended waiting for a signal. (This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the interval may overrun by a small amount.) This argument is a *timespec*(3) structure.

If both fields of this structure are specified as 0, a poll is performed: **sigtimedwait**() returns immediately, either with information about a signal that was pending for the caller, or with an error if none of the signals in *set* was pending.

RETURN VALUE

On success, both **sigwaitinfo**() and **sigtimedwait**() return a signal number (i.e., a value greater than zero). On failure both calls return −1, with *errno* set to indicate the error.

ERRORS

EAGAIN
No signal in *set* became pending within the *timeout* period specified to **sigtimedwait**().

EINTR
The wait was interrupted by a signal handler; see *signal*(7). (This handler was for a signal other than one of those in *set*.)

EINVAL
timeout was invalid.

VERSIONS

C library/kernel differences

On Linux, **sigwaitinfo**() is a library function implemented on top of **sigtimedwait**().

The glibc wrapper functions for **sigwaitinfo**() and **sigtimedwait**() silently ignore attempts to wait for the two real-time signals that are used internally by the NPTL threading implementation. See *nptl*(7) for details.

The original Linux system call was named **sigtimedwait**(). However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit *sigset_t* type supported by that system call was no longer fit for purpose. Consequently, a new system call, **rt_sigtimedwait**(), was added to support an enlarged *sigset_t* type. The new system call takes a fourth argument, *size_t sigsetsize*, which specifies the size in bytes of the signal set in *set*. This argument is currently required to have the value *sizeof(sigset_t)* (or the error **EINVAL** results). The glibc **sigtimedwait**() wrapper function hides these details from us,

transparently calling **rt_sigtimedwait**() when the kernel provides it.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

In normal usage, the calling program blocks the signals in *set* via a prior call to *sigprocmask*(2) (so that the default disposition for these signals does not occur if they become pending between successive calls to **sigwaitinfo**() or *sigtimedwait*()) and does not establish handlers for these signals. In a multithreaded program, the signal should be blocked in all threads, in order to prevent the signal being treated according to its default disposition in a thread other than the one calling **sigwaitinfo**() or *sigtimedwait*())

The set of signals that is pending for a given thread is the union of the set of signals that is pending specifically for that thread and the set of signals that is pending for the process as a whole (see *signal*(7)).

Attempts to wait for **SIGKILL** and **SIGSTOP** are silently ignored.

If multiple threads of a process are blocked waiting for the same signal(s) in **sigwaitinfo**() or **sigtimedwait**(), then exactly one of the threads will actually receive the signal if it becomes pending for the process as a whole; which of the threads receives the signal is indeterminate.

sigwaitinfo() or **sigtimedwait**(), can't be used to receive signals that are synchronously generated, such as the **SIGSEGV** signal that results from accessing an invalid memory address or the **SIGFPE** signal that results from an arithmetic error. Such signals can be caught only via signal handler.

POSIX leaves the meaning of a NULL value for the *timeout* argument of **sigtimedwait**() unspecified, permitting the possibility that this has the same meaning as a call to **sigwaitinfo**(), and indeed this is what is done on Linux.

SEE ALSO

kill(2), *sigaction*(2), *signal*(2), *signalfd*(2), *sigpending*(2), *sigprocmask*(2), *sigqueue*(3), *sigsetops*(3), *sigwait*(3), *timespec*(3), *signal*(7), *time*(7)

NAME

socket − create an endpoint for communication

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

int socket(int *domain***, int** *type***, int** *protocol***);**

DESCRIPTION

socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The *domain* argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <*sys/socket.h*>. The formats currently understood by the Linux kernel include:

Name	Purpose	Man page
AF_UNIX	Local communication	*unix*(7)
AF_LOCAL	Synonym for **AF_UNIX**	
AF_INET	IPv4 Internet protocols	*ip*(7)
AF_AX25	Amateur radio AX.25 protocol	*ax25*(4)
AF_IPX	IPX − Novell protocols	
AF_APPLETALK	AppleTalk	*ddp*(7)
AF_X25	ITU-T X.25 / ISO/IEC 8208 protocol	*x25*(7)
AF_INET6	IPv6 Internet protocols	*ipv6*(7)
AF_DECnet	DECet protocol sockets	
AF_KEY	Key management protocol, originally developed for usage with IPsec	
AF_NETLINK	Kernel user interface device	*netlink*(7)
AF_PACKET	Low-level packet interface	*packet*(7)
AF_RDS	Reliable Datagram Sockets (RDS) protocol	*rds*(7) *rds-rdma*(7)
AF_PPPOX	Generic PPP transport layer, for setting up L2 tunnels (L2TP and PPPoE)	
AF_LLC	Logical link control (IEEE 802.2 LLC) protocol	
AF_IB	InfiniBand native addressing	
AF_MPLS	Multiprotocol Label Switching	
AF_CAN	Controller Area Network automotive bus protocol	
AF_TIPC	TIPC, "cluster domain sockets" protocol	
AF_BLUETOOTH	Bluetooth low-level socket protocol	
AF_ALG	Interface to kernel crypto API	
AF_VSOCK	VSOCK (originally "VMWare VSockets") protocol for hypervisor-guest communication	*vsock*(7)
AF_KCM	KCM (kernel connection multiplexer) interface	
AF_XDP	XDP (express data path) interface	

Further details of the above address families, as well as information on several other address families, can be found in *address_families*(7).

The socket has the indicated *type*, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM
: Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM
: Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET
: Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire

packet with each input system call.

SOCK_RAW Provides raw network protocol access.

SOCK_RDM Provides a reliable datagram layer that does not guarantee ordering.

SOCK_PACKET Obsolete and should not be used in new programs; see *packet*(7).

Some socket types may not be implemented by all protocol families.

Since Linux 2.6.27, the *type* argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of **socket**():

SOCK_NONBLOCK
Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

SOCK_CLOEXEC
Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case *protocol* can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the "communication domain" in which communication is to take place; see *protocols*(5). See *getprotoent*(3) on how to map protocol name strings to protocol numbers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams. They do not preserve record boundaries. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols which implement a **SOCK_STREAM** ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead. When **SO_KEEPALIVE** is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A **SIGPIPE** signal is raised if a process sends or receives on a broken stream; this causes naive processes, which do not handle the signal, to exit. **SOCK_SEQPACKET** sockets employ the same system calls as **SOCK_STREAM** sockets. The only difference is that *read*(2) calls will return only the amount of data requested, and any data remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

SOCK_DGRAM and **SOCK_RAW** sockets allow sending of datagrams to correspondents named in *sendto*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram along with the address of its sender.

SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver. Use *packet*(7) instead.

An *fcntl*(2) **F_SETOWN** operation can be used to specify a process or process group to receive a **SIGURG** signal when the out-of-band data arrives or **SIGPIPE** signal when a **SOCK_STREAM** connection breaks unexpectedly. This operation may also be used to set the process or process group that receives the I/O and asynchronous notification of I/O events via **SIGIO**. Using **F_SETOWN** is equivalent to an *ioctl*(2) call with the **FIOSETOWN** or **SIOCSPGRP** argument.

When the network signals an error condition to the protocol module (e.g., using an ICMP message for IP) the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error. For some protocols it is possible to enable a per-socket error queue to retrieve detailed information about the error; see **IP_RECVERR** in *ip*(7).

The operation of sockets is controlled by socket level *options*. These options are defined in <*sys/socket.h*>. The functions *setsockopt*(2) and *getsockopt*(2) are used to set and get options.

RETURN VALUE

On success, a file descriptor for the new socket is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Permission to create a socket of the specified type and/or protocol is denied.

EAFNOSUPPORT
The implementation does not support the specified address family.

EINVAL
Unknown protocol, or protocol family not available.

EINVAL
Invalid flags in *type*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOBUFS or **ENOMEM**
Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

EPROTONOSUPPORT
The protocol type or the specified protocol is not supported within this domain.

Other errors may be generated by the underlying protocol modules.

STANDARDS

POSIX.1-2008.

SOCK_NONBLOCK and **SOCK_CLOEXEC** are Linux-specific.

HISTORY

POSIX.1-2001, 4.4BSD.

socket() appeared in 4.2BSD. It is generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).

The manifest constants used under 4.x BSD for protocol families are **PF_UNIX**, **PF_INET**, and so on, while **AF_UNIX**, **AF_INET**, and so on are used for address families. However, already the BSD man page promises: "The protocol family generally is the same as the address family", and subsequent standards use AF_* everywhere.

EXAMPLES

An example of the use of **socket**() is shown in *getaddrinfo*(3).

SEE ALSO

accept(2), *bind*(2), *close*(2), *connect*(2), *fcntl*(2), *getpeername*(2), *getsockname*(2), *getsockopt*(2), *ioctl*(2), *listen*(2), *read*(2), *recv*(2), *select*(2), *send*(2), *shutdown*(2), *socketpair*(2), *write*(2), *getprotoent*(3), *address_families*(7), *ip*(7), *socket*(7), *tcp*(7), *udp*(7), *unix*(7)

"An Introductory 4.3BSD Interprocess Communication Tutorial" and "BSD Interprocess Communication Tutorial", reprinted in *UNIX Programmer's Supplementary Documents Volume 1*.

NAME

socketcall − socket system calls

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <linux/net.h>      /* Definition of SYS_* constants */
#include <sys/syscall.h>    /* Definition of SYS_socketcall */
#include <unistd.h>

int syscall(SYS_socketcall, int call, unsigned long *args);
```

Note: glibc provides no wrapper for **socketcall**(), necessitating the use of *syscall*(2).

DESCRIPTION

socketcall() is a common kernel entry point for the socket system calls. *call* determines which socket function to invoke. *args* points to a block containing the actual arguments, which are passed through to the appropriate call.

User programs should call the appropriate functions by their usual names. Only standard library implementors and kernel hackers need to know about **socketcall**().

call	Man page
SYS_SOCKET	*socket*(2)
SYS_BIND	*bind*(2)
SYS_CONNECT	*connect*(2)
SYS_LISTEN	*listen*(2)
SYS_ACCEPT	*accept*(2)
SYS_GETSOCKNAME	*getsockname*(2)
SYS_GETPEERNAME	*getpeername*(2)
SYS_SOCKETPAIR	*socketpair*(2)
SYS_SEND	*send*(2)
SYS_RECV	*recv*(2)
SYS_SENDTO	*sendto*(2)
SYS_RECVFROM	*recvfrom*(2)
SYS_SHUTDOWN	*shutdown*(2)
SYS_SETSOCKOPT	*setsockopt*(2)
SYS_GETSOCKOPT	*getsockopt*(2)
SYS_SENDMSG	*sendmsg*(2)
SYS_RECVMSG	*recvmsg*(2)
SYS_ACCEPT4	*accept4*(2)
SYS_RECVMMSG	*recvmmsg*(2)
SYS_SENDMMSG	*sendmmsg*(2)

VERSIONS

On some architectures—for example, x86-64 and ARM—there is no **socketcall**() system call; instead *socket*(2), *accept*(2), *bind*(2), and so on really are implemented as separate system calls.

STANDARDS

Linux.

On x86-32, **socketcall**() was historically the only entry point for the sockets API. However, starting in Linux 4.3, direct system calls are provided on x86-32 for the sockets API. This facilitates the creation of *seccomp*(2) filters that filter sockets system calls (for new user-space binaries that are compiled to use the new entry points) and also provides a (very) small performance improvement.

SEE ALSO

accept(2), *bind*(2), *connect*(2), *getpeername*(2), *getsockname*(2), *getsockopt*(2), *listen*(2), *recv*(2), *recvfrom*(2), *recvmsg*(2), *send*(2), *sendmsg*(2), *sendto*(2), *setsockopt*(2), *shutdown*(2), *socket*(2), *socketpair*(2)

NAME

socketpair − create a pair of connected sockets

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

int socketpair(int *domain***, int** *type***, int** *protocol***, int** *sv***[2]);**

DESCRIPTION

The **socketpair**() call creates an unnamed pair of connected sockets in the specified *domain*, of the specified *type*, and using the optionally specified *protocol*. For further details of these arguments, see *socket*(2).

The file descriptors used in referencing the new sockets are returned in *sv[0]* and *sv[1]*. The two sockets are indistinguishable.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, *errno* is set to indicate the error, and *sv* is left unchanged

On Linux (and other systems), **socketpair**() does not modify *sv* on failure. A requirement standardizing this behavior was added in POSIX.1-2008 TC2.

ERRORS

EAFNOSUPPORT
The specified address family is not supported on this machine.

EFAULT
The address *sv* does not specify a valid part of the process address space.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

EOPNOTSUPP
The specified protocol does not support creation of socket pairs.

EPROTONOSUPPORT
The specified protocol is not supported on this machine.

VERSIONS

On Linux, the only supported domains for this call are **AF_UNIX** (or synonymously, **AF_LOCAL**) and **AF_TIPC** (since Linux 4.12).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.4BSD.

socketpair() first appeared in 4.2BSD. It is generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).

Since Linux 2.6.27, **socketpair**() supports the **SOCK_NONBLOCK** and **SOCK_CLOEXEC** flags in the *type* argument, as described in *socket*(2).

SEE ALSO

pipe(2), *read*(2), *socket*(2), *write*(2), *socket*(7), *unix*(7)

NAME

splice − splice data to/from a pipe

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>

ssize_t splice(int  fd_in, off_t *_Nullable off_in,
               int  fd_out, off_t *_Nullable off_out,
               size_t len, unsigned int  flags);
```

DESCRIPTION

splice() moves data between two file descriptors without copying between kernel address space and user address space. It transfers up to *len* bytes of data from the file descriptor *fd_in* to the file descriptor *fd_out*, where one of the file descriptors must refer to a pipe.

The following semantics apply for *fd_in* and *off_in*:

- If *fd_in* refers to a pipe, then *off_in* must be NULL.
- If *fd_in* does not refer to a pipe and *off_in* is NULL, then bytes are read from *fd_in* starting from the file offset, and the file offset is adjusted appropriately.
- If *fd_in* does not refer to a pipe and *off_in* is not NULL, then *off_in* must point to a buffer which specifies the starting offset from which bytes will be read from *fd_in*; in this case, the file offset of *fd_in* is not changed.

Analogous statements apply for *fd_out* and *off_out*.

The *flags* argument is a bit mask that is composed by ORing together zero or more of the following values:

SPLICE_F_MOVE
: Attempt to move pages instead of copying. This is only a hint to the kernel: pages may still be copied if the kernel cannot move the pages from the pipe, or if the pipe buffers don't refer to full pages. The initial implementation of this flag was buggy: therefore starting in Linux 2.6.21 it is a no-op (but is still permitted in a **splice**() call); in the future, a correct implementation may be restored.

SPLICE_F_NONBLOCK
: Do not block on I/O. This makes the splice pipe operations nonblocking, but **splice**() may nevertheless block because the file descriptors that are spliced to/from may block (unless they have the **O_NONBLOCK** flag set).

SPLICE_F_MORE
: More data will be coming in a subsequent splice. This is a helpful hint when the *fd_out* refers to a socket (see also the description of **MSG_MORE** in *send*(2), and the description of **TCP_CORK** in *tcp*(7)).

SPLICE_F_GIFT
: Unused for **splice**(); see *vmsplice*(2).

RETURN VALUE

Upon successful completion, **splice**() returns the number of bytes spliced to or from the pipe.

A return value of 0 means end of input. If *fd_in* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

On error, **splice**() returns −1 and *errno* is set to indicate the error.

ERRORS

EAGAIN
: **SPLICE_F_NONBLOCK** was specified in *flags* or one of the file descriptors had been marked as nonblocking (**O_NONBLOCK**), and the operation would block.

EBADF
One or both file descriptors are not valid, or do not have proper read-write mode.

EINVAL
The target filesystem doesn't support splicing.

EINVAL
The target file is opened in append mode.

EINVAL
Neither of the file descriptors refers to a pipe.

EINVAL
An offset was given for nonseekable device (e.g., a pipe).

EINVAL
fd_in and *fd_out* refer to the same pipe.

ENOMEM
Out of memory.

ESPIPE
Either *off_in* or *off_out* was not NULL, but the corresponding file descriptor refers to a pipe.

STANDARDS
Linux.

HISTORY
Linux 2.6.17, glibc 2.5.

In Linux 2.6.30 and earlier, exactly one of *fd_in* and *fd_out* was required to be a pipe. Since Linux 2.6.31, both arguments may refer to pipes.

NOTES
The three system calls **splice**(), *vmsplice*(2), and *tee*(2), provide user-space programs with full control over an arbitrary kernel buffer, implemented within the kernel using the same type of buffer that is used for a pipe. In overview, these system calls perform the following tasks:

splice() moves data from the buffer to an arbitrary file descriptor, or vice versa, or from one buffer to another.

tee(2) "copies" the data from one buffer to another.

vmsplice(2)
"copies" data from user space into the buffer.

Though we talk of copying, actual copies are generally avoided. The kernel does this by implementing a pipe buffer as a set of reference-counted pointers to pages of kernel memory. The kernel creates "copies" of pages in a buffer by creating new pointers (for the output buffer) referring to the pages, and increasing the reference counts for the pages: only pointers are copied, not the pages of the buffer.

_FILE_OFFSET_BITS should be defined to be 64 in code that uses non-null *off_in* or *off_out* or that takes the address of **splice**, if the code is intended to be portable to traditional 32-bit x86 and ARM platforms where **off_t**'s width defaults to 32 bits.

EXAMPLES
See *tee*(2).

SEE ALSO
copy_file_range(2), *sendfile*(2), *tee*(2), *vmsplice*(2), *pipe*(7)

NAME

spu_create − create a new spu context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/spu.h>        /* Definition of SPU_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_spu_create, const char *pathname, unsigned int flags,
        mode_t mode, int neighbor_fd);
```

Note: glibc provides no wrapper for **spu_create**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **spu_create**() system call is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs). It creates a new logical context for an SPU in *pathname* and returns a file descriptor associated with it. *pathname* must refer to a nonexistent directory in the mount point of the SPU filesystem (**spufs**). If **spu_create**() is successful, a directory is created at *pathname* and it is populated with the files described in *spufs*(7).

When a context is created, the returned file descriptor can only be passed to *spu_run*(2), used as the *dirfd* argument to the ***at** family of system calls (e.g., *openat*(2)), or closed; other operations are not defined. A logical SPU context is destroyed (along with all files created within the context's *pathname* directory) once the last reference to the context has gone; this usually occurs when the file descriptor returned by **spu_create**() is closed.

The *mode* argument (minus any bits set in the process's *umask*(2)) specifies the permissions used for creating the new directory in **spufs**. See *stat*(2) for a full list of the possible *mode* values.

The *neighbor_fd* is used only when the **SPU_CREATE_AFFINITY_SPU** flag is specified; see below.

The *flags* argument can be zero or any bitwise OR-ed combination of the following constants:

SPU_CREATE_EVENTS_ENABLED
: Rather than using signals for reporting DMA errors, use the *event* argument to *spu_run*(2).

SPU_CREATE_GANG
: Create an SPU gang instead of a context. (A gang is a group of SPU contexts that are functionally related to each other and which share common scheduling parameters—priority and policy. In the future, gang scheduling may be implemented causing the group to be switched in and out as a single unit.)

  A new directory will be created at the location specified by the *pathname* argument. This gang may be used to hold other SPU contexts, by providing a pathname that is within the gang directory to further calls to **spu_create**().

SPU_CREATE_NOSCHED
: Create a context that is not affected by the SPU scheduler. Once the context is run, it will not be scheduled out until it is destroyed by the creating process.

  Because the context cannot be removed from the SPU, some functionality is disabled for **SPU_CREATE_NOSCHED** contexts. Only a subset of the files will be available in this context directory in **spufs**. Additionally, **SPU_CREATE_NOSCHED** contexts cannot dump a core file when crashing.

  Creating **SPU_CREATE_NOSCHED** contexts requires the **CAP_SYS_NICE** capability.

SPU_CREATE_ISOLATE
: Create an isolated SPU context. Isolated contexts are protected from some PPE (PowerPC Processing Element) operations, such as access to the SPU local store and the NPC register.

  Creating **SPU_CREATE_ISOLATE** contexts also requires the **SPU_CREATE_NOSCHED** flag.

SPU_CREATE_AFFINITY_SPU (since Linux 2.6.23)
Create a context with affinity to another SPU context. This affinity information is used within the SPU scheduling algorithm. Using this flag requires that a file descriptor referring to the other SPU context be passed in the *neighbor_fd* argument.

SPU_CREATE_AFFINITY_MEM (since Linux 2.6.23)
Create a context with affinity to system memory. This affinity information is used within the SPU scheduling algorithm.

RETURN VALUE

On success, **spu_create**() returns a new file descriptor. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
The current user does not have write access to the *spufs*(7) mount point.

EEXIST
An SPU context already exists at the given pathname.

EFAULT
pathname is not a valid string pointer in the calling process's address space.

EINVAL
pathname is not a directory in the *spufs*(7) mount point, or invalid flags have been provided.

ELOOP
Too many symbolic links were found while resolving *pathname*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG
pathname is too long.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
An isolated context was requested, but the hardware does not support SPU isolation.

ENOENT
Part of *pathname* could not be resolved.

ENOMEM
The kernel could not allocate all resources required.

ENOSPC
There are not enough SPU resources available to create a new context or the user-specific limit for the number of SPU contexts has been reached.

ENOSYS
The functionality is not provided by the current system, because either the hardware does not provide SPUs or the spufs module is not loaded.

ENOTDIR
A part of *pathname* is not a directory.

EPERM
The **SPU_CREATE_NOSCHED** flag has been given, but the user does not have the **CAP_SYS_NICE** capability.

FILES

pathname must point to a location beneath the mount point of **spufs**. By convention, it gets mounted in */spu*.

STANDARDS

Linux on PowerPC.

HISTORY

Linux 2.6.16.

Prior to the addition of the **SPU_CREATE_AFFINITY_SPU** flag in Linux 2.6.23, the **spu_create**() system call took only three arguments (i.e., there was no *neighbor_fd* argument).

NOTES

spu_create() is meant to be used from libraries that implement a more abstract interface to SPUs, not to be used from regular applications. See for the recommended libraries.

EXAMPLES

See *spu_run*(2) for an example of the use of **spu_create**()

SEE ALSO

close(2), *spu_run*(2), *capabilities*(7), *spufs*(7)

NAME

spu_run − execute an SPU context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/spu.h> /* Definition of **SPU_*** constants */
#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_spu_run, int *fd***, uint32_t *****npc***, uint32_t *****event***);**

Note: glibc provides no wrapper for **spu_run**(), necessitating the use of *syscall*(2).

DESCRIPTION

The **spu_run**() system call is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs). The *fd* argument is a file descriptor returned by *spu_create*(2) that refers to a specific SPU context. When the context gets scheduled to a physical SPU, it starts execution at the instruction pointer passed in *npc*.

Execution of SPU code happens synchronously, meaning that **spu_run**() blocks while the SPU is still running. If there is a need to execute SPU code in parallel with other code on either the main CPU or other SPUs, a new thread of execution must be created first (e.g., using *pthread_create*(3)).

When **spu_run**() returns, the current value of the SPU program counter is written to *npc*, so successive calls to **spu_run**() can use the same *npc* pointer.

The *event* argument provides a buffer for an extended status code. If the SPU context was created with the **SPU_CREATE_EVENTS_ENABLED** flag, then this buffer is populated by the Linux kernel before **spu_run**() returns.

The status code may be one (or more) of the following constants:

SPE_EVENT_DMA_ALIGNMENT
A DMA alignment error occurred.

SPE_EVENT_INVALID_DMA
An invalid MFC DMA command was attempted.

SPE_EVENT_SPE_DATA_STORAGE
A DMA storage error occurred.

SPE_EVENT_SPE_ERROR
An illegal instruction was executed.

NULL is a valid value for the *event* argument. In this case, the events will not be reported to the calling process.

RETURN VALUE

On success, **spu_run**() returns the value of the *spu_status* register. On failure, it returns −1 and sets *errno* is set to indicate the error.

The *spu_status* register value is a bit mask of status codes and optionally a 14-bit code returned from the **stop-and-signal** instruction on the SPU. The bit masks for the status codes are:

0x02 SPU was stopped by a **stop-and-signal** instruction.

0x04 SPU was stopped by a **halt** instruction.

0x08 SPU is waiting for a channel.

0x10 SPU is in single-step mode.

0x20 SPU has tried to execute an invalid instruction.

0x40 SPU has tried to access an invalid channel.

0x3fff0000
The bits masked with this value contain the code returned from a **stop-and-signal** instruction. These bits are valid only if the 0x02 bit is set.

If **spu_run**() has not returned an error, one or more bits among the lower eight ones are always set.

ERRORS

EBADF
fd is not a valid file descriptor.

EFAULT
npc is not a valid pointer, or *event* is non-NULL and an invalid pointer.

EINTR
A signal occurred while **spu_run**() was in progress; see *signal*(7). The *npc* value has been updated to the new program counter value if necessary.

EINVAL
fd is not a valid file descriptor returned from *spu_create*(2).

ENOMEM
There was not enough memory available to handle a page fault resulting from a Memory Flow Controller (MFC) direct memory access.

ENOSYS
The functionality is not provided by the current system, because either the hardware does not provide SPUs or the spufs module is not loaded.

STANDARDS

Linux on PowerPC.

HISTORY

Linux 2.6.16.

NOTES

spu_run() is meant to be used from libraries that implement a more abstract interface to SPUs, not to be used from regular applications. See for the recommended libraries.

EXAMPLES

The following is an example of running a simple, one-instruction SPU program with the **spu_run**() system call.

```
#include <err.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int       context, fd, spu_status;
    uint32_t  instruction, npc;

    context = syscall(SYS_spu_create, "/spu/example-context", 0, 0755);
    if (context == -1)
        err(EXIT_FAILURE, "spu_create");

    /*
     * Write a 'stop 0x1234' instruction to the SPU's
     * local store memory.
     */
    instruction = 0x00001234;

    fd = open("/spu/example-context/mem", O_RDWR);
    if (fd == -1)
        err(EXIT_FAILURE, "open");
    write(fd, &instruction, sizeof(instruction));

    /*
```

```
             * set npc to the starting instruction address of the
             * SPU program. Since we wrote the instruction at the
             * start of the mem file, the entry point will be 0x0.
             */
            npc = 0;

            spu_status = syscall(SYS_spu_run, context, &npc, NULL);
            if (spu_status == -1)
                err(EXIT_FAILURE, "open");

            /*
             * We should see a status code of 0x12340002:
             *   0x00000002 (spu was stopped due to stop-and-signal)
             * | 0x12340000 (the stop-and-signal code)
             */
            printf("SPU Status: %#08x\n", spu_status);

            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

close(2), *spu_create*(2), *capabilities*(7), *spufs*(7)

NAME

stat, fstat, lstat, fstatat − get file status

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/stat.h>

int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *restrict pathname,
         struct stat *restrict statbuf);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fstatat(int dirfd, const char *restrict pathname,
         struct stat *restrict statbuf, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

lstat():

```
/* Since glibc 2.20 */ _DEFAULT_SOURCE
   || _XOPEN_SOURCE >= 500
   || /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L
   || /* glibc 2.19 and earlier */ _BSD_SOURCE
```

fstatat():

```
Since glibc 2.10:
   _POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
   _ATFILE_SOURCE
```

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *statbuf*. No permissions are required on the file itself, but—in the case of **stat**(), **fstatat**(), and **lstat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() and **fstatat**() retrieve information about the file pointed to by *pathname*; the differences for **fstatat**() are described below.

lstat() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

The stat structure

All of these system calls return a *stat* structure (see *stat*(3type)).

Note: for performance and simplicity reasons, different fields in the *stat* structure may contain state information from different moments during the execution of the system call. For example, if *st_mode* or *st_uid* is changed by another process by calling *chmod*(2) or *chown*(2), **stat**() might return the old *st_mode* together with the new *st_uid*, or the old *st_uid* together with the new *st_mode*.

fstatat()

The **fstatat**() system call is a more general interface for accessing file information which can still provide exactly the behavior of each of **stat**(), **lstat**(), and **fstat**().

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **stat**() and **lstat**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **stat**() and *lstat*())

If *pathname* is absolute, then *dirfd* is ignored.

flags can either be 0, or include one or more of the following flags ORed:

AT_EMPTY_PATH (since Linux 2.6.39)
If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory, and the behavior of **fstatat**() is similar to that of **fstat**(). If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_NO_AUTOMOUNT (since Linux 2.6.38)
Don't automount the terminal ("basename") component of *pathname*. Since Linux 3.1 this flag is ignored. Since Linux 4.11 this flag is implied.

AT_SYMLINK_NOFOLLOW
If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself, like **lstat**(). (By default, **fstatat**() dereferences symbolic links, like *stat*().)

See *openat*(2) for an explanation of the need for **fstatat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Search permission is denied for one of the directories in the path prefix of *pathname*. (See also *path_resolution*(7).)

EBADF
fd is not a valid open file descriptor.

EBADF
(**fstatat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
Bad address.

EINVAL
(**fstatat**()) Invalid flag specified in *flags*.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
pathname is too long.

ENOENT
A component of *pathname* does not exist or is a dangling symbolic link.

ENOENT
pathname is an empty string and **AT_EMPTY_PATH** was not specified in *flags*.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path prefix of *pathname* is not a directory.

ENOTDIR
(**fstatat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EOVERFLOW
pathname or *fd* refers to a file whose size, inode number, or number of blocks cannot be represented in, respectively, the types *off_t*, *ino_t*, or *blkcnt_t*. This error can occur when, for example, an application compiled on a 32-bit platform without *−D_FILE_OFFSET_BITS=64* calls **stat**() on a file whose size exceeds *(1<<31)−1* bytes.

STANDARDS

POSIX.1-2008.

HISTORY

stat()
fstat()
lstat() SVr4, 4.3BSD, POSIX.1-2001.

fstatat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

According to POSIX.1-2001, **lstat**() on a symbolic link need return valid information only in the *st_size* field and the file type of the *st_mode* field of the *stat* structure. POSIX.1-2008 tightens the specification, requiring **lstat**() to return valid information in all fields except the mode bits in *st_mode*.

Use of the *st_blocks* and *st_blksize* fields may be less portable. (They were introduced in BSD. The interpretation differs between systems, and possibly on a single system when NFS mounts are involved.)

C library/kernel differences

Over time, increases in the size of the *stat* structure have led to three successive versions of **stat**(): *sys_stat*() (slot *__NR_oldstat*), *sys_newstat*() (slot *__NR_stat*), and *sys_stat64()* (slot *__NR_stat64*) on 32-bit platforms such as i386. The first two versions were already present in Linux 1.0 (albeit with different names); the last was added in Linux 2.4. Similar remarks apply for **fstat**() and **lstat**().

The kernel-internal versions of the *stat* structure dealt with by the different versions are, respectively:

__old_kernel_stat
The original structure, with rather narrow fields, and no padding.

stat Larger *st_ino* field and padding added to various parts of the structure to allow for future expansion.

stat64 Even larger *st_ino* field, larger *st_uid* and *st_gid* fields to accommodate the Linux-2.4 expansion of UIDs and GIDs to 32 bits, and various other enlarged fields and further padding in the structure. (Various padding bytes were eventually consumed in Linux 2.6, with the advent of 32-bit device IDs and nanosecond components for the timestamp fields.)

The glibc **stat**() wrapper function hides these details from applications, invoking the most recent version of the system call provided by the kernel, and repacking the returned information if required for old binaries.

On modern 64-bit systems, life is simpler: there is a single **stat**() system call and the kernel deals with a *stat* structure that contains fields of a sufficient size.

The underlying system call employed by the glibc **fstatat**() wrapper function is actually called **fstatat64**() or, on some architectures, **newfstatat**().

EXAMPLES

The following program calls **lstat**() and displays selected fields in the returned *stat* structure.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (lstat(argv[1], &sb) == -1) {
```

```
            perror("lstat");
            exit(EXIT_FAILURE);
        }

        printf("ID of containing device:  [%x,%x]\n",
               major(sb.st_dev),
               minor(sb.st_dev));

        printf("File type:                ");

        switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:  printf("block device\n");            break;
        case S_IFCHR:  printf("character device\n");        break;
        case S_IFDIR:  printf("directory\n");               break;
        case S_IFIFO:  printf("FIFO/pipe\n");               break;
        case S_IFLNK:  printf("symlink\n");                 break;
        case S_IFREG:  printf("regular file\n");            break;
        case S_IFSOCK: printf("socket\n");                  break;
        default:       printf("unknown?\n");                break;
        }

        printf("I-node number:            %ju\n", (uintmax_t) sb.st_ino);

        printf("Mode:                     %jo (octal)\n",
               (uintmax_t) sb.st_mode);

        printf("Link count:               %ju\n", (uintmax_t) sb.st_nlink);
        printf("Ownership:                UID=%ju   GID=%ju\n",
               (uintmax_t) sb.st_uid, (uintmax_t) sb.st_gid);

        printf("Preferred I/O block size: %jd bytes\n",
               (intmax_t) sb.st_blksize);
        printf("File size:                %jd bytes\n",
               (intmax_t) sb.st_size);
        printf("Blocks allocated:         %jd\n",
               (intmax_t) sb.st_blocks);

        printf("Last status change:       %s", ctime(&sb.st_ctime));
        printf("Last file access:         %s", ctime(&sb.st_atime));
        printf("Last file modification:   %s", ctime(&sb.st_mtime));

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ls(1), *stat*(1), *access*(2), *chmod*(2), *chown*(2), *readlink*(2), *statx*(2), *utime*(2), *stat*(3type), *capabilities*(7), *inode*(7), *symlink*(7)

NAME

statfs, fstatfs − get filesystem statistics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/vfs.h> /* or <sys/statfs.h> */

int statfs(const char * *path*, **struct statfs *** *buf* **);**
int fstatfs(int *fd*, **struct statfs *** *buf* **);**

Unless you need the *f_type* field, you should use the standard *statvfs*(3) interface instead.

DESCRIPTION

The **statfs**() system call returns information about a mounted filesystem. *path* is the pathname of any file within the mounted filesystem. *buf* is a pointer to a *statfs* structure defined approximately as follows:

```
struct statfs {
    __fsword_t f_type;    /* Type of filesystem (see below) */
    __fsword_t f_bsize;   /* Optimal transfer block size */
    fsblkcnt_t f_blocks;  /* Total data blocks in filesystem */
    fsblkcnt_t f_bfree;   /* Free blocks in filesystem */
    fsblkcnt_t f_bavail;  /* Free blocks available to
                             unprivileged user */
    fsfilcnt_t f_files;   /* Total inodes in filesystem */
    fsfilcnt_t f_ffree;   /* Free inodes in filesystem */
    fsid_t     f_fsid;    /* Filesystem ID */
    __fsword_t f_namelen; /* Maximum length of filenames */
    __fsword_t f_frsize;  /* Fragment size (since Linux 2.6) */
    __fsword_t f_flags;   /* Mount flags of filesystem
                             (since Linux 2.6.36) */
    __fsword_t f_spare[xxx];
                     /* Padding bytes reserved for future use */
};
```

The following filesystem types may appear in *f_type*:

```
ADFS_SUPER_MAGIC      0xadf5
AFFS_SUPER_MAGIC      0xadff
AFS_SUPER_MAGIC       0x5346414f
ANON_INODE_FS_MAGIC   0x09041934 /* Anonymous inode FS (for
                                    pseudofiles that have no name;
                                    e.g., epoll, signalfd, bpf) */
AUTOFS_SUPER_MAGIC    0x0187
BDEVFS_MAGIC          0x62646576
BEFS_SUPER_MAGIC      0x42465331
BFS_MAGIC             0x1badface
BINFMTFS_MAGIC        0x42494e4d
BPF_FS_MAGIC          0xcafe4a11
BTRFS_SUPER_MAGIC     0x9123683e
BTRFS_TEST_MAGIC      0x73727279
CGROUP_SUPER_MAGIC    0x27e0eb   /* Cgroup pseudo FS */
CGROUP2_SUPER_MAGIC   0x63677270 /* Cgroup v2 pseudo FS */
CIFS_MAGIC_NUMBER     0xff534d42
CODA_SUPER_MAGIC      0x73757245
COH_SUPER_MAGIC       0x012ff7b7
CRAMFS_MAGIC          0x28cd3d45
DEBUGFS_MAGIC         0x64626720
DEVFS_SUPER_MAGIC     0x1373     /* Linux 2.6.17 and earlier */
DEVPTS_SUPER_MAGIC    0x1cd1
ECRYPTFS_SUPER_MAGIC  0xf15f
```

```
EFIVARFS_MAGIC        0xde5e81e4
EFS_SUPER_MAGIC       0x00414a53
EXT_SUPER_MAGIC       0x137d     /* Linux 2.0 and earlier */
EXT2_OLD_SUPER_MAGIC  0xef51
EXT2_SUPER_MAGIC      0xef53
EXT3_SUPER_MAGIC      0xef53
EXT4_SUPER_MAGIC      0xef53
F2FS_SUPER_MAGIC      0xf2f52010
FUSE_SUPER_MAGIC      0x65735546
FUTEXFS_SUPER_MAGIC   0xbad1dea  /* Unused */
HFS_SUPER_MAGIC       0x4244
HOSTFS_SUPER_MAGIC    0x00c0ffee
HPFS_SUPER_MAGIC      0xf995e849
HUGETLBFS_MAGIC       0x958458f6
ISOFS_SUPER_MAGIC     0x9660
JFFS2_SUPER_MAGIC     0x72b6
JFS_SUPER_MAGIC       0x3153464a
MINIX_SUPER_MAGIC     0x137f     /* original minix FS */
MINIX_SUPER_MAGIC2    0x138f     /* 30 char minix FS */
MINIX2_SUPER_MAGIC    0x2468     /* minix V2 FS */
MINIX2_SUPER_MAGIC2   0x2478     /* minix V2 FS, 30 char names */
MINIX3_SUPER_MAGIC    0x4d5a     /* minix V3 FS, 60 char names */
MQUEUE_MAGIC          0x19800202 /* POSIX message queue FS */
MSDOS_SUPER_MAGIC     0x4d44
MTD_INODE_FS_MAGIC    0x11307854
NCP_SUPER_MAGIC       0x564c
NFS_SUPER_MAGIC       0x6969
NILFS_SUPER_MAGIC     0x3434
NSFS_MAGIC            0x6e736673
NTFS_SB_MAGIC         0x5346544e
OCFS2_SUPER_MAGIC     0x7461636f
OPENPROM_SUPER_MAGIC  0x9fa1
OVERLAYFS_SUPER_MAGIC 0x794c7630
PIPEFS_MAGIC          0x50495045
PROC_SUPER_MAGIC      0x9fa0     /* /proc FS */
PSTOREFS_MAGIC        0x6165676c
QNX4_SUPER_MAGIC      0x002f
QNX6_SUPER_MAGIC      0x68191122
RAMFS_MAGIC           0x858458f6
REISERFS_SUPER_MAGIC  0x52654973
ROMFS_MAGIC           0x7275
SECURITYFS_MAGIC      0x73636673
SELINUX_MAGIC         0xf97cff8c
SMACK_MAGIC           0x43415d53
SMB_SUPER_MAGIC       0x517b
SMB2_MAGIC_NUMBER     0xfe534d42
SOCKFS_MAGIC          0x534f434b
SQUASHFS_MAGIC        0x73717368
SYSFS_MAGIC           0x62656572
SYSV2_SUPER_MAGIC     0x012ff7b6
SYSV4_SUPER_MAGIC     0x012ff7b5
TMPFS_MAGIC           0x01021994
TRACEFS_MAGIC         0x74726163
UDF_SUPER_MAGIC       0x15013346
UFS_MAGIC             0x00011954
USBDEVICE_SUPER_MAGIC 0x9fa2
V9FS_MAGIC            0x01021997
VXFS_SUPER_MAGIC      0xa501fcf5
XENFS_SUPER_MAGIC     0xabba1974
```

```
XENIX_SUPER_MAGIC     0x012ff7b4
XFS_SUPER_MAGIC       0x58465342
_XIAFS_SUPER_MAGIC    0x012fd16d /* Linux 2.0 and earlier */
```

Most of these MAGIC constants are defined in */usr/include/linux/magic.h*, and some are hardcoded in kernel sources.

The *f_flags* field is a bit mask indicating mount options for the filesystem. It contains zero or more of the following bits:

ST_MANDLOCK
Mandatory locking is permitted on the filesystem (see *fcntl*(2)).

ST_NOATIME
Do not update access times; see *mount*(2).

ST_NODEV
Disallow access to device special files on this filesystem.

ST_NODIRATIME
Do not update directory access times; see *mount*(2).

ST_NOEXEC
Execution of programs is disallowed on this filesystem.

ST_NOSUID
The set-user-ID and set-group-ID bits are ignored by *exec*(3) for executable files on this filesystem

ST_RDONLY
This filesystem is mounted read-only.

ST_RELATIME
Update atime relative to mtime/ctime; see *mount*(2).

ST_SYNCHRONOUS
Writes are synched to the filesystem immediately (see the description of **O_SYNC** in *open*(2)).

ST_NOSYMFOLLOW (since Linux 5.10)
Symbolic links are not followed when resolving paths; see *mount*(2).

Nobody knows what *f_fsid* is supposed to contain (but see below).

Fields that are undefined for a particular filesystem are set to 0.

fstatfs() returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
(**statfs**()) Search permission is denied for a component of the path prefix of *path*. (See also *path_resolution*(7).)

EBADF
(**fstatfs**()) *fd* is not a valid open file descriptor.

EFAULT
buf or *path* points to an invalid address.

EINTR
The call was interrupted by a signal; see *signal*(7).

EIO An I/O error occurred while reading from the filesystem.

ELOOP
(**statfs**()) Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG
(**statfs**()) *path* is too long.

ENOENT
(**statfs**()) The file referred to by *path* does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOSYS
The filesystem does not support this call.

ENOTDIR
(**statfs**()) A component of the path prefix of *path* is not a directory.

EOVERFLOW
Some values were too large to be represented in the returned struct.

VERSIONS

The f_fsid field

Solaris, Irix, and POSIX have a system call *statvfs*(2) that returns a *struct statvfs* (defined in <*sys/statvfs.h*>) containing an *unsigned long f_fsid*. Linux, SunOS, HP-UX, 4.4BSD have a system call **statfs**() that returns a *struct statfs* (defined in <*sys/vfs.h*>) containing a *fsid_t f_fsid*, where *fsid_t* is defined as *struct { int val[2]; }*. The same holds for FreeBSD, except that it uses the include file <*sys/mount.h*>.

The general idea is that *f_fsid* contains some random stuff such that the pair (*f_fsid*,*ino*) uniquely determines a file. Some operating systems use (a variation on) the device number, or the device number combined with the filesystem type. Several operating systems restrict giving out the *f_fsid* field to the superuser only (and zero it for unprivileged users), because this field is used in the filehandle of the filesystem when NFS-exported, and giving it out is a security concern.

Under some operating systems, the *fsid* can be used as the second argument to the *sysfs*(2) system call.

STANDARDS

Linux.

HISTORY

The Linux **statfs**() was inspired by the 4.4BSD one (but they do not use the same structure).

The original Linux **statfs**() and **fstatfs**() system calls were not designed with extremely large file sizes in mind. Subsequently, Linux 2.6 added new **statfs64**() and **fstatfs64**() system calls that employ a new structure, *statfs64*. The new structure contains the same fields as the original *statfs* structure, but the sizes of various fields are increased, to accommodate large file sizes. The glibc **statfs**() and **fstatfs**() wrapper functions transparently deal with the kernel differences.

LSB has deprecated the library calls **statfs**() and **fstatfs**() and tells us to use *statvfs*(3) and *fstatvfs*(3) instead.

NOTES

The *__fsword_t* type used for various fields in the *statfs* structure definition is a glibc internal type, not intended for public use. This leaves the programmer in a bit of a conundrum when trying to copy or compare these fields to local variables in a program. Using *unsigned int* for such variables suffices on most systems.

Some systems have only <*sys/vfs.h*>, other systems also have <*sys/statfs.h*>, where the former includes the latter. So it seems including the former is the best choice.

BUGS

From Linux 2.6.38 up to and including Linux 3.1, **fstatfs**() failed with the error **ENOSYS** for file descriptors created by *pipe*(2).

SEE ALSO

stat(2), *statvfs*(3), *path_resolution*(7)

NAME

statx − get file status (extended)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>          /* Definition of AT_* constants */
#include <sys/stat.h>

int statx(int dirfd, const char *restrict pathname, int flags,
          unsigned int mask, struct statx *restrict statxbuf);
```

DESCRIPTION

This function returns information about a file, storing it in the buffer pointed to by *statxbuf*. The returned buffer is a structure of the following type:

```
struct statx {
    __u32 stx_mask;        /* Mask of bits indicating
                              filled fields */
    __u32 stx_blksize;     /* Block size for filesystem I/O */
    __u64 stx_attributes;  /* Extra file attribute indicators */
    __u32 stx_nlink;       /* Number of hard links */
    __u32 stx_uid;         /* User ID of owner */
    __u32 stx_gid;         /* Group ID of owner */
    __u16 stx_mode;        /* File type and mode */
    __u64 stx_ino;         /* Inode number */
    __u64 stx_size;        /* Total size in bytes */
    __u64 stx_blocks;      /* Number of 512B blocks allocated */
    __u64 stx_attributes_mask;
                           /* Mask to show what's supported
                              in stx_attributes */

    /* The following fields are file timestamps */
    struct statx_timestamp stx_atime;  /* Last access */
    struct statx_timestamp stx_btime;  /* Creation */
    struct statx_timestamp stx_ctime;  /* Last status change */
    struct statx_timestamp stx_mtime;  /* Last modification */

    /* If this file represents a device, then the next two
       fields contain the ID of the device */
    __u32 stx_rdev_major;  /* Major ID */
    __u32 stx_rdev_minor;  /* Minor ID */

    /* The next two fields contain the ID of the device
       containing the filesystem where the file resides */
    __u32 stx_dev_major;   /* Major ID */
    __u32 stx_dev_minor;   /* Minor ID */

    __u64 stx_mnt_id;      /* Mount ID */

    /* Direct I/O alignment restrictions */
    __u32 stx_dio_mem_align;
    __u32 stx_dio_offset_align;
};
```

The file timestamps are structures of the following type:

```
struct statx_timestamp {
    __s64 tv_sec;    /* Seconds since the Epoch (UNIX time) */
    __u32 tv_nsec;   /* Nanoseconds since tv_sec */
};
```

(Note that reserved space and padding is omitted.)

Invoking statx():

To access a file's status, no permissions are required on the file itself, but in the case of **statx**() with a pathname, execute (search) permission is required on all of the directories in *pathname* that lead to the file.

statx() uses *pathname*, *dirfd*, and *flags* to identify the target file in one of the following ways:

An absolute pathname

If *pathname* begins with a slash, then it is an absolute pathname that identifies the target file. In this case, *dirfd* is ignored.

A relative pathname

If *pathname* is a string that begins with a character other than a slash and *dirfd* is **AT_FDCWD**, then *pathname* is a relative pathname that is interpreted relative to the process's current working directory.

A directory-relative pathname

If *pathname* is a string that begins with a character other than a slash and *dirfd* is a file descriptor that refers to a directory, then *pathname* is a relative pathname that is interpreted relative to the directory referred to by *dirfd*. (See *openat*(2) for an explanation of why this is useful.)

By file descriptor

If *pathname* is an empty string and the **AT_EMPTY_PATH** flag is specified in *flags* (see below), then the target file is the one referred to by the file descriptor *dirfd*.

flags can be used to influence a pathname-based lookup. A value for *flags* is constructed by ORing together zero or more of the following constants:

AT_EMPTY_PATH

If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory.

If *dirfd* is **AT_FDCWD**, the call operates on the current working directory.

AT_NO_AUTOMOUNT

Don't automount the terminal ("basename") component of *pathname* if it is a directory that is an automount point. This allows the caller to gather attributes of an automount point (rather than the location it would mount). This flag has no effect if the mount point has already been mounted over.

The **AT_NO_AUTOMOUNT** flag can be used in tools that scan directories to prevent mass-automounting of a directory of automount points.

All of *stat*(2), *lstat*(2), and *fstatat*(2) act as though **AT_NO_AUTOMOUNT** was set.

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself, like *lstat*(2).

flags can also be used to control what sort of synchronization the kernel will do when querying a file on a remote filesystem. This is done by ORing in one of the following values:

AT_STATX_SYNC_AS_STAT

Do whatever *stat*(2) does. This is the default and is very much filesystem-specific.

AT_STATX_FORCE_SYNC

Force the attributes to be synchronized with the server. This may require that a network filesystem perform a data writeback to get the timestamps correct.

AT_STATX_DONT_SYNC

Don't synchronize anything, but rather just take whatever the system has cached if possible. This may mean that the information returned is approximate, but, on a network filesystem, it may not involve a round trip to the server - even if no lease is held.

The *mask* argument to **statx**() is used to tell the kernel which fields the caller is interested in. *mask* is

an ORed combination of the following constants:

STATX_TYPE	Want stx_mode & S_IFMT
STATX_MODE	Want stx_mode & ~S_IFMT
STATX_NLINK	Want stx_nlink
STATX_UID	Want stx_uid
STATX_GID	Want stx_gid
STATX_ATIME	Want stx_atime
STATX_MTIME	Want stx_mtime
STATX_CTIME	Want stx_ctime
STATX_INO	Want stx_ino
STATX_SIZE	Want stx_size
STATX_BLOCKS	Want stx_blocks
STATX_BASIC_STATS	[All of the above]
STATX_BTIME	Want stx_btime
STATX_ALL	The same as STATX_BASIC_STATS | STATX_BTIME. It is deprecated and should not be used.
STATX_MNT_ID	Want stx_mnt_id (since Linux 5.8)
STATX_DIOALIGN	Want stx_dio_mem_align and stx_dio_offset_align (since Linux 6.1; support varies by filesystem)

Note that, in general, the kernel does *not* reject values in *mask* other than the above. (For an exception, see **EINVAL** in errors.) Instead, it simply informs the caller which values are supported by this kernel and filesystem via the *statx.stx_mask* field. Therefore, *do not* simply set *mask* to **UINT_MAX** (all bits set), as one or more bits may, in the future, be used to specify an extension to the buffer.

The returned information

The status information for the target file is returned in the *statx* structure pointed to by *statxbuf*. Included in this is *stx_mask* which indicates what other information has been returned. *stx_mask* has the same format as the *mask* argument and bits are set in it to indicate which fields have been filled in.

It should be noted that the kernel may return fields that weren't requested and may fail to return fields that were requested, depending on what the backing filesystem supports. (Fields that are given values despite being unrequested can just be ignored.) In either case, *stx_mask* will not be equal *mask*.

If a filesystem does not support a field or if it has an unrepresentable value (for instance, a file with an exotic type), then the mask bit corresponding to that field will be cleared in *stx_mask* even if the user asked for it and a dummy value will be filled in for compatibility purposes if one is available (e.g., a dummy UID and GID may be specified to mount under some circumstances).

A filesystem may also fill in fields that the caller didn't ask for if it has values for them available and the information is available at no extra cost. If this happens, the corresponding bits will be set in *stx_mask*.

Note: for performance and simplicity reasons, different fields in the *statx* structure may contain state information from different moments during the execution of the system call. For example, if *stx_mode* or *stx_uid* is changed by another process by calling *chmod*(2) or *chown*(2), **stat**() might return the old *stx_mode* together with the new *stx_uid*, or the old *stx_uid* together with the new *stx_mode*.

Apart from *stx_mask* (which is described above), the fields in the *statx* structure are:

stx_blksize
: The "preferred" block size for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

stx_attributes
: Further status information about the file (see below for more information).

stx_nlink
: The number of hard links on a file.

stx_uid This field contains the user ID of the owner of the file.

stx_gid This field contains the ID of the group owner of the file.

stx_mode
: The file type and mode. See *inode*(7) for details.

stx_ino The inode number of the file.

stx_size
: The size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

stx_blocks
: The number of blocks allocated to the file on the medium, in 512-byte units. (This may be smaller than *stx_size*/512 when the file has holes.)

stx_attributes_mask
: A mask indicating which bits in *stx_attributes* are supported by the VFS and the filesystem.

stx_atime
: The file's last access timestamp.

stx_btime
: The file's creation timestamp.

stx_ctime
: The file's last status change timestamp.

stx_mtime
: The file's last modification timestamp.

stx_dev_major and *stx_dev_minor*
: The device on which this file (inode) resides.

stx_rdev_major and *stx_rdev_minor*
: The device that this file (inode) represents if the file is of block or character device type.

stx_mnt_id
: The mount ID of the mount containing the file. This is the same number reported by *name_to_handle_at*(2) and corresponds to the number in the first field in one of the records in */proc/self/mountinfo*.

stx_dio_mem_align
: The alignment (in bytes) required for user memory buffers for direct I/O (**O_DIRECT**) on this file, or 0 if direct I/O is not supported on this file.

  STATX_DIOALIGN (*stx_dio_mem_align* and *stx_dio_offset_align*) is supported on block devices since Linux 6.1. The support on regular files varies by filesystem; it is supported by ext4, f2fs, and xfs since Linux 6.1.

stx_dio_offset_align
: The alignment (in bytes) required for file offsets and I/O segment lengths for direct I/O (**O_DIRECT**) on this file, or 0 if direct I/O is not supported on this file. This will only be nonzero if *stx_dio_mem_align* is nonzero, and vice versa.

For further information on the above fields, see *inode*(7).

File attributes

The *stx_attributes* field contains a set of ORed flags that indicate additional attributes of the file. Note that any attribute that is not indicated as supported by *stx_attributes_mask* has no usable value here. The bits in *stx_attributes_mask* correspond bit-by-bit to *stx_attributes*.

The flags are as follows:

STATX_ATTR_COMPRESSED
: The file is compressed by the filesystem and may take extra resources to access.

STATX_ATTR_IMMUTABLE
: The file cannot be modified: it cannot be deleted or renamed, no hard links can be created to this file and no data can be written to it. See *chattr*(1)

STATX_ATTR_APPEND
: The file can only be opened in append mode for writing. Random access writing is not permitted. See *chattr*(1)

STATX_ATTR_NODUMP
File is not a candidate for backup when a backup program such as *dump*(8) is run. See *chattr*(1)

STATX_ATTR_ENCRYPTED
A key is required for the file to be encrypted by the filesystem.

STATX_ATTR_VERITY (since Linux 5.5)
The file has fs-verity enabled. It cannot be written to, and all reads from it will be verified against a cryptographic hash that covers the entire file (e.g., via a Merkle tree).

STATX_ATTR_DAX (since Linux 5.8)
The file is in the DAX (cpu direct access) state. DAX state attempts to minimize software cache effects for both I/O and memory mappings of this file. It requires a file system which has been configured to support DAX.

DAX generally assumes all accesses are via CPU load / store instructions which can minimize overhead for small accesses, but may adversely affect CPU utilization for large transfers.

File I/O is done directly to/from user-space buffers and memory mapped I/O may be performed with direct memory mappings that bypass the kernel page cache.

While the DAX property tends to result in data being transferred synchronously, it does not give the same guarantees as the **O_SYNC** flag (see *open*(2)), where data and the necessary metadata are transferred together.

A DAX file may support being mapped with the **MAP_SYNC** flag, which enables a program to use CPU cache flush instructions to persist CPU store operations without an explicit *fsync*(2). See *mmap*(2) for more information.

STATX_ATTR_MOUNT_ROOT (since Linux 5.8)
The file is the root of a mount.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Search permission is denied for one of the directories in the path prefix of *pathname*. (See also *path_resolution*(7).)

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
pathname or *statxbuf* is NULL or points to a location outside the process's accessible address space.

EINVAL
Invalid flag specified in *flags*.

EINVAL
Reserved flag specified in *mask*. (Currently, there is one such flag, designated by the constant **STATX__RESERVED**, with the value 0x80000000U.)

ELOOP
Too many symbolic links encountered while traversing the pathname.

ENAMETOOLONG
pathname is too long.

ENOENT
A component of *pathname* does not exist, or *pathname* is an empty string and **AT_EMPTY_PATH** was not specified in *flags*.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path prefix of *pathname* is not a directory or *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

STANDARDS
Linux.

HISTORY
Linux 4.11, glibc 2.28.

SEE ALSO
ls(1), *stat*(1), *access*(2), *chmod*(2), *chown*(2), *name_to_handle_at*(2), *readlink*(2), *stat*(2), *utime*(2), *proc*(5), *capabilities*(7), *inode*(7), *symlink*(7)

NAME

stime − set time

SYNOPSIS

#include <time.h>

[[deprecated]] int stime(const time_t *_t_);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

stime():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

NOTE: This function is deprecated; use *clock_settime*(2) instead.

stime() sets the system's idea of the time and date. The time, pointed to by *t*, is measured in seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). **stime**() may be executed only by the superuser.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
Error in getting information from user space.

EPERM
The calling process has insufficient privilege. Under Linux, the **CAP_SYS_TIME** privilege is required.

STANDARDS

None.

HISTORY

SVr4.

Starting with glibc 2.31, this function is no longer available to newly linked applications and is no longer declared in <*time.h*>.

SEE ALSO

date(1), *settimeofday*(2), *capabilities*(7)

NAME

subpage_prot − define a subpage protection for an address range

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_subpage_prot, unsigned long addr, unsigned long len,
        uint32_t *map);
```

Note: glibc provides no wrapper for **subpage_prot**(), necessitating the use of *syscall*(2).

DESCRIPTION

The PowerPC-specific **subpage_prot**() system call provides the facility to control the access permissions on individual 4 kB subpages on systems configured with a page size of 64 kB.

The protection map is applied to the memory pages in the region starting at *addr* and continuing for *len* bytes. Both of these arguments must be aligned to a 64-kB boundary.

The protection map is specified in the buffer pointed to by *map*. The map has 2 bits per 4 kB subpage; thus each 32-bit word specifies the protections of 16 4 kB subpages inside a 64 kB page (so, the number of 32-bit words pointed to by *map* should equate to the number of 64-kB pages specified by *len*). Each 2-bit field in the protection map is either 0 to allow any access, 1 to prevent writes, or 2 or 3 to prevent all accesses.

RETURN VALUE

On success, **subpage_prot**() returns 0. Otherwise, one of the error codes specified below is returned.

ERRORS

EFAULT
: The buffer referred to by *map* is not accessible.

EINVAL
: The *addr* or *len* arguments are incorrect. Both of these arguments must be aligned to a multiple of the system page size, and they must not refer to a region outside of the address space of the process or to a region that consists of huge pages.

ENOMEM
: Out of memory.

STANDARDS

Linux.

HISTORY

Linux 2.6.25 (PowerPC).

The system call is provided only if the kernel is configured with **CONFIG_PPC_64K_PAGES**.

NOTES

Normal page protections (at the 64-kB page level) also apply; the subpage protection mechanism is an additional constraint, so putting 0 in a 2-bit field won't allow writes to a page that is otherwise write-protected.

Rationale

This system call is provided to assist writing emulators that operate using 64-kB pages on PowerPC systems. When emulating systems such as x86, which uses a smaller page size, the emulator can no longer use the memory-management unit (MMU) and normal system calls for controlling page protections. (The emulator could emulate the MMU by checking and possibly remapping the address for each memory access in software, but that is slow.) The idea is that the emulator supplies an array of protection masks to apply to a specified range of virtual addresses. These masks are applied at the level where hardware page-table entries (PTEs) are inserted into the hardware page table based on the Linux PTEs, so the Linux PTEs are not affected. Implicit in this is that the regions of the address space that are protected are switched to use 4-kB hardware pages rather than 64-kB hardware pages (on machines with hardware 64-kB page support).

SEE ALSO

mprotect(2), *syscall*(2)

Documentation/admin−guide/mm/hugetlbpage.rst in the Linux kernel source tree

NAME

swapon, swapoff − start/stop swapping to file/device

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/swap.h>

int swapon(const char *path, int swapflags);
int swapoff(const char *path);
```

DESCRIPTION

swapon() sets the swap area to the file or block device specified by *path*. **swapoff**() stops swapping to the file or block device specified by *path*.

If the **SWAP_FLAG_PREFER** flag is specified in the **swapon**() *swapflags* argument, the new swap area will have a higher priority than default. The priority is encoded within *swapflags* as:

```
(prio << SWAP_FLAG_PRIO_SHIFT) & SWAP_FLAG_PRIO_MASK
```

If the **SWAP_FLAG_DISCARD** flag is specified in the **swapon**() *swapflags* argument, freed swap pages will be discarded before they are reused, if the swap device supports the discard or trim operation. (This may improve performance on some Solid State Devices, but often it does not.) See also NOTES.

These functions may be used only by a privileged process (one having the **CAP_SYS_ADMIN** capability).

Priority

Each swap area has a priority, either high or low. The default priority is low. Within the low-priority areas, newer areas are even lower priority than older areas.

All priorities set with *swapflags* are high-priority, higher than default. They may have any nonnegative value chosen by the caller. Higher numbers mean higher priority.

Swap pages are allocated from areas in priority order, highest priority first. For areas with different priorities, a higher-priority area is exhausted before using a lower-priority area. If two or more areas have the same priority, and it is the highest priority available, pages are allocated on a round-robin basis between them.

As of Linux 1.3.6, the kernel usually follows these rules, but there are exceptions.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBUSY
(for *swapon*()) The specified *path* is already being used as a swap area.

EINVAL
The file *path* exists, but refers neither to a regular file nor to a block device;

EINVAL
(**swapon**()) The indicated path does not contain a valid swap signature or resides on an in-memory filesystem such as *tmpfs*(5).

EINVAL (since Linux 3.4)
(**swapon**()) An invalid flag value was specified in *swapflags*.

EINVAL
(**swapoff**()) *path* is not currently a swap area.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
The file *path* does not exist.

ENOMEM
The system has insufficient memory to start swapping.

EPERM
The caller does not have the **CAP_SYS_ADMIN** capability. Alternatively, the maximum number of swap files are already in use; see NOTES below.

STANDARDS
Linux.

HISTORY
The *swapflags* argument was introduced in Linux 1.3.2.

NOTES
The partition or path must be prepared with *mkswap*(8)

There is an upper limit on the number of swap files that may be used, defined by the kernel constant **MAX_SWAPFILES**. Before Linux 2.4.10, **MAX_SWAPFILES** has the value 8; since Linux 2.4.10, it has the value 32. Since Linux 2.6.18, the limit is decreased by 2 (thus 30), since Linux 5.19, the limit is decreased by 3 (thus: 29) if the kernel is built with the **CONFIG_MIGRATION** option (which reserves two swap table entries for the page migration features of *mbind*(2) and *migrate_pages*(2)). Since Linux 2.6.32, the limit is further decreased by 1 if the kernel is built with the **CONFIG_MEMORY_FAILURE** option. Since Linux 5.14, the limit is further decreased by 4 if the kernel is built with the **CONFIG_DEVICE_PRIVATE** option. Since Linux 5.19, the limit is further decreased by 1 if the kernel is built with the **CONFIG_PTE_MARKER** option.

Discard of swap pages was introduced in Linux 2.6.29, then made conditional on the **SWAP_FLAG_DISCARD** flag in Linux 2.6.36, which still discards the entire swap area when **swapon**() is called, even if that flag bit is not set.

SEE ALSO
mkswap(8), *swapoff* (8), *swapon*(8)

NAME

symlink, symlinkat − make a new name for a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int symlinkat(const char *target, int newdirfd, const char *linkpath);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

symlink():
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200112L
|| /* glibc <= 2.19: */ _BSD_SOURCE

symlinkat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

symlink() creates a symbolic link named *linkpath* which contains the string *target*.

Symbolic links are interpreted at run time as if the contents of the link had been substituted into the path being followed to find a file or directory.

Symbolic links may contain .. path components, which (if used at the start of the link) refer to the parent directories of that in which the link resides.

A symbolic link (also known as a soft link) may point to an existing file or to a nonexistent one; the latter case is known as a dangling link.

The permissions of a symbolic link are irrelevant; the ownership is ignored when following the link (except when the *protected_symlinks* feature is enabled, as explained in *proc*(5)), but is checked when removal or renaming of the link is requested and the link is in a directory with the sticky bit (**S_ISVTX**) set.

If *linkpath* exists, it will *not* be overwritten.

symlinkat()

The **symlinkat**() system call operates in exactly the same way as **symlink**(), except for the differences described here.

If the pathname given in *linkpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *newdirfd* (rather than relative to the current working directory of the calling process, as is done by **symlink**() for a relative pathname).

If *linkpath* is relative and *newdirfd* is the special value **AT_FDCWD**, then *linkpath* is interpreted relative to the current working directory of the calling process (like *symlink*())

If *linkpath* is absolute, then *newdirfd* is ignored.

See *openat*(2) for an explanation of the need for **symlinkat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Write access to the directory containing *linkpath* is denied, or one of the directories in the path prefix of *linkpath* did not allow search permission. (See also *path_resolution*(7).)

EBADF
(**symlinkat**()) *linkpath* is relative but *newdirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EDQUOT
The user's quota of resources on the filesystem has been exhausted. The resources could be inodes or disk blocks, depending on the filesystem implementation.

EEXIST
linkpath already exists.

EFAULT
target or *linkpath* points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *linkpath*.

ENAMETOOLONG
target or *linkpath* was too long.

ENOENT
A directory component in *linkpath* does not exist or is a dangling symbolic link, or *target* or *linkpath* is an empty string.

ENOENT
(**symlinkat**()) *linkpath* is a relative pathname and *newdirfd* refers to a directory that has been deleted.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing the file has no room for the new directory entry.

ENOTDIR
A component used as a directory in *linkpath* is not, in fact, a directory.

ENOTDIR
(**symlinkat**()) *linkpath* is relative and *newdirfd* is a file descriptor referring to a file other than a directory.

EPERM
The filesystem containing *linkpath* does not support the creation of symbolic links.

EROFS
linkpath is on a read-only filesystem.

STANDARDS
POSIX.1-2008.

HISTORY
symlink()
SVr4, 4.3BSD, POSIX.1-2001.

symlinkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

glibc notes
On older kernels where **symlinkat**() is unavailable, the glibc wrapper function falls back to the use of **symlink**(). When *linkpath* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *newdirfd* argument.

NOTES
No checking of *target* is done.

Deleting the name referred to by a symbolic link will actually delete the file (unless it also has other hard links). If this behavior is not desired, use *link*(2).

SEE ALSO

ln(1), *namei*(1), *lchown*(2), *link*(2), *lstat*(2), *open*(2), *readlink*(2), *rename*(2), *unlink*(2), *path_resolution*(7), *symlink*(7)

NAME

sync, syncfs − commit filesystem caches to disk

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

void sync(void);

int syncfs(int fd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sync():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE

syncfs():
_GNU_SOURCE

DESCRIPTION

sync() causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems.

syncfs() is like **sync**(), but synchronizes just the filesystem containing file referred to by the open file descriptor *fd*.

RETURN VALUE

syncfs() returns 0 on success; on error, it returns −1 and sets *errno* to indicate the error.

ERRORS

sync() is always successful.

syncfs() can fail for at least the following reasons:

EBADF
fd is not a valid file descriptor.

EIO An error occurred during synchronization. This error may relate to data written to any file on the filesystem, or on metadata related to the filesystem itself.

ENOSPC
Disk space was exhausted while synchronizing.

ENOSPC
EDQUOT
Data was written to a file on NFS or another filesystem which does not allocate space at the time of a *write*(2) system call, and some previous write failed due to insufficient storage space.

VERSIONS

According to the standard specification (e.g., POSIX.1-2001), **sync**() schedules the writes, but may return before the actual writing is done. However Linux waits for I/O completions, and thus **sync**() or **syncfs**() provide the same guarantees as **fsync**() called on every file in the system or filesystem respectively.

STANDARDS

sync() POSIX.1-2008.

syncfs()
Linux.

HISTORY

sync() POSIX.1-2001, SVr4, 4.3BSD.

syncfs()
Linux 2.6.39, glibc 2.14.

Since glibc 2.2.2, the Linux prototype for **sync**() is as listed above, following the various standards. In glibc 2.2.1 and earlier, it was "int sync(void)", and **sync**() always returned 0.

In mainline kernel versions prior to Linux 5.8, **syncfs**() will fail only when passed a bad file descriptor (**EBADF**). Since Linux 5.8, **syncfs**() will also report an error if one or more inodes failed to be written back since the last **syncfs**() call.

BUGS

Before Linux 1.3.20, Linux did not wait for I/O to complete before returning.

SEE ALSO

sync(1), *fdatasync*(2), *fsync*(2)

NAME

sync_file_range − sync a file segment with disk

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>

int sync_file_range(int fd, off_t offset, off_t nbytes,
                    unsigned int flags);
```

DESCRIPTION

sync_file_range() permits fine control when synchronizing the open file referred to by the file descriptor *fd* with disk.

offset is the starting byte of the file range to be synchronized. *nbytes* specifies the length of the range to be synchronized, in bytes; if *nbytes* is zero, then all bytes from *offset* through to the end of file are synchronized. Synchronization is in units of the system page size: *offset* is rounded down to a page boundary; *(offset+nbytes−1)* is rounded up to a page boundary.

The *flags* bit-mask argument can include any of the following values:

SYNC_FILE_RANGE_WAIT_BEFORE
: Wait upon write-out of all pages in the specified range that have already been submitted to the device driver for write-out before performing any write.

SYNC_FILE_RANGE_WRITE
: Initiate write-out of all dirty pages in the specified range which are not presently submitted write-out. Note that even this may block if you attempt to write more than request queue size.

SYNC_FILE_RANGE_WAIT_AFTER
: Wait upon write-out of all pages in the range after performing any write.

Specifying *flags* as 0 is permitted, as a no-op.

Warning

This system call is extremely dangerous and should not be used in portable programs. None of these operations writes out the file's metadata. Therefore, unless the application is strictly performing overwrites of already-instantiated disk blocks, there are no guarantees that the data will be available after a crash. There is no user interface to know if a write is purely an overwrite. On filesystems using copy-on-write semantics (e.g., *btrfs*) an overwrite of existing allocated blocks is impossible. When writing into preallocated space, many filesystems also require calls into the block allocator, which this system call does not sync out to disk. This system call does not flush disk write caches and thus does not provide any data integrity on systems with volatile disk write caches.

Some details

SYNC_FILE_RANGE_WAIT_BEFORE and **SYNC_FILE_RANGE_WAIT_AFTER** will detect any I/O errors or **ENOSPC** conditions and will return these to the caller.

Useful combinations of the *flags* bits are:

SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE
: Ensures that all pages in the specified range which were dirty when **sync_file_range**() was called are placed under write-out. This is a start-write-for-data-integrity operation.

SYNC_FILE_RANGE_WRITE
: Start write-out of all dirty pages in the specified range which are not presently under write-out. This is an asynchronous flush-to-disk operation. This is not suitable for data integrity operations.

SYNC_FILE_RANGE_WAIT_BEFORE (or **SYNC_FILE_RANGE_WAIT_AFTER**)
: Wait for completion of write-out of all pages in the specified range. This can be used after an earlier **SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE** operation to wait for completion of that operation, and obtain its result.

SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE | SYNC_FILE_RANGE_WAIT_AFTER
This is a write-for-data-integrity operation that will ensure that all pages in the specified range which were dirty when **sync_file_range**() was called are committed to disk.

RETURN VALUE

On success, **sync_file_range**() returns 0; on failure −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not a valid file descriptor.

EINVAL
flags specifies an invalid bit; or *offset* or *nbytes* is invalid.

EIO I/O error.

ENOMEM
Out of memory.

ENOSPC
Out of disk space.

ESPIPE
fd refers to something other than a regular file, a block device, or a directory.

VERSIONS

sync_file_range2()
Some architectures (e.g., PowerPC, ARM) need 64-bit arguments to be aligned in a suitable pair of registers. On such architectures, the call signature of **sync_file_range**() shown in the SYNOPSIS would force a register to be wasted as padding between the *fd* and *offset* arguments. (See *syscall*(2) for details.) Therefore, these architectures define a different system call that orders the arguments suitably:

```
int sync_file_range2(int fd, unsigned int flags,
                     off_t offset, off_t nbytes);
```

The behavior of this system call is otherwise exactly the same as **sync_file_range**().

STANDARDS

Linux.

HISTORY

Linux 2.6.17.

sync_file_range2()
A system call with this signature first appeared on the ARM architecture in Linux 2.6.20, with the name **arm_sync_file_range**(). It was renamed in Linux 2.6.22, when the analogous system call was added for PowerPC. On architectures where glibc support is provided, glibc transparently wraps **sync_file_range2**() under the name **sync_file_range**().

NOTES

_FILE_OFFSET_BITS should be defined to be 64 in code that takes the address of **sync_file_range**, if the code is intended to be portable to traditional 32-bit x86 and ARM platforms where **off_t**'s width defaults to 32 bits.

SEE ALSO

fdatasync(2), *fsync*(2), *msync*(2), *sync*(2)

NAME

syscall − indirect system call

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

long syscall(long *number***, ...);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

syscall():
Since glibc 2.19:
_DEFAULT_SOURCE
Before glibc 2.19:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

syscall() is a small library function that invokes the system call whose assembly language interface has the specified *number* with the specified arguments. Employing **syscall**() is useful, for example, when invoking a system call that has no wrapper function in the C library.

syscall() saves CPU registers before making the system call, restores the registers upon return from the system call, and stores any error returned by the system call in *errno*(3).

Symbolic constants for system call numbers can be found in the header file <*sys/syscall.h*>.

RETURN VALUE

The return value is defined by the system call being invoked. In general, a 0 return value indicates success. A −1 return value indicates an error, and an error number is stored in *errno*.

ERRORS

ENOSYS
The requested system call number is not implemented.

Other errors are specific to the invoked system call.

NOTES

syscall() first appeared in 4BSD.

Architecture-specific requirements

Each architecture ABI has its own requirements on how system call arguments are passed to the kernel. For system calls that have a glibc wrapper (e.g., most system calls), glibc handles the details of copying arguments to the right registers in a manner suitable for the architecture. However, when using **syscall**() to make a system call, the caller might need to handle architecture-dependent details; this requirement is most commonly encountered on certain 32-bit architectures.

For example, on the ARM architecture Embedded ABI (EABI), a 64-bit value (e.g., *long long*) must be aligned to an even register pair. Thus, using **syscall**() instead of the wrapper provided by glibc, the *readahead*(2) system call would be invoked as follows on the ARM architecture with the EABI in little endian mode:

```
syscall(SYS_readahead, fd, 0,
        (unsigned int) (offset & 0xFFFFFFFF),
        (unsigned int) (offset >> 32),
        count);
```

Since the offset argument is 64 bits, and the first argument (*fd*) is passed in *r0*, the caller must manually split and align the 64-bit value so that it is passed in the *r2*/*r3* register pair. That means inserting a dummy value into *r1* (the second argument of 0). Care also must be taken so that the split follows endian conventions (according to the C ABI for the platform).

Similar issues can occur on MIPS with the O32 ABI, on PowerPC and parisc with the 32-bit ABI, and on Xtensa.

Note that while the parisc C ABI also uses aligned register pairs, it uses a shim layer to hide the issue

from user space.

The affected system calls are *fadvise64_64*(2), *ftruncate64*(2), *posix_fadvise*(2), *pread64*(2), *pwrite64*(2), *readahead*(2), *sync_file_range*(2), and *truncate64*(2).

This does not affect syscalls that manually split and assemble 64-bit values such as *_llseek*(2), *preadv*(2), *preadv2*(2), *pwritev*(2), and *pwritev2*(2). Welcome to the wonderful world of historical baggage.

Architecture calling conventions

Every architecture has its own way of invoking and passing arguments to the kernel. The details for various architectures are listed in the two tables below.

The first table lists the instruction used to transition to kernel mode (which might not be the fastest or best way to transition to the kernel, so you might have to refer to *vdso*(7)), the register used to indicate the system call number, the register(s) used to return the system call result, and the register used to signal an error.

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	int $0x80	eax	eax	edx	-	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
loongarch	syscall 0	a7	a0	-	-	
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-	
powerpc	sc	r0	r3	-	r0	1
powerpc64	sc	r0	r3	-	cr0.SO	1
riscv	ecall	a7	a0	a1	-	
s390	svc 0	r1	r2	r3	-	3
s390x	svc 0	r1	r2	r3	-	3
superh	trapa #31	r3	r0	r1	-	4, 6
sparc/32	t 0x10	g1	o0	o1	psr/csr	1, 6
sparc/64	t 0x6d	g1	o0	o1	psr/csr	1, 6
tile	swint1	R10	R00	-	R01	1
x86-64	syscall	rax	rax	rdx	-	5
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

Notes:

- On a few architectures, a register is used as a boolean (0 indicating no error, and −1 indicating an error) to signal that the system call failed. The actual error value is still contained in the return register. On sparc, the carry bit (*csr*) in the processor status register (*psr*) is used instead of a full register. On powerpc64, the summary overflow bit (*SO*) in field 0 of the condition register (*cr0*) is used.
- *NR* is the system call number.
- For s390 and s390x, *NR* (the system call number) may be passed directly with *svc NR* if it is less than 256.
- On SuperH additional trap numbers are supported for historic reasons, but **trapa**#31 is the recommended "unified" ABI.

- The x32 ABI shares syscall table with x86-64 ABI, but there are some nuances:
  - In order to indicate that a system call is called under the x32 ABI, an additional bit, **__X32_SYSCALL_BIT**, is bitwise ORed with the system call number. The ABI used by a process affects some process behaviors, including signal handling or system call restarting.
  - Since x32 has different sizes for *long* and pointer types, layouts of some (but not all; *struct timeval* or *struct rlimit* are 64-bit, for example) structures are different. In order to handle this, additional system calls are added to the system call table, starting from number 512 (without the **__X32_SYSCALL_BIT**). For example, **__NR_readv** is defined as 19 for the x86-64 ABI and as *__X32_SYSCALL_BIT* | **515** for the x32 ABI. Most of these additional system calls are actually identical to the system calls used for providing i386 compat. There are some notable exceptions, however, such as *preadv2*(2), which uses *struct iovec* entities with 4-byte pointers and sizes ("compat_iovec" in kernel terms), but passes an 8-byte *pos* argument in a single register and not two, as is done in every other ABI.
- Some architectures (namely, Alpha, IA-64, MIPS, SuperH, sparc/32, and sparc/64) use an additional register ("Retval2" in the above table) to pass back a second return value from the *pipe*(2) system call; Alpha uses this technique in the architecture-specific *getxpid*(2), *getxuid*(2), and *getxgid*(2) system calls as well. Other architectures do not use the second return value register in the system call interface, even if it is defined in the System V ABI.

The second table shows the registers used to pass the system call arguments.

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	r0	r1	r2	r3	r4	r5	r6	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
loongarch	a0	a1	a2	a3	a4	a5	a6	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	1
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
powerpc64	r3	r4	r5	r6	r7	r8	-	
riscv	a0	a1	a2	a3	a4	a5	-	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
superh	r4	r5	r6	r7	r0	r1	r2	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
tile	R00	R01	R02	R03	R04	R05	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	
xtensa	a6	a3	a4	a5	a8	a9	-	

Notes:

- The mips/o32 system call convention passes arguments 5 through 8 on the user stack.

Note that these tables don’t cover the entire calling convention—some architectures may indiscriminately clobber other registers not listed here.

EXAMPLES

```
#define _GNU_SOURCE
#include <signal.h>
#include <sys/syscall.h>
#include <unistd.h>

int
main(void)
{
    pid_t tid;

    tid = syscall(SYS_gettid);
    syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

SEE ALSO

_syscall(2), *intro*(2), *syscalls*(2), *errno*(3), *vdso*(7)

NAME

_syscall − invoking a system call without library support (OBSOLETE)

SYNOPSIS

#include <linux/unistd.h>

A _syscall macro

desired system call

DESCRIPTION

The important thing to know about a system call is its prototype. You need to know how many arguments, their types, and the function return type. There are seven macros that make the actual call into the system easier. They have the form:

```
_syscallX(type,name,type1,arg1,type2,arg2,...)
```

where

X is 0–6, which are the number of arguments taken by the system call

type is the return type of the system call

name is the name of the system call

typeN is the Nth argument's type

argN is the name of the Nth argument

These macros create a function called *name* with the arguments you specify. Once you include the _syscall() in your source file, you call the system call by *name*.

FILES

/usr/include/linux/unistd.h

STANDARDS

Linux.

HISTORY

Starting around Linux 2.6.18, the _syscall macros were removed from header files supplied to user space. Use *syscall*(2) instead. (Some architectures, notably ia64, never provided the _syscall macros; on those architectures, *syscall*(2) was always required.)

NOTES

The _syscall() macros *do not* produce a prototype. You may have to create one, especially for C++ users.

System calls are not required to return only positive or negative error codes. You need to read the source to be sure how it will return errors. Usually, it is the negative of a standard error code, for example, −*EPERM*. The _syscall() macros will return the result *r* of the system call when *r* is nonnegative, but will return −1 and set the variable *errno* to −*r* when *r* is negative. For the error codes, see *errno*(3).

When defining a system call, the argument types *must* be passed by-value or by-pointer (for aggregates like structs).

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/unistd.h>       /* for _syscallX macros/related stuff */
#include <linux/kernel.h>       /* for struct sysinfo */

_syscall1(int, sysinfo, struct sysinfo *, info);

int
main(void)
{
    struct sysinfo s_info;
    int error;
```

```
        error = sysinfo(&s_info);
        printf("code error = %d\n", error);
        printf("Uptime = %lds\nLoad: 1 min %lu / 5 min %lu / 15 min %lu\n"
               "RAM: total %lu / free %lu / shared %lu\n"
               "Memory in buffers = %lu\nSwap: total %lu / free %lu\n"
               "Number of processes = %d\n",
               s_info.uptime, s_info.loads[0],
               s_info.loads[1], s_info.loads[2],
               s_info.totalram, s_info.freeram,
               s_info.sharedram, s_info.bufferram,
               s_info.totalswap, s_info.freeswap,
               s_info.procs);
        exit(EXIT_SUCCESS);
    }
```

Sample output

```
    code error = 0
    uptime = 502034s
    Load: 1 min 13376 / 5 min 5504 / 15 min 1152
    RAM: total 15343616 / free 827392 / shared 8237056
    Memory in buffers = 5066752
    Swap: total 27881472 / free 24698880
    Number of processes = 40
```

SEE ALSO

intro(2), *syscall*(2), *errno*(3)

NAME

syscalls − Linux system calls

SYNOPSIS

Linux system calls.

DESCRIPTION

The system call is the fundamental interface between an application and the Linux kernel.

System calls and library wrapper functions

System calls are generally not invoked directly, but rather via wrapper functions in glibc (or perhaps some other library). For details of direct invocation of a system call, see *intro*(2). Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes. For example, glibc contains a function **chdir**() which invokes the underlying "chdir" system call.

Often the glibc wrapper function is quite thin, doing little work other than copying arguments to the right registers before invoking the system call, and then setting *errno* appropriately after the system call has returned. (These are the same steps that are performed by *syscall*(2), which can be used to invoke system calls for which no wrapper function is provided.) Note: system calls indicate a failure by returning a negative error number to the caller on architectures without a separate error register/flag, as noted in *syscall*(2); when this happens, the wrapper function negates the returned error number (to make it positive), copies it to *errno*, and returns −1 to the caller of the wrapper.

Sometimes, however, the wrapper function does some extra work before invoking the system call. For example, nowadays there are (for reasons described below) two related system calls, *truncate*(2) and *truncate64*(2), and the glibc **truncate**() wrapper function checks which of those system calls are provided by the kernel and determines which should be employed.

System call list

Below is a list of the Linux system calls. In the list, the *Kernel* column indicates the kernel version for those system calls that were new in Linux 2.2, or have appeared since that kernel version. Note the following points:

- Where no kernel version is indicated, the system call appeared in Linux 1.0 or earlier.
- Where a system call is marked "1.2" this means the system call probably appeared in a Linux 1.1.x kernel version, and first appeared in a stable kernel with 1.2. (Development of the Linux 1.2 kernel was initiated from a branch of Linux 1.0.6 via the Linux 1.1.x unstable kernel series.)
- Where a system call is marked "2.0" this means the system call probably appeared in a Linux 1.3.x kernel version, and first appeared in a stable kernel with Linux 2.0. (Development of the Linux 2.0 kernel was initiated from a branch of Linux 1.2.x, somewhere around Linux 1.2.10, via the Linux 1.3.x unstable kernel series.)
- Where a system call is marked "2.2" this means the system call probably appeared in a Linux 2.1.x kernel version, and first appeared in a stable kernel with Linux 2.2.0. (Development of the Linux 2.2 kernel was initiated from a branch of Linux 2.0.21 via the Linux 2.1.x unstable kernel series.)
- Where a system call is marked "2.4" this means the system call probably appeared in a Linux 2.3.x kernel version, and first appeared in a stable kernel with Linux 2.4.0. (Development of the Linux 2.4 kernel was initiated from a branch of Linux 2.2.8 via the Linux 2.3.x unstable kernel series.)
- Where a system call is marked "2.6" this means the system call probably appeared in a Linux 2.5.x kernel version, and first appeared in a stable kernel with Linux 2.6.0. (Development of Linux 2.6 was initiated from a branch of Linux 2.4.15 via the Linux 2.5.x unstable kernel series.)
- Starting with Linux 2.6.0, the development model changed, and new system calls may appear in each Linux 2.6.x release. In this case, the exact version number where the system call appeared is shown. This convention continues with the Linux 3.x kernel series, which followed on from Linux 2.6.39; and the Linux 4.x kernel series, which followed on from Linux 3.19; and the Linux 5.x kernel series, which followed on from Linux 4.20; and the Linux 6.x kernel series, which followed on from Linux 5.19.
- In some cases, a system call was added to a stable kernel series after it branched from the previous stable kernel series, and then backported into the earlier stable kernel series. For example some system calls that appeared in Linux 2.6.x were also backported into a Linux 2.4.x release after Linux 2.4.15. When this is so, the version where the system call appeared in both of the major

kernel series is listed.

The list of system calls that are available as at Linux 5.14 (or in a few cases only on older kernels) is as follows:

System call	Kernel	Notes
_llseek(2)	1.2	
_newselect(2)	2.0	
_sysctl(2)	2.0	Removed in 5.5
accept(2)	2.0	See notes on *socketcall*(2)
accept4(2)	2.6.28	
access(2)	1.0	
acct(2)	1.0	
add_key(2)	2.6.10	
adjtimex(2)	1.0	
alarm(2)	1.0	
alloc_hugepages(2)	2.5.36	Removed in 2.5.44
arc_gettls(2)	3.9	ARC only
arc_settls(2)	3.9	ARC only
arc_usr_cmpxchg(2)	4.9	ARC only
arch_prctl(2)	2.6	x86_64, x86 since 4.12
atomic_barrier(2)	2.6.34	m68k only
atomic_cmpxchg_32(2)	2.6.34	m68k only
bdflush(2)	1.2	Deprecated (does nothing) since 2.6
bind(2)	2.0	See notes on *socketcall*(2)
bpf(2)	3.18	
brk(2)	1.0	
breakpoint(2)	2.2	ARM OABI only, defined with **__ARM_NR** prefix
cacheflush(2)	1.2	Not on x86
capget(2)	2.2	
capset(2)	2.2	
chdir(2)	1.0	
chmod(2)	1.0	
chown(2)	2.2	See *chown*(2) for version details
chown32(2)	2.4	
chroot(2)	1.0	
clock_adjtime(2)	2.6.39	
clock_getres(2)	2.6	
clock_gettime(2)	2.6	
clock_nanosleep(2)	2.6	
clock_settime(2)	2.6	
clone2(2)	2.4	IA-64 only
clone(2)	1.0	
clone3(2)	5.3	
close(2)	1.0	
close_range(2)	5.9	
connect(2)	2.0	See notes on *socketcall*(2)
copy_file_range(2)	4.5	
creat(2)	1.0	
create_module(2)	1.0	Removed in 2.6
delete_module(2)	1.0	
dup(2)	1.0	
dup2(2)	1.0	
dup3(2)	2.6.27	
epoll_create(2)	2.6	
epoll_create1(2)	2.6.27	

epoll_ctl(2)	2.6	
epoll_pwait(2)	2.6.19	
epoll_pwait2(2)	5.11	
epoll_wait(2)	2.6	
eventfd(2)	2.6.22	
eventfd2(2)	2.6.27	
execv(2)	2.0	SPARC/SPARC64 only, for compatibility with SunOS
execve(2)	1.0	
execveat(2)	3.19	
exit(2)	1.0	
exit_group(2)	2.6	
faccessat(2)	2.6.16	
faccessat2(2)	5.8	
fadvise64(2)	2.6	
fadvise64_64(2)	2.6	
fallocate(2)	2.6.23	
fanotify_init(2)	2.6.37	
fanotify_mark(2)	2.6.37	
fchdir(2)	1.0	
fchmod(2)	1.0	
fchmodat(2)	2.6.16	
fchown(2)	1.0	
fchown32(2)	2.4	
fchownat(2)	2.6.16	
fcntl(2)	1.0	
fcntl64(2)	2.4	
fdatasync(2)	2.0	
fgetxattr(2)	2.6; 2.4.18	
finit_module(2)	3.8	
flistxattr(2)	2.6; 2.4.18	
flock(2)	2.0	
fork(2)	1.0	
free_hugepages(2)	2.5.36	Removed in 2.5.44
fremovexattr(2)	2.6; 2.4.18	
fsconfig(2)	5.2	
fsetxattr(2)	2.6; 2.4.18	
fsmount(2)	5.2	
fsopen(2)	5.2	
fspick(2)	5.2	
fstat(2)	1.0	
fstat64(2)	2.4	
fstatat64(2)	2.6.16	
fstatfs(2)	1.0	
fstatfs64(2)	2.6	
fsync(2)	1.0	
ftruncate(2)	1.0	
ftruncate64(2)	2.4	
futex(2)	2.6	
futimesat(2)	2.6.16	
get_kernel_syms(2)	1.0	Removed in 2.6
get_mempolicy(2)	2.6.6	
get_robust_list(2)	2.6.17	
get_thread_area(2)	2.6	
get_tls(2)	4.15	ARM OABI only, has **__ARM_NR** prefix
getcpu(2)	2.6.19	
getcwd(2)	2.2	

getdents(2)	2.0	
getdents64(2)	2.4	
getdomainname(2)	2.2	SPARC, SPARC64; available as*osf_getdomainname*(2) on Alpha since Linux 2.0
getdtablesize(2)	2.0	SPARC (removed in 2.6.26), available on Alpha as *osf_getdtablesize*(2)
getegid(2)	1.0	
getegid32(2)	2.4	
geteuid(2)	1.0	
geteuid32(2)	2.4	
getgid(2)	1.0	
getgid32(2)	2.4	
getgroups(2)	1.0	
getgroups32(2)	2.4	
gethostname(2)	2.0	Alpha, was available on SPARC up to Linux 2.6.26
getitimer(2)	1.0	
getpeername(2)	2.0	See notes on *socketcall*(2)
getpagesize(2)	2.0	Not on x86
getpgid(2)	1.0	
getpgrp(2)	1.0	
getpid(2)	1.0	
getppid(2)	1.0	
getpriority(2)	1.0	
getrandom(2)	3.17	
getresgid(2)	2.2	
getresgid32(2)	2.4	
getresuid(2)	2.2	
getresuid32(2)	2.4	
getrlimit(2)	1.0	
getrusage(2)	1.0	
getsid(2)	2.0	
getsockname(2)	2.0	See notes on *socketcall*(2)
getsockopt(2)	2.0	See notes on *socketcall*(2)
gettid(2)	2.4.11	
gettimeofday(2)	1.0	
getuid(2)	1.0	
getuid32(2)	2.4	
getunwind(2)	2.4.8	IA-64 only; deprecated
getxattr(2)	2.6; 2.4.18	
getxgid(2)	2.0	Alpha only; see NOTES
getxpid(2)	2.0	Alpha only; see NOTES
getxuid(2)	2.0	Alpha only; see NOTES
init_module(2)	1.0	
inotify_add_watch(2)	2.6.13	
inotify_init(2)	2.6.13	
inotify_init1(2)	2.6.27	
inotify_rm_watch(2)	2.6.13	
io_cancel(2)	2.6	
io_destroy(2)	2.6	
io_getevents(2)	2.6	
io_pgetevents(2)	4.18	
io_setup(2)	2.6	
io_submit(2)	2.6	
io_uring_enter(2)	5.1	

io_uring_register(2)	5.1	
io_uring_setup(2)	5.1	
ioctl(2)	1.0	
ioperm(2)	1.0	
iopl(2)	1.0	
ioprio_get(2)	2.6.13	
ioprio_set(2)	2.6.13	
ipc(2)	1.0	
kcmp(2)	3.5	
kern_features(2)	3.7	SPARC64 only
kexec_file_load(2)	3.17	
kexec_load(2)	2.6.13	
keyctl(2)	2.6.10	
kill(2)	1.0	
landlock_add_rule(2)	5.13	
landlock_create_ruleset(2)	5.13	
landlock_restrict_self(2)	5.13	
lchown(2)	1.0	See *chown*(2) for version details
lchown32(2)	2.4	
lgetxattr(2)	2.6; 2.4.18	
link(2)	1.0	
linkat(2)	2.6.16	
listen(2)	2.0	See notes on *socketcall*(2)
listxattr(2)	2.6; 2.4.18	
llistxattr(2)	2.6; 2.4.18	
lookup_dcookie(2)	2.6	
lremovexattr(2)	2.6; 2.4.18	
lseek(2)	1.0	
lsetxattr(2)	2.6; 2.4.18	
lstat(2)	1.0	
lstat64(2)	2.4	
madvise(2)	2.4	
mbind(2)	2.6.6	
memory_ordering(2)	2.2	SPARC64 only
membarrier(2)	3.17	
memfd_create(2)	3.17	
memfd_secret(2)	5.14	
migrate_pages(2)	2.6.16	
mincore(2)	2.4	
mkdir(2)	1.0	
mkdirat(2)	2.6.16	
mknod(2)	1.0	
mknodat(2)	2.6.16	
mlock(2)	2.0	
mlock2(2)	4.4	
mlockall(2)	2.0	
mmap(2)	1.0	
mmap2(2)	2.4	
modify_ldt(2)	1.0	
mount(2)	1.0	
move_mount(2)	5.2	
move_pages(2)	2.6.18	
mprotect(2)	1.0	
mq_getsetattr(2)	2.6.6	
mq_notify(2)	2.6.6	
mq_open(2)	2.6.6	
mq_timedreceive(2)	2.6.6	

mq_timedsend(2)	2.6.6	
mq_unlink(2)	2.6.6	
mremap(2)	2.0	
msgctl(2)	2.0	See notes on *ipc*(2)
msgget(2)	2.0	See notes on *ipc*(2)
msgrcv(2)	2.0	See notes on *ipc*(2)
msgsnd(2)	2.0	See notes on *ipc*(2)
msync(2)	2.0	
munlock(2)	2.0	
munlockall(2)	2.0	
munmap(2)	1.0	
name_to_handle_at(2)	2.6.39	
nanosleep(2)	2.0	
newfstatat(2)	2.6.16	See *stat*(2)
nfsservctl(2)	2.2	Removed in 3.1
nice(2)	1.0	
old_adjtimex(2)	2.0	Alpha only; see NOTES
old_getrlimit(2)	2.4	Old variant of *getrlimit*(2) that used a different value for **RLIM_INFINITY**
oldfstat(2)	1.0	
oldlstat(2)	1.0	
oldolduname(2)	1.0	
oldstat(2)	1.0	
oldumount(2)	2.4.116	Name of the old *umount*(2) syscall on Alpha
olduname(2)	1.0	
open(2)	1.0	
open_by_handle_at(2)	2.6.39	
open_tree(2)	5.2	
openat(2)	2.6.16	
openat2(2)	5.6	
or1k_atomic(2)	3.1	OpenRISC 1000 only
pause(2)	1.0	
pciconfig_iobase(2)	2.2.15; 2.4	Not on x86
pciconfig_read(2)	2.0.26; 2.2	Not on x86
pciconfig_write(2)	2.0.26; 2.2	Not on x86
perf_event_open(2)	2.6.31	Was perf_counter_open() in 2.6.31; renamed in 2.6.32
personality(2)	1.2	
perfctr(2)	2.2	SPARC only; removed in 2.6.34
perfmonctl(2)	2.4	IA-64 only; removed in 5.10
pidfd_getfd(2)	5.6	
pidfd_send_signal(2)	5.1	
pidfd_open(2)	5.3	
pipe(2)	1.0	
pipe2(2)	2.6.27	
pivot_root(2)	2.4	
pkey_alloc(2)	4.8	
pkey_free(2)	4.8	
pkey_mprotect(2)	4.8	
poll(2)	2.0.36; 2.2	
ppoll(2)	2.6.16	
prctl(2)	2.2	
pread64(2)		Added as "pread" in 2.2; renamed "pread64" in 2.6
preadv(2)	2.6.30	

preadv2(2)	4.6	
prlimit64(2)	2.6.36	
process_madvise(2)	5.10	
process_vm_readv(2)	3.2	
process_vm_writev(2)	3.2	
pselect6(2)	2.6.16	
ptrace(2)	1.0	
pwrite64(2)		Added as "pwrite" in 2.2; renamed "pwrite64" in 2.6
pwritev(2)	2.6.30	
pwritev2(2)	4.6	
query_module(2)	2.2	Removed in 2.6
quotactl(2)	1.0	
quotactl_fd(2)	5.14	
read(2)	1.0	
readahead(2)	2.4.13	
readdir(2)	1.0	
readlink(2)	1.0	
readlinkat(2)	2.6.16	
readv(2)	2.0	
reboot(2)	1.0	
recv(2)	2.0	See notes on *socketcall*(2)
recvfrom(2)	2.0	See notes on *socketcall*(2)
recvmsg(2)	2.0	See notes on *socketcall*(2)
recvmmsg(2)	2.6.33	
remap_file_pages(2)	2.6	Deprecated since 3.16
removexattr(2)	2.6; 2.4.18	
rename(2)	1.0	
renameat(2)	2.6.16	
renameat2(2)	3.15	
request_key(2)	2.6.10	
restart_syscall(2)	2.6	
riscv_flush_icache(2)	4.15	RISC-V only
rmdir(2)	1.0	
rseq(2)	4.18	
rt_sigaction(2)	2.2	
rt_sigpending(2)	2.2	
rt_sigprocmask(2)	2.2	
rt_sigqueueinfo(2)	2.2	
rt_sigreturn(2)	2.2	
rt_sigsuspend(2)	2.2	
rt_sigtimedwait(2)	2.2	
rt_tgsigqueueinfo(2)	2.6.31	
rtas(2)	2.6.2	PowerPC/PowerPC64 only
s390_runtime_instr(2)	3.7	s390 only
s390_pci_mmio_read(2)	3.19	s390 only
s390_pci_mmio_write(2)	3.19	s390 only
s390_sthyi(2)	4.15	s390 only
s390_guarded_storage(2)	4.12	s390 only
sched_get_affinity(2)	2.6	Name of **sched_getaffinity**(2) on SPARC and SPARC64
sched_get_priority_max(2)	2.0	
sched_get_priority_min(2)	2.0	
sched_getaffinity(2)	2.6	
sched_getattr(2)	3.14	
sched_getparam(2)	2.0	
sched_getscheduler(2)	2.0	

sched_rr_get_interval(2)	2.0	
sched_set_affinity(2)	2.6	Name of **sched_setaffinity**(2) on SPARC and SPARC64
sched_setaffinity(2)	2.6	
sched_setattr(2)	3.14	
sched_setparam(2)	2.0	
sched_setscheduler(2)	2.0	
sched_yield(2)	2.0	
seccomp(2)	3.17	
select(2)	1.0	
semctl(2)	2.0	See notes on *ipc*(2)
semget(2)	2.0	See notes on *ipc*(2)
semop(2)	2.0	See notes on *ipc*(2)
semtimedop(2)	2.6; 2.4.22	
send(2)	2.0	See notes on *socketcall*(2)
sendfile(2)	2.2	
sendfile64(2)	2.6; 2.4.19	
sendmmsg(2)	3.0	
sendmsg(2)	2.0	See notes on *socketcall*(2)
sendto(2)	2.0	See notes on *socketcall*(2)
set_mempolicy(2)	2.6.6	
set_robust_list(2)	2.6.17	
set_thread_area(2)	2.6	
set_tid_address(2)	2.6	
set_tls(2)	2.6.11	ARM OABI/EABI only (constant has **__ARM_NR** prefix)
setdomainname(2)	1.0	
setfsgid(2)	1.2	
setfsgid32(2)	2.4	
setfsuid(2)	1.2	
setfsuid32(2)	2.4	
setgid(2)	1.0	
setgid32(2)	2.4	
setgroups(2)	1.0	
setgroups32(2)	2.4	
sethae(2)	2.0	Alpha only; see NOTES
sethostname(2)	1.0	
setitimer(2)	1.0	
setns(2)	3.0	
setpgid(2)	1.0	
setpgrp(2)	2.0	Alternative name for *setpgid*(2) on Alpha
setpriority(2)	1.0	
setregid(2)	1.0	
setregid32(2)	2.4	
setresgid(2)	2.2	
setresgid32(2)	2.4	
setresuid(2)	2.2	
setresuid32(2)	2.4	
setreuid(2)	1.0	
setreuid32(2)	2.4	
setrlimit(2)	1.0	
setsid(2)	1.0	
setsockopt(2)	2.0	See notes on *socketcall*(2)
settimeofday(2)	1.0	

setuid(2)	1.0	
setuid32(2)	2.4	
setup(2)	1.0	Removed in 2.2
setxattr(2)	2.6; 2.4.18	
sgetmask(2)	1.0	
shmat(2)	2.0	See notes on *ipc*(2)
shmctl(2)	2.0	See notes on *ipc*(2)
shmdt(2)	2.0	See notes on *ipc*(2)
shmget(2)	2.0	See notes on *ipc*(2)
shutdown(2)	2.0	See notes on *socketcall*(2)
sigaction(2)	1.0	
sigaltstack(2)	2.2	
signal(2)	1.0	
signalfd(2)	2.6.22	
signalfd4(2)	2.6.27	
sigpending(2)	1.0	
sigprocmask(2)	1.0	
sigreturn(2)	1.0	
sigsuspend(2)	1.0	
socket(2)	2.0	See notes on *socketcall*(2)
socketcall(2)	1.0	
socketpair(2)	2.0	See notes on *socketcall*(2)
spill(2)	2.6.13	Xtensa only
splice(2)	2.6.17	
spu_create(2)	2.6.16	PowerPC/PowerPC64 only
spu_run(2)	2.6.16	PowerPC/PowerPC64 only
ssetmask(2)	1.0	
stat(2)	1.0	
stat64(2)	2.4	
statfs(2)	1.0	
statfs64(2)	2.6	
statx(2)	4.11	
stime(2)	1.0	
subpage_prot(2)	2.6.25	PowerPC/PowerPC64 only
swapcontext(2)	2.6.3	PowerPC/PowerPC64 only
switch_endian(2)	4.1	PowerPC64 only
swapoff(2)	1.0	
swapon(2)	1.0	
symlink(2)	1.0	
symlinkat(2)	2.6.16	
sync(2)	1.0	
sync_file_range(2)	2.6.17	
sync_file_range2(2)	2.6.22	
syncfs(2)	2.6.39	
sys_debug_setcontext(2)	2.6.11	PowerPC only
syscall(2)	1.0	Still available on ARM OABI and MIPS O32 ABI
sysfs(2)	1.2	
sysinfo(2)	1.0	
syslog(2)	1.0	
sysmips(2)	2.6.0	MIPS only
tee(2)	2.6.17	
tgkill(2)	2.6	
time(2)	1.0	
timer_create(2)	2.6	
timer_delete(2)	2.6	
timer_getoverrun(2)	2.6	
timer_gettime(2)	2.6	

timer_settime(2)	2.6	
timerfd_create(2)	2.6.25	
timerfd_gettime(2)	2.6.25	
timerfd_settime(2)	2.6.25	
times(2)	1.0	
tkill(2)	2.6; 2.4.22	
truncate(2)	1.0	
truncate64(2)	2.4	
ugetrlimit(2)	2.4	
umask(2)	1.0	
umount(2)	1.0	
umount2(2)	2.2	
uname(2)	1.0	
unlink(2)	1.0	
unlinkat(2)	2.6.16	
unshare(2)	2.6.16	
uselib(2)	1.0	
ustat(2)	1.0	
userfaultfd(2)	4.3	
usr26(2)	2.4.8.1	ARM OABI only
usr32(2)	2.4.8.1	ARM OABI only
utime(2)	1.0	
utimensat(2)	2.6.22	
utimes(2)	2.2	
utrap_install(2)	2.2	SPARC64 only
vfork(2)	2.2	
vhangup(2)	1.0	
vm86old(2)	1.0	Was "vm86"; renamed in 2.0.28/2.2
vm86(2)	2.0.28; 2.2	
vmsplice(2)	2.6.17	
wait4(2)	1.0	
waitid(2)	2.6.10	
waitpid(2)	1.0	
write(2)	1.0	
writev(2)	2.0	
xtensa(2)	2.6.13	Xtensa only

On many platforms, including x86-32, socket calls are all multiplexed (via glibc wrapper functions) through *socketcall*(2) and similarly System V IPC calls are multiplexed through *ipc*(2).

Although slots are reserved for them in the system call table, the following system calls are not implemented in the standard kernel: *afs_syscall*(2), *break*(2), *ftime*(2), *getpmsg*(2), *gtty*(2), *idle*(2), *lock*(2), *madvise1*(2), *mpx*(2), *phys*(2), *prof*(2), *profil*(2), *putpmsg*(2), *security*(2), *stty*(2), *tuxcall*(2), *ulimit*(2), and *vserver*(2) (see also *unimplemented*(2)). However, *ftime*(3), *profil*(3), and *ulimit*(3) exist as library routines. The slot for *phys*(2) is in use since Linux 2.1.116 for *umount*(2); *phys*(2) will never be implemented. The *getpmsg*(2) and *putpmsg*(2) calls are for kernels patched to support STREAMS, and may never be in the standard kernel.

There was briefly *set_zone_reclaim*(2), added in Linux 2.6.13, and removed in Linux 2.6.16; this system call was never available to user space.

System calls on removed ports

Some system calls only ever existed on Linux architectures that have since been removed from the kernel:

AVR32 (port removed in Linux 4.12)

- *pread*(2)
- *pwrite*(2)

Blackfin (port removed in Linux 4.17)

- *bfin_spinlock*(2) (added in Linux 2.6.22)
- *dma_memcpy*(2) (added in Linux 2.6.22)
- *pread*(2) (added in Linux 2.6.22)
- *pwrite*(2) (added in Linux 2.6.22)
- *sram_alloc*(2) (added in Linux 2.6.22)
- *sram_free*(2) (added in Linux 2.6.22)

Metag (port removed in Linux 4.17)

- *metag_get_tls*(2) (add in Linux 3.9)
- *metag_set_fpu_flags*(2) (add in Linux 3.9)
- *metag_set_tls*(2) (add in Linux 3.9)
- *metag_setglobalbit*(2) (add in Linux 3.9)

Tile (port removed in Linux 4.17)

- *cmpxchg_badaddr*(2) (added in Linux 2.6.36)

NOTES

Roughly speaking, the code belonging to the system call with number __NR_xxx defined in */usr/include/asm/unistd.h* can be found in the Linux kernel source in the routine *sys_xxx*(). There are many exceptions, however, mostly because older system calls were superseded by newer ones, and this has been treated somewhat unsystematically. On platforms with proprietary operating-system emulation, such as sparc, sparc64, and alpha, there are many additional system calls; mips64 also contains a full set of 32-bit system calls.

Over time, changes to the interfaces of some system calls have been necessary. One reason for such changes was the need to increase the size of structures or scalar values passed to the system call. Because of these changes, certain architectures (notably, longstanding 32-bit architectures such as i386) now have various groups of related system calls (e.g., *truncate*(2) and *truncate64*(2)) which perform similar tasks, but which vary in details such as the size of their arguments. (As noted earlier, applications are generally unaware of this: the glibc wrapper functions do some work to ensure that the right system call is invoked, and that ABI compatibility is preserved for old binaries.) Examples of system calls that exist in multiple versions are the following:

- By now there are three different versions of *stat*(2): *sys_stat*() (slot *__NR_oldstat*), *sys_newstat*() (slot *__NR_stat*), and *sys_stat64*() (slot *__NR_stat64*), with the last being the most current. A similar story applies for *lstat*(2) and *fstat*(2).
- Similarly, the defines *__NR_oldolduname*, *__NR_olduname*, and *__NR_uname* refer to the routines *sys_olduname*(), *sys_uname*(), and *sys_newuname*().
- In Linux 2.0, a new version of *vm86*(2) appeared, with the old and the new kernel routines being named *sys_vm86old*() and *sys_vm86*().
- In Linux 2.4, a new version of *getrlimit*(2) appeared, with the old and the new kernel routines being named *sys_old_getrlimit*() (slot *__NR_getrlimit*) and *sys_getrlimit*() (slot *__NR_ugetrlimit*).
- Linux 2.4 increased the size of user and group IDs from 16 to 32 bits. To support this change, a range of system calls were added (e.g., *chown32*(2), *getuid32*(2), *getgroups32*(2), *setresuid32*(2)), superseding earlier calls of the same name without the "32" suffix.
- Linux 2.4 added support for applications on 32-bit architectures to access large files (i.e., files for which the sizes and file offsets can't be represented in 32 bits.) To support this change, replacements were required for system calls that deal with file offsets and sizes. Thus the following system calls were added: *fcntl64*(2), *getdents64*(2), *stat64*(2), *statfs64*(2), *truncate64*(2), and their analogs that work with file descriptors or symbolic links. These system calls supersede the older system calls which, except in the case of the "stat" calls, have the same name without the "64" suffix.

  On newer platforms that only have 64-bit file access and 32-bit UIDs/GIDs (e.g., alpha, ia64, s390x, x86-64), there is just a single version of the UID/GID and file access system calls. On platforms (typically, 32-bit platforms) where the *64 and *32 calls exist, the other versions are obsolete.

- The *rt_sig** calls were added in Linux 2.2 to support the addition of real-time signals (see *signal*(7)). These system calls supersede the older system calls of the same name without the "rt_" prefix.
- The *select*(2) and *mmap*(2) system calls use five or more arguments, which caused problems in the way argument passing on the i386 used to be set up. Thus, while other architectures have *sys_select*() and *sys_mmap*() corresponding to *__NR_select* and *__NR_mmap*, on i386 one finds *old_select*() and *old_mmap*() (routines that use a pointer to an argument block) instead. These days passing five arguments is not a problem any more, and there is a *__NR__newselect* that corresponds directly to *sys_select*() and similarly *__NR_mmap2*. s390x is the only 64-bit architecture that has *old_mmap*().

Architecture-specific details: Alpha

getxgid(2)
: returns a pair of GID and effective GID via registers **r0** and **r20**; it is provided instead of **getgid**(2) and **getegid**(2).

getxpid(2)
: returns a pair of PID and parent PID via registers **r0** and **r20**; it is provided instead of **getpid**(2) and **getppid**(2).

old_adjtimex(2)
: is a variant of **adjtimex**(2) that uses *struct timeval32*, for compatibility with OSF/1.

getxuid(2)
: returns a pair of GID and effective GID via registers **r0** and **r20**; it is provided instead of **getuid**(2) and **geteuid**(2).

sethae(2)
: is used for configuring the Host Address Extension register on low-cost Alphas in order to access address space beyond first 27 bits.

SEE ALSO

ausyscall(1), *intro*(2), *syscall*(2), *unimplemented*(2), *errno*(3), *libc*(7), *vdso*(7)

NAME

sysctl − read/write system parameters

SYNOPSIS

#include <unistd.h>
#include <linux/sysctl.h>

[[deprecated]] int _sysctl(struct __sysctl_args **args*);

DESCRIPTION

This system call no longer exists on current kernels! See NOTES.

The **_sysctl**() call reads and/or writes kernel parameters. For example, the hostname, or the maximum number of open files. The argument has the form

```
struct __sysctl_args {
    int    *name;    /* integer vector describing variable */
    int     nlen;    /* length of this vector */
    void   *oldval;  /* 0 or address where to store old value */
    size_t *oldlenp; /* available room for old value,
                        overwritten by actual size of old value */
    void   *newval;  /* 0 or address of new value */
    size_t  newlen;  /* size of new value */
};
```

This call does a search in a tree structure, possibly resembling a directory tree under */proc/sys*, and if the requested item is found calls some appropriate routine to read or modify the value.

RETURN VALUE

Upon successful completion, **_sysctl**() returns 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
EPERM
No search permission for one of the encountered "directories", or no read permission where *oldval* was nonzero, or no write permission where *newval* was nonzero.

EFAULT
The invocation asked for the previous value by setting *oldval* non-NULL, but allowed zero room in *oldlenp*.

ENOTDIR
name was not found.

STANDARDS

Linux.

HISTORY

Linux 1.3.57. Removed in Linux 5.5, glibc 2.32.

It originated in 4.4BSD. Only Linux has the */proc/sys* mirror, and the object naming schemes differ between Linux and 4.4BSD, but the declaration of the **sysctl**() function is the same in both.

NOTES

Use of this system call was long discouraged: since Linux 2.6.24, uses of this system call result in warnings in the kernel log, and in Linux 5.5, the system call was finally removed. Use the */proc/sys* interface instead.

Note that on older kernels where this system call still exists, it is available only if the kernel was configured with the **CONFIG_SYSCTL_SYSCALL** option. Furthermore, glibc does not provide a wrapper for this system call, necessitating the use of *syscall*(2).

BUGS

The object names vary between kernel versions, making this system call worthless for applications.

Not all available objects are properly documented.

It is not yet possible to change operating system by writing to */proc/sys/kernel/ostype*.

EXAMPLES

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

#include <linux/sysctl.h>

#define ARRAY_SIZE(arr)  (sizeof(arr) / sizeof((arr)[0]))

int _sysctl(struct __sysctl_args *args);

#define OSNAMESZ 100

int
main(void)
{
    int                  name[] = { CTL_KERN, KERN_OSTYPE };
    char                 osname[OSNAMESZ];
    size_t               osnamelth;
    struct __sysctl_args  args;

    memset(&args, 0, sizeof(args));
    args.name = name;
    args.nlen = ARRAY_SIZE(name);
    args.oldval = osname;
    args.oldlenp = &osnamelth;

    osnamelth = sizeof(osname);

    if (syscall(SYS__sysctl, &args) == -1) {
        perror("_sysctl");
        exit(EXIT_FAILURE);
    }
    printf("This machine is running %*s\n", (int) osnamelth, osname);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

proc(5)

NAME

sysfs − get filesystem type information

SYNOPSIS

[[deprecated]] int sysfs(int *option***, const char *** *fsname***);**
[[deprecated]] int sysfs(int *option***, unsigned int** *fs_index***, char *** *buf* **);**
[[deprecated]] int sysfs(int *option***);**

DESCRIPTION

Note: if you are looking for information about the **sysfs** filesystem that is normally mounted at */sys*, see *sysfs*(5).

The (obsolete) **sysfs**() system call returns information about the filesystem types currently present in the kernel. The specific form of the **sysfs**() call and the information returned depends on the *option* in effect:

1 Translate the filesystem identifier string *fsname* into a filesystem type index.

2 Translate the filesystem type index *fs_index* into a null-terminated filesystem identifier string. This string will be written to the buffer pointed to by *buf* . Make sure that *buf* has enough space to accept the string.

3 Return the total number of filesystem types currently present in the kernel.

The numbering of the filesystem type indexes begins with zero.

RETURN VALUE

On success, **sysfs**() returns the filesystem index for option **1**, zero for option **2**, and the number of currently configured filesystems for option **3**. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
Either *fsname* or *buf* is outside your accessible address space.

EINVAL
fsname is not a valid filesystem type identifier; *fs_index* is out-of-bounds; *option* is invalid.

STANDARDS

None.

HISTORY

SVr4.

This System-V derived system call is obsolete; don't use it. On systems with */proc*, the same information can be obtained via */proc*; use that interface instead.

BUGS

There is no libc or glibc support. There is no way to guess how large *buf* should be.

SEE ALSO

proc(5), *sysfs*(5)

NAME

sysinfo − return system information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sysinfo.h>

int sysinfo(struct sysinfo *****info*);**

DESCRIPTION

sysinfo() returns certain statistics on memory and swap usage, as well as the load average.

Until Linux 2.3.16, **sysinfo**() returned information in the following structure:

```
struct sysinfo {
    long uptime;             /* Seconds since boot */
    unsigned long loads[3];  /* 1, 5, and 15 minute load averages */
    unsigned long totalram;  /* Total usable main memory size */
    unsigned long freeram;   /* Available memory size */
    unsigned long sharedram; /* Amount of shared memory */
    unsigned long bufferram; /* Memory used by buffers */
    unsigned long totalswap; /* Total swap space size */
    unsigned long freeswap;  /* Swap space still available */
    unsigned short procs;    /* Number of current processes */
    char _f[22];             /* Pads structure to 64 bytes */
};
```

In the above structure, the sizes of the memory and swap fields are given in bytes.

Since Linux 2.3.23 (i386) and Linux 2.3.48 (all architectures) the structure is:

```
struct sysinfo {
    long uptime;             /* Seconds since boot */
    unsigned long loads[3];  /* 1, 5, and 15 minute load averages */
    unsigned long totalram;  /* Total usable main memory size */
    unsigned long freeram;   /* Available memory size */
    unsigned long sharedram; /* Amount of shared memory */
    unsigned long bufferram; /* Memory used by buffers */
    unsigned long totalswap; /* Total swap space size */
    unsigned long freeswap;  /* Swap space still available */
    unsigned short procs;    /* Number of current processes */
    unsigned long totalhigh; /* Total high memory size */
    unsigned long freehigh;  /* Available high memory size */
    unsigned int mem_unit;   /* Memory unit size in bytes */
    char _f[20-2*sizeof(long)-sizeof(int)];
                             /* Padding to 64 bytes */
};
```

In the above structure, sizes of the memory and swap fields are given as multiples of *mem_unit* bytes.

RETURN VALUE

On success, **sysinfo**() returns zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT

info is not a valid address.

STANDARDS

Linux.

HISTORY

Linux 0.98.pl6.

NOTES

All of the information provided by this system call is also available via */proc/meminfo* and */proc/loadavg*.

SEE ALSO

proc(5)

NAME

syslog, klogctl − read and/or clear kernel message ring buffer; set console_loglevel

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/klog.h>      /* Definition of SYSLOG_* constants */
#include <sys/syscall.h>   /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_syslog, int type, char *bufp, int len);

/* The glibc interface */
#include <sys/klog.h>

int klogctl(int type, char *bufp, int len);
```

DESCRIPTION

Note: Probably, you are looking for the C library function **syslog**(), which talks to *syslogd*(8); see *syslog*(3) for details.

This page describes the kernel **syslog**() system call, which is used to control the kernel *printk*() buffer; the glibc wrapper function for the system call is called **klogctl**().

The kernel log buffer

The kernel has a cyclic buffer of length **LOG_BUF_LEN** in which messages given as arguments to the kernel function **printk**() are stored (regardless of their log level). In early kernels, **LOG_BUF_LEN** had the value 4096; from Linux 1.3.54, it was 8192; from Linux 2.1.113, it was 16384; since Linux 2.4.23/2.6, the value is a kernel configuration option (**CONFIG_LOG_BUF_SHIFT**, default value dependent on the architecture). Since Linux 2.6.6, the size can be queried with command type 10 (see below).

Commands

The *type* argument determines the action taken by this function. The list below specifies the values for *type*. The symbolic names are defined in the kernel source, but are not exported to user space; you will either need to use the numbers, or define the names yourself.

SYSLOG_ACTION_CLOSE (0)
: Close the log. Currently a NOP.

SYSLOG_ACTION_OPEN (1)
: Open the log. Currently a NOP.

SYSLOG_ACTION_READ (2)
: Read from the log. The call waits until the kernel log buffer is nonempty, and then reads at most *len* bytes into the buffer pointed to by *bufp*. The call returns the number of bytes read. Bytes read from the log disappear from the log buffer: the information can be read only once. This is the function executed by the kernel when a user program reads */proc/kmsg*.

SYSLOG_ACTION_READ_ALL (3)
: Read all messages remaining in the ring buffer, placing them in the buffer pointed to by *bufp*. The call reads the last *len* bytes from the log buffer (nondestructively), but will not read more than was written into the buffer since the last "clear ring buffer" command (see command 5 below)). The call returns the number of bytes read.

SYSLOG_ACTION_READ_CLEAR (4)
: Read and clear all messages remaining in the ring buffer. The call does precisely the same as for a *type* of 3, but also executes the "clear ring buffer" command.

SYSLOG_ACTION_CLEAR (5)
: The call executes just the "clear ring buffer" command. The *bufp* and *len* arguments are ignored.

  This command does not really clear the ring buffer. Rather, it sets a kernel bookkeeping variable that determines the results returned by commands 3 (**SYSLOG_ACTION_READ_ALL**) and 4 (**SYSLOG_ACTION_READ_CLEAR**). This command has no effect on commands 2 (**SYSLOG_ACTION_READ**) and 9 (**SYSLOG_ACTION_SIZE_UNREAD**).

SYSLOG_ACTION_CONSOLE_OFF (6)
The command saves the current value of *console_loglevel* and then sets *console_loglevel* to *minimum_console_loglevel*, so that no messages are printed to the console. Before Linux 2.6.32, the command simply sets *console_loglevel* to *minimum_console_loglevel*. See the discussion of */proc/sys/kernel/printk*, below.

The *bufp* and *len* arguments are ignored.

SYSLOG_ACTION_CONSOLE_ON (7)
If a previous **SYSLOG_ACTION_CONSOLE_OFF** command has been performed, this command restores *console_loglevel* to the value that was saved by that command. Before Linux 2.6.32, this command simply sets *console_loglevel* to *default_console_loglevel*. See the discussion of */proc/sys/kernel/printk*, below.

The *bufp* and *len* arguments are ignored.

SYSLOG_ACTION_CONSOLE_LEVEL (8)
The call sets *console_loglevel* to the value given in *len*, which must be an integer between 1 and 8 (inclusive). The kernel silently enforces a minimum value of *minimum_console_loglevel* for *len*. See the *log level* section for details. The *bufp* argument is ignored.

SYSLOG_ACTION_SIZE_UNREAD (9) (since Linux 2.4.10)
The call returns the number of bytes currently available to be read from the kernel log buffer via command 2 (**SYSLOG_ACTION_READ**). The *bufp* and *len* arguments are ignored.

SYSLOG_ACTION_SIZE_BUFFER (10) (since Linux 2.6.6)
This command returns the total size of the kernel log buffer. The *bufp* and *len* arguments are ignored.

All commands except 3 and 10 require privilege. In Linux kernels before Linux 2.6.37, command types 3 and 10 are allowed to unprivileged processes; since Linux 2.6.37, these commands are allowed to unprivileged processes only if */proc/sys/kernel/dmesg_restrict* has the value 0. Before Linux 2.6.37, "privileged" means that the caller has the **CAP_SYS_ADMIN** capability. Since Linux 2.6.37, "privileged" means that the caller has either the **CAP_SYS_ADMIN** capability (now deprecated for this purpose) or the (new) **CAP_SYSLOG** capability.

/proc/sys/kernel/printk
/proc/sys/kernel/printk is a writable file containing four integer values that influence kernel *printk()* behavior when printing or logging error messages. The four values are:

console_loglevel
Only messages with a log level lower than this value will be printed to the console. The default value for this field is **DEFAULT_CONSOLE_LOGLEVEL** (7), but it is set to 4 if the kernel command line contains the word "quiet", 10 if the kernel command line contains the word "debug", and to 15 in case of a kernel fault (the 10 and 15 are just silly, and equivalent to 8). The value of *console_loglevel* can be set (to a value in the range 1–8) by a **syslog**() call with a *type* of 8.

default_message_loglevel
This value will be used as the log level for *printk()* messages that do not have an explicit level. Up to and including Linux 2.6.38, the hard-coded default value for this field was 4 (**KERN_WARNING**); since Linux 2.6.39, the default value is defined by the kernel configuration option **CONFIG_DEFAULT_MESSAGE_LOGLEVEL**, which defaults to 4.

minimum_console_loglevel
The value in this field is the minimum value to which *console_loglevel* can be set.

default_console_loglevel
This is the default value for *console_loglevel*.

The log level
Every *printk*() message has its own log level. If the log level is not explicitly specified as part of the message, it defaults to *default_message_loglevel*. The conventional meaning of the log level is as follows:

Kernel constant **Level value** **Meaning**

KERN_EMERG	0	System is unusable
KERN_ALERT	1	Action must be taken immediately
KERN_CRIT	2	Critical conditions
KERN_ERR	3	Error conditions
KERN_WARNING	4	Warning conditions
KERN_NOTICE	5	Normal but significant condition
KERN_INFO	6	Informational
KERN_DEBUG	7	Debug-level messages

The kernel *printk()* routine will print a message on the console only if it has a log level less than the value of *console_loglevel*.

RETURN VALUE

For *type* equal to 2, 3, or 4, a successful call to **syslog**() returns the number of bytes read. For *type* 9, **syslog**() returns the number of bytes currently available to be read on the kernel log buffer. For *type* 10, **syslog**() returns the total size of the kernel log buffer. For other values of *type*, 0 is returned on success.

In case of error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: Bad arguments (e.g., bad *type*; or for *type* 2, 3, or 4, *buf* is NULL, or *len* is less than zero; or for *type* 8, the *level* is outside the range 1 to 8).

ENOSYS
: This **syslog**() system call is not available, because the kernel was compiled with the **CONFIG_PRINTK** kernel-configuration option disabled.

EPERM
: An attempt was made to change *console_loglevel* or clear the kernel message ring buffer by a process without sufficient privilege (more precisely: without the **CAP_SYS_ADMIN** or **CAP_SYSLOG** capability).

ERESTARTSYS
: System call was interrupted by a signal; nothing was read. (This can be seen only during a trace.)

STANDARDS

Linux.

HISTORY

From the very start, people noted that it is unfortunate that a system call and a library routine of the same name are entirely different animals.

SEE ALSO

dmesg(1), *syslog*(3), *capabilities*(7)

NAME

tee − duplicating pipe content

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>

ssize_t tee(int fd_in, int fd_out, size_t len, unsigned int flags);
```

DESCRIPTION

tee() duplicates up to *len* bytes of data from the pipe referred to by the file descriptor *fd_in* to the pipe referred to by the file descriptor *fd_out*. It does not consume the data that is duplicated from *fd_in*; therefore, that data can be copied by a subsequent *splice*(2).

flags is a bit mask that is composed by ORing together zero or more of the following values:

SPLICE_F_MOVE Currently has no effect for **tee**(); see *splice*(2).

SPLICE_F_NONBLOCK Do not block on I/O; see *splice*(2) for further details.

SPLICE_F_MORE Currently has no effect for **tee**(), but may be implemented in the future; see *splice*(2).

SPLICE_F_GIFT Unused for **tee**(); see *vmsplice*(2).

RETURN VALUE

Upon successful completion, **tee**() returns the number of bytes that were duplicated between the input and output. A return value of 0 means that there was no data to transfer, and it would not make sense to block, because there are no writers connected to the write end of the pipe referred to by *fd_in*.

On error, **tee**() returns −1 and *errno* is set to indicate the error.

ERRORS

EAGAIN
: **SPLICE_F_NONBLOCK** was specified in *flags* or one of the file descriptors had been marked as nonblocking (**O_NONBLOCK**), and the operation would block.

EINVAL
: *fd_in* or *fd_out* does not refer to a pipe; or *fd_in* and *fd_out* refer to the same pipe.

ENOMEM
: Out of memory.

STANDARDS

Linux.

HISTORY

Linux 2.6.17, glibc 2.5.

NOTES

Conceptually, **tee**() copies the data between the two pipes. In reality no real data copying takes place though: under the covers, **tee**() assigns data to the output by merely grabbing a reference to the input.

EXAMPLES

The example below implements a basic *tee*(1) program using the **tee**() system call. Here is an example of its use:

```
$ date | ./a.out out.log | cat
Tue Oct 28 10:06:00 CET 2014
$ cat out.log
Tue Oct 28 10:06:00 CET 2014
```

Program source

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int      fd;
    ssize_t  len, slen;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        /*
         * tee stdin to stdout.
         */
        len = tee(STDIN_FILENO, STDOUT_FILENO,
                  INT_MAX, SPLICE_F_NONBLOCK);
        if (len < 0) {
            if (errno == EAGAIN)
                continue;
            perror("tee");
            exit(EXIT_FAILURE);
        }
        if (len == 0)
            break;

        /*
         * Consume stdin by splicing it to a file.
         */
        while (len > 0) {
            slen = splice(STDIN_FILENO, NULL, fd, NULL,
                          len, SPLICE_F_MOVE);
            if (slen < 0) {
                perror("splice");
                exit(EXIT_FAILURE);
            }
            len -= slen;
        }
    }

    close(fd);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

splice(2), *vmsplice*(2), *pipe*(7)

NAME

time − get time in seconds

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

time_t time(time_t *_Nullable tloc);
```

DESCRIPTION

time() returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

If *tloc* is non-NULL, the return value is also stored in the memory pointed to by *tloc*.

RETURN VALUE

On success, the value of time in seconds since the Epoch is returned. On error, *((time_t) −1)* is returned, and *errno* is set to indicate the error.

ERRORS

EOVERFLOW

The time cannot be represented as a *time_t* value. This can happen if an executable with 32-bit *time_t* is run on a 64-bit kernel when the time is 2038-01-19 03:14:08 UTC or later. However, when the system time is out of *time_t* range in other situations, the behavior is undefined.

EFAULT

tloc points outside your accessible address space (but see BUGS).

On systems where the C library **time**() wrapper function invokes an implementation provided by the *vdso*(7) (so that there is no trap into the kernel), an invalid address may instead trigger a **SIGSEGV** signal.

VERSIONS

POSIX.1 defines *seconds since the Epoch* using a formula that approximates the number of seconds between a specified time and the Epoch. This formula takes account of the facts that all years that are evenly divisible by 4 are leap years, but years that are evenly divisible by 100 are not leap years unless they are also evenly divisible by 400, in which case they are leap years. This value is not the same as the actual number of seconds between the time and the Epoch, because of leap seconds and because system clocks are not required to be synchronized to a standard reference. Linux systems normally follow the POSIX requirement that this value ignore leap seconds, so that conforming systems interpret it consistently; see POSIX.1-2018 Rationale A.4.16.

Applications intended to run after 2038 should use ABIs with *time_t* wider than 32 bits; see *time_t*(3type).

C library/kernel differences

On some architectures, an implementation of **time**() is provided in the *vdso*(7).

STANDARDS

C11, POSIX.1-2008.

HISTORY

SVr4, 4.3BSD, C89, POSIX.1-2001.

BUGS

Error returns from this system call are indistinguishable from successful reports that the time is a few seconds *before* the Epoch, so the C library wrapper function never sets *errno* as a result of this call.

The *tloc* argument is obsolescent and should always be NULL in new code. When *tloc* is NULL, the call cannot fail.

SEE ALSO

date(1), *gettimeofday*(2), *ctime*(3), *ftime*(3), *time*(7), *vdso*(7)

NAME

timer_create − create a POSIX per-process timer

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <signal.h>          /* Definition of SIGEV_* constants */
#include <time.h>

int timer_create(clockid_t clockid,
                 struct sigevent *_Nullable restrict sevp,
                 timer_t *restrict timerid);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

timer_create():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

timer_create() creates a new per-process interval timer. The ID of the new timer is returned in the buffer pointed to by *timerid*, which must be a non-null pointer. This ID is unique within the process, until the timer is deleted. The new timer is initially disarmed.

The *clockid* argument specifies the clock that the new timer uses to measure time. It can be specified as one of the following values:

CLOCK_REALTIME
A settable system-wide real-time clock.

CLOCK_MONOTONIC
A nonsettable monotonically increasing clock that measures time from some unspecified point in the past that does not change after system startup.

CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by (all of the threads in) the calling process.

CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by the calling thread.

CLOCK_BOOTTIME (Since Linux 2.6.39)
Like **CLOCK_MONOTONIC**, this is a monotonically increasing clock. However, whereas the **CLOCK_MONOTONIC** clock does not measure the time while a system is suspended, the **CLOCK_BOOTTIME** clock does include the time during which the system is suspended. This is useful for applications that need to be suspend-aware. **CLOCK_REALTIME** is not suitable for such applications, since that clock is affected by discontinuous changes to the system clock.

CLOCK_REALTIME_ALARM (since Linux 3.0)
This clock is like **CLOCK_REALTIME**, but will wake the system if it is suspended. The caller must have the **CAP_WAKE_ALARM** capability in order to set a timer against this clock.

CLOCK_BOOTTIME_ALARM (since Linux 3.0)
This clock is like **CLOCK_BOOTTIME**, but will wake the system if it is suspended. The caller must have the **CAP_WAKE_ALARM** capability in order to set a timer against this clock.

CLOCK_TAI (since Linux 3.10)
A system-wide clock derived from wall-clock time but counting leap seconds.

See *clock_getres*(2) for some further details on the above clocks.

As well as the above values, *clockid* can be specified as the *clockid* returned by a call to *clock_getcpuclockid*(3) or *pthread_getcpuclockid*(3).

The *sevp* argument points to a *sigevent* structure that specifies how the caller should be notified when the timer expires. For the definition and general details of this structure, see *sigevent*(3type).

The *sevp.sigev_notify* field can have the following values:

SIGEV_NONE
Don't asynchronously notify when the timer expires. Progress of the timer can be monitored using *timer_gettime*(2).

SIGEV_SIGNAL
Upon timer expiration, generate the signal *sigev_signo* for the process. See *sigevent*(3type) for general details. The *si_code* field of the *siginfo_t* structure will be set to **SI_TIMER**. At any point in time, at most one signal is queued to the process for a given timer; see *timer_getoverrun*(2) for more details.

SIGEV_THREAD
Upon timer expiration, invoke *sigev_notify_function* as if it were the start function of a new thread. See *sigevent*(3type) for details.

SIGEV_THREAD_ID (Linux-specific)
As for **SIGEV_SIGNAL**, but the signal is targeted at the thread whose ID is given in *sigev_notify_thread_id*, which must be a thread in the same process as the caller. The *sigev_notify_thread_id* field specifies a kernel thread ID, that is, the value returned by *clone*(2) or *gettid*(2). This flag is intended only for use by threading libraries.

Specifying *sevp* as NULL is equivalent to specifying a pointer to a *sigevent* structure in which *sigev_notify* is **SIGEV_SIGNAL**, *sigev_signo* is **SIGALRM**, and *sigev_value.sival_int* is the timer ID.

RETURN VALUE

On success, **timer_create**() returns 0, and the ID of the new timer is placed in **timerid*. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
Temporary error during kernel allocation of timer structures.

EINVAL
Clock ID, *sigev_notify*, *sigev_signo*, or *sigev_notify_thread_id* is invalid.

ENOMEM
Could not allocate memory.

ENOTSUP
The kernel does not support creating a timer against this *clockid*.

EPERM
clockid was **CLOCK_REALTIME_ALARM** or **CLOCK_BOOTTIME_ALARM** but the caller did not have the **CAP_WAKE_ALARM** capability.

VERSIONS

C library/kernel differences

Part of the implementation of the POSIX timers API is provided by glibc. In particular:

- Much of the functionality for **SIGEV_THREAD** is implemented within glibc, rather than the kernel. (This is necessarily so, since the thread involved in handling the notification is one that must be managed by the C library POSIX threads implementation.) Although the notification delivered to the process is via a thread, internally the NPTL implementation uses a *sigev_notify* value of **SIGEV_THREAD_ID** along with a real-time signal that is reserved by the implementation (see *nptl*(7)).
- The implementation of the default case where *evp* is NULL is handled inside glibc, which invokes the underlying system call with a suitably populated *sigevent* structure.
- The timer IDs presented at user level are maintained by glibc, which maps these IDs to the timer IDs employed by the kernel.

STANDARDS

POSIX.1-2008.

HISTORY

Linux 2.6. POSIX.1-2001.

Prior to Linux 2.6, glibc provided an incomplete user-space implementation (**CLOCK_REALTIME** timers only) using POSIX threads, and before glibc 2.17, the implementation falls back to this technique on systems running kernels older than Linux 2.6.

NOTES

A program may create multiple interval timers using **timer_create**().

Timers are not inherited by the child of a *fork*(2), and are disarmed and deleted during an *execve*(2).

The kernel preallocates a "queued real-time signal" for each timer created using **timer_create**(). Consequently, the number of timers is limited by the **RLIMIT_SIGPENDING** resource limit (see *setrlimit*(2)).

The timers created by **timer_create**() are commonly known as "POSIX (interval) timers". The POSIX timers API consists of the following interfaces:

timer_create()
: Create a timer.

timer_settime(2)
: Arm (start) or disarm (stop) a timer.

timer_gettime(2)
: Fetch the time remaining until the next expiration of a timer, along with the interval setting of the timer.

timer_getoverrun(2)
: Return the overrun count for the last timer expiration.

timer_delete(2)
: Disarm and delete a timer.

Since Linux 3.10, the */proc/*pid*/timers* file can be used to list the POSIX timers for the process with PID *pid*. See *proc*(5) for further information.

Since Linux 4.10, support for POSIX timers is a configurable option that is enabled by default. Kernel support can be disabled via the **CONFIG_POSIX_TIMERS** option.

EXAMPLES

The program below takes two arguments: a sleep period in seconds, and a timer frequency in nanoseconds. The program establishes a handler for the signal it uses for the timer, blocks that signal, creates and arms a timer that expires with the given frequency, sleeps for the specified number of seconds, and then unblocks the timer signal. Assuming that the timer expired at least once while the program slept, the signal handler will be invoked, and the handler displays some information about the timer notification. The program terminates after one invocation of the signal handler.

In the following example run, the program sleeps for 1 second, after creating a timer that has a frequency of 100 nanoseconds. By the time the signal is unblocked and delivered, there have been around ten million overruns.

```
$ ./a.out 1 100
Establishing handler for signal 34
Blocking signal 34
timer ID is 0x804c008
Sleeping for 1 seconds
Unblocking signal 34
Caught signal 34
    sival_ptr = 0xbfb174f4;     *sival_ptr = 0x804c008
    overrun count = 10004886
```

Program source

```
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define CLOCKID CLOCK_REALTIME
#define SIG SIGRTMIN

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static void
print_siginfo(siginfo_t *si)
{
    int      or;
    timer_t  *tidp;

    tidp = si->si_value.sival_ptr;

    printf("    sival_ptr = %p; ", si->si_value.sival_ptr);
    printf("    *sival_ptr = %#jx\n", (uintmax_t) *tidp);

    or = timer_getoverrun(*tidp);
    if (or == -1)
        errExit("timer_getoverrun");
    else
        printf("    overrun count = %d\n", or);
}

static void
handler(int sig, siginfo_t *si, void *uc)
{
    /* Note: calling printf() from a signal handler is not safe
       (and should not be done in production programs), since
       printf() is not async-signal-safe; see signal-safety(7).
       Nevertheless, we use printf() here as a simple way of
       showing that the handler was called. */

    printf("Caught signal %d\n", sig);
    print_siginfo(si);
    signal(sig, SIG_IGN);
}

int
main(int argc, char *argv[])
{
    timer_t            timerid;
    sigset_t           mask;
    long long          freq_nanosecs;
    struct sigevent    sev;
    struct sigaction   sa;
    struct itimerspec  its;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <sleep-secs> <freq-nanosecs>\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Establish handler for timer signal. */
```

```
        printf("Establishing handler for signal %d\n", SIG);
        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = handler;
        sigemptyset(&sa.sa_mask);
        if (sigaction(SIG, &sa, NULL) == -1)
            errExit("sigaction");

        /* Block timer signal temporarily. */

        printf("Blocking signal %d\n", SIG);
        sigemptyset(&mask);
        sigaddset(&mask, SIG);
        if (sigprocmask(SIG_SETMASK, &mask, NULL) == -1)
            errExit("sigprocmask");

        /* Create the timer. */

        sev.sigev_notify = SIGEV_SIGNAL;
        sev.sigev_signo = SIG;
        sev.sigev_value.sival_ptr = &timerid;
        if (timer_create(CLOCKID, &sev, &timerid) == -1)
            errExit("timer_create");

        printf("timer ID is %#jx\n", (uintmax_t) timerid);

        /* Start the timer. */

        freq_nanosecs = atoll(argv[2]);
        its.it_value.tv_sec = freq_nanosecs / 1000000000;
        its.it_value.tv_nsec = freq_nanosecs % 1000000000;
        its.it_interval.tv_sec = its.it_value.tv_sec;
        its.it_interval.tv_nsec = its.it_value.tv_nsec;

        if (timer_settime(timerid, 0, &its, NULL) == -1)
             errExit("timer_settime");

        /* Sleep for a while; meanwhile, the timer may expire
           multiple times. */

        printf("Sleeping for %d seconds\n", atoi(argv[1]));
        sleep(atoi(argv[1]));

        /* Unlock the timer signal, so that timer notification
           can be delivered. */

        printf("Unblocking signal %d\n", SIG);
        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) == -1)
            errExit("sigprocmask");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

clock_gettime(2), *setitimer*(2), *timer_delete*(2), *timer_getoverrun*(2), *timer_settime*(2), *timerfd_create*(2), *clock_getcpuclockid*(3), *pthread_getcpuclockid*(3), *pthreads*(7), *sigevent*(3type), *signal*(7), *time*(7)

NAME

timer_delete − delete a POSIX per-process timer

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <time.h>

int timer_delete(timer_t *timerid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

timer_delete():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

timer_delete() deletes the timer whose ID is given in *timerid*. If the timer was armed at the time of this call, it is disarmed before being deleted. The treatment of any pending signal generated by the deleted timer is unspecified.

RETURN VALUE

On success, **timer_delete**() returns 0. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
timerid is not a valid timer ID.

STANDARDS

POSIX.1-2008.

HISTORY

Linux 2.6. POSIX.1-2001.

SEE ALSO

clock_gettime(2), *timer_create*(2), *timer_getoverrun*(2), *timer_settime*(2), *time*(7)

NAME

timer_getoverrun − get overrun count for a POSIX per-process timer

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <time.h>

int timer_getoverrun(timer_t *timerid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

timer_getoverrun():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

timer_getoverrun() returns the "overrun count" for the timer referred to by *timerid*. An application can use the overrun count to accurately calculate the number of timer expirations that would have occurred over a given time interval. Timer overruns can occur both when receiving expiration notifications via signals (**SIGEV_SIGNAL**), and via threads (**SIGEV_THREAD**).

When expiration notifications are delivered via a signal, overruns can occur as follows. Regardless of whether or not a real-time signal is used for timer notifications, the system queues at most one signal per timer. (This is the behavior specified by POSIX.1. The alternative, queuing one signal for each timer expiration, could easily result in overflowing the allowed limits for queued signals on the system.) Because of system scheduling delays, or because the signal may be temporarily blocked, there can be a delay between the time when the notification signal is generated and the time when it is delivered (e.g., caught by a signal handler) or accepted (e.g., using *sigwaitinfo*(2)). In this interval, further timer expirations may occur. The timer overrun count is the number of additional timer expirations that occurred between the time when the signal was generated and when it was delivered or accepted.

Timer overruns can also occur when expiration notifications are delivered via invocation of a thread, since there may be an arbitrary delay between an expiration of the timer and the invocation of the notification thread, and in that delay interval, additional timer expirations may occur.

RETURN VALUE

On success, **timer_getoverrun**() returns the overrun count of the specified timer; this count may be 0 if no overruns have occurred. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
timerid is not a valid timer ID.

VERSIONS

When timer notifications are delivered via signals (**SIGEV_SIGNAL**), on Linux it is also possible to obtain the overrun count via the *si_overrun* field of the *siginfo_t* structure (see *sigaction*(2)). This allows an application to avoid the overhead of making a system call to obtain the overrun count, but is a nonportable extension to POSIX.1.

POSIX.1 discusses timer overruns only in the context of timer notifications using signals.

STANDARDS

POSIX.1-2008.

HISTORY

Linux 2.6. POSIX.1-2001.

BUGS

POSIX.1 specifies that if the timer overrun count is equal to or greater than an implementation-defined maximum, **DELAYTIMER_MAX**, then **timer_getoverrun**() should return **DELAYTIMER_MAX**. However, before Linux 4.19, if the timer overrun value exceeds the maximum representable integer, the counter cycles, starting once more from low values. Since Linux 4.19, **timer_getoverrun**() returns **DELAYTIMER_MAX** (defined as **INT_MAX** in <*limits.h*>) in this case (and the overrun value is reset to 0).

EXAMPLES

See *timer_create*(2).

SEE ALSO

clock_gettime(2), *sigaction*(2), *signalfd*(2), *sigwaitinfo*(2), *timer_create*(2), *timer_delete*(2), *timer_settime*(2), *signal*(7), *time*(7)

NAME

timer_settime, timer_gettime − arm/disarm and fetch state of POSIX per-process timer

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <time.h>

int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict new_value,
                  struct itimerspec *_Nullable restrict old_value);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

timer_settime(), **timer_gettime**():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

timer_settime() arms or disarms the timer identified by *timerid*. The *new_value* argument is pointer to an *itimerspec* structure that specifies the new initial value and the new interval for the timer. The *itimerspec* structure is described in *itimerspec*(3type).

Each of the substructures of the *itimerspec* structure is a *timespec*(3) structure that allows a time value to be specified in seconds and nanoseconds. These time values are measured according to the clock that was specified when the timer was created by *timer_create*(2).

If *new_value−>it_value* specifies a nonzero value (i.e., either subfield is nonzero), then **timer_settime**() arms (starts) the timer, setting it to initially expire at the given time. (If the timer was already armed, then the previous settings are overwritten.) If *new_value−>it_value* specifies a zero value (i.e., both subfields are zero), then the timer is disarmed.

The *new_value−>it_interval* field specifies the period of the timer, in seconds and nanoseconds. If this field is nonzero, then each time that an armed timer expires, the timer is reloaded from the value specified in *new_value−>it_interval*. If *new_value−>it_interval* specifies a zero value, then the timer expires just once, at the time specified by *it_value*.

By default, the initial expiration time specified in *new_value−>it_value* is interpreted relative to the current time on the timer's clock at the time of the call. This can be modified by specifying **TIMER_ABSTIME** in *flags*, in which case *new_value−>it_value* is interpreted as an absolute value as measured on the timer's clock; that is, the timer will expire when the clock value reaches the value specified by *new_value−>it_value*. If the specified absolute time has already passed, then the timer expires immediately, and the overrun count (see *timer_getoverrun*(2)) will be set correctly.

If the value of the **CLOCK_REALTIME** clock is adjusted while an absolute timer based on that clock is armed, then the expiration of the timer will be appropriately adjusted. Adjustments to the **CLOCK_REALTIME** clock have no effect on relative timers based on that clock.

If *old_value* is not NULL, then it points to a buffer that is used to return the previous interval of the timer (in *old_value−>it_interval*) and the amount of time until the timer would previously have next expired (in *old_value−>it_value*).

timer_gettime() returns the time until next expiration, and the interval, for the timer specified by *timerid*, in the buffer pointed to by *curr_value*. The time remaining until the next timer expiration is returned in *curr_value−>it_value*; this is always a relative value, regardless of whether the **TIMER_ABSTIME** flag was used when arming the timer. If the value returned in *curr_value−>it_value* is zero, then the timer is currently disarmed. The timer interval is returned in *curr_value−>it_interval*. If the value returned in *curr_value−>it_interval* is zero, then this is a "one-shot" timer.

RETURN VALUE

On success, **timer_settime**() and **timer_gettime**() return 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

These functions may fail with the following errors:

EFAULT
new_value, *old_value*, or *curr_value* is not a valid pointer.

EINVAL
timerid is invalid.

timer_settime() may fail with the following errors:

EINVAL
new_value.it_value is negative; or *new_value.it_value.tv_nsec* is negative or greater than 999,999,999.

STANDARDS
POSIX.1-2008.

HISTORY
Linux 2.6. POSIX.1-2001.

EXAMPLES
See *timer_create*(2).

SEE ALSO
timer_create(2), *timer_getoverrun*(2), *timespec*(3), *time*(7)

NAME

timerfd_create, timerfd_settime, timerfd_gettime − timers that notify via file descriptors

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);

int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspec *_Nullable old_value);
int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

DESCRIPTION

These system calls create and operate on a timer that delivers timer expiration notifications via a file descriptor. They provide an alternative to the use of *setitimer*(2) or *timer_create*(2), with the advantage that the file descriptor may be monitored by *select*(2), *poll*(2), and *epoll*(7).

The use of these three system calls is analogous to the use of *timer_create*(2), *timer_settime*(2), and *timer_gettime*(2). (There is no analog of *timer_getoverrun*(2), since that functionality is provided by *read*(2), as described below.)

timerfd_create()

timerfd_create() creates a new timer object, and returns a file descriptor that refers to that timer. The *clockid* argument specifies the clock that is used to mark the progress of the timer, and must be one of the following:

CLOCK_REALTIME
: A settable system-wide real-time clock.

CLOCK_MONOTONIC
: A nonsettable monotonically increasing clock that measures time from some unspecified point in the past that does not change after system startup.

CLOCK_BOOTTIME (Since Linux 3.15)
: Like **CLOCK_MONOTONIC**, this is a monotonically increasing clock. However, whereas the **CLOCK_MONOTONIC** clock does not measure the time while a system is suspended, the **CLOCK_BOOTTIME** clock does include the time during which the system is suspended. This is useful for applications that need to be suspend-aware. **CLOCK_REALTIME** is not suitable for such applications, since that clock is affected by discontinuous changes to the system clock.

CLOCK_REALTIME_ALARM (since Linux 3.11)
: This clock is like **CLOCK_REALTIME**, but will wake the system if it is suspended. The caller must have the **CAP_WAKE_ALARM** capability in order to set a timer against this clock.

CLOCK_BOOTTIME_ALARM (since Linux 3.11)
: This clock is like **CLOCK_BOOTTIME**, but will wake the system if it is suspended. The caller must have the **CAP_WAKE_ALARM** capability in order to set a timer against this clock.

See *clock_getres*(2) for some further details on the above clocks.

The current value of each of these clocks can be retrieved using *clock_gettime*(2).

Starting with Linux 2.6.27, the following values may be bitwise ORed in *flags* to change the behavior of **timerfd_create**():

TFD_NONBLOCK
: Set the **O_NONBLOCK** file status flag on the open file description (see *open*(2)) referred to by the new file descriptor. Using this flag saves extra calls to *fcntl*(2) to achieve the same result.

TFD_CLOEXEC
Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

In Linux versions up to and including 2.6.26, *flags* must be specified as zero.

timerfd_settime()

timerfd_settime() arms (starts) or disarms (stops) the timer referred to by the file descriptor *fd*.

The *new_value* argument specifies the initial expiration and interval for the timer. The *itimerspec* structure used for this argument is described in *itimerspec*(3type).

new_value.it_value specifies the initial expiration of the timer, in seconds and nanoseconds. Setting either field of *new_value.it_value* to a nonzero value arms the timer. Setting both fields of *new_value.it_value* to zero disarms the timer.

Setting one or both fields of *new_value.it_interval* to nonzero values specifies the period, in seconds and nanoseconds, for repeated timer expirations after the initial expiration. If both fields of *new_value.it_interval* are zero, the timer expires just once, at the time specified by *new_value.it_value*.

By default, the initial expiration time specified in *new_value* is interpreted relative to the current time on the timer's clock at the time of the call (i.e., *new_value.it_value* specifies a time relative to the current value of the clock specified by *clockid*). An absolute timeout can be selected via the *flags* argument.

The *flags* argument is a bit mask that can include the following values:

TFD_TIMER_ABSTIME
Interpret *new_value.it_value* as an absolute value on the timer's clock. The timer will expire when the value of the timer's clock reaches the value specified in *new_value.it_value*.

TFD_TIMER_CANCEL_ON_SET
If this flag is specified along with **TFD_TIMER_ABSTIME** and the clock for this timer is **CLOCK_REALTIME** or **CLOCK_REALTIME_ALARM**, then mark this timer as cancelable if the real-time clock undergoes a discontinuous change (**settimeofday**(2), *clock_settime*(2), or similar). When such changes occur, a current or future *read*(2) from the file descriptor will fail with the error **ECANCELED**.

If the *old_value* argument is not NULL, then the *itimerspec* structure that it points to is used to return the setting of the timer that was current at the time of the call; see the description of **timerfd_gettime**() following.

timerfd_gettime()

timerfd_gettime() returns, in *curr_value*, an *itimerspec* structure that contains the current setting of the timer referred to by the file descriptor *fd*.

The *it_value* field returns the amount of time until the timer will next expire. If both fields of this structure are zero, then the timer is currently disarmed. This field always contains a relative value, regardless of whether the **TFD_TIMER_ABSTIME** flag was specified when setting the timer.

The *it_interval* field returns the interval of the timer. If both fields of this structure are zero, then the timer is set to expire just once, at the time specified by *curr_value.it_value*.

Operating on a timer file descriptor

The file descriptor returned by **timerfd_create**() supports the following additional operations:

read(2) If the timer has already expired one or more times since its settings were last modified using **timerfd_settime**(), or since the last successful *read*(2), then the buffer given to *read*(2) returns an unsigned 8-byte integer (*uint64_t*) containing the number of expirations that have occurred. (The returned value is in host byte order—that is, the native byte order for integers on the host machine.)

If no timer expirations have occurred at the time of the *read*(2), then the call either blocks until the next timer expiration, or fails with the error **EAGAIN** if the file descriptor has been made nonblocking (via the use of the *fcntl*(2) **F_SETFL** operation to set the **O_NONBLOCK** flag).

A *read*(2) fails with the error **EINVAL** if the size of the supplied buffer is less than 8 bytes.

If the associated clock is either **CLOCK_REALTIME** or **CLOCK_REALTIME_ALARM**, the timer is absolute (**TFD_TIMER_ABSTIME**), and the flag **TFD_TIMER_CANCEL_ON_SET** was specified when calling **timerfd_settime**(), then *read*(2) fails with the error **ECANCELED** if the real-time clock undergoes a discontinuous change. (This allows the reading application to discover such discontinuous changes to the clock.)

If the associated clock is either **CLOCK_REALTIME** or **CLOCK_REALTIME_ALARM**, the timer is absolute (**TFD_TIMER_ABSTIME**), and the flag **TFD_TIMER_CANCEL_ON_SET** was *not* specified when calling **timerfd_settime**(), then a discontinuous negative change to the clock (e.g., *clock_settime*(2)) may cause *read*(2) to unblock, but return a value of 0 (i.e., no bytes read), if the clock change occurs after the time expired, but before the *read*(2) on the file descriptor.

poll(2)
select(2)
(and similar)

The file descriptor is readable (the *select*(2) *readfds* argument; the *poll*(2) **POLLIN** flag) if one or more timer expirations have occurred.

The file descriptor also supports the other file-descriptor multiplexing APIs: *pselect*(2), *ppoll*(2), and *epoll*(7).

ioctl(2) The following timerfd-specific command is supported:

TFD_IOC_SET_TICKS (since Linux 3.17)

Adjust the number of timer expirations that have occurred. The argument is a pointer to a nonzero 8-byte integer (*uint64_t**) containing the new number of expirations. Once the number is set, any waiter on the timer is woken up. The only purpose of this command is to restore the expirations for the purpose of checkpoint/restore. This operation is available only if the kernel was configured with the **CONFIG_CHECKPOINT_RESTORE** option.

close(2)

When the file descriptor is no longer required it should be closed. When all file descriptors associated with the same timer object have been closed, the timer is disarmed and its resources are freed by the kernel.

fork(2) semantics

After a *fork*(2), the child inherits a copy of the file descriptor created by **timerfd_create**(). The file descriptor refers to the same underlying timer object as the corresponding file descriptor in the parent, and *read*(2)s in the child will return information about expirations of the timer.

execve(2) semantics

A file descriptor created by **timerfd_create**() is preserved across *execve*(2), and continues to generate timer expirations if the timer was armed.

RETURN VALUE

On success, **timerfd_create**() returns a new file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

timerfd_settime() and **timerfd_gettime**() return 0 on success; on error they return −1, and set *errno* to indicate the error.

ERRORS

timerfd_create() can fail with the following errors:

EINVAL

The *clockid* is not valid.

EINVAL

flags is invalid; or, in Linux 2.6.26 or earlier, *flags* is nonzero.

EMFILE

The per-process limit on the number of open file descriptors has been reached.

ENFILE

The system-wide limit on the total number of open files has been reached.

ENODEV
Could not mount (internal) anonymous inode device.

ENOMEM
There was insufficient kernel memory to create the timer.

EPERM
clockid was **CLOCK_REALTIME_ALARM** or **CLOCK_BOOTTIME_ALARM** but the caller did not have the **CAP_WAKE_ALARM** capability.

timerfd_settime() and **timerfd_gettime**() can fail with the following errors:

EBADF
fd is not a valid file descriptor.

EFAULT
new_value, *old_value*, or *curr_value* is not a valid pointer.

EINVAL
fd is not a valid timerfd file descriptor.

timerfd_settime() can also fail with the following errors:

ECANCELED
See NOTES.

EINVAL
new_value is not properly initialized (one of the *tv_nsec* falls outside the range zero to 999,999,999).

EINVAL
flags is invalid.

STANDARDS
Linux.

HISTORY
Linux 2.6.25, glibc 2.8.

NOTES
Suppose the following scenario for **CLOCK_REALTIME** or **CLOCK_REALTIME_ALARM** timer that was created with **timerfd_create**():

(1) The timer has been started (**timerfd_settime**()) with the **TFD_TIMER_ABSTIME** and **TFD_TIMER_CANCEL_ON_SET** flags;

(2) A discontinuous change (e.g., *settimeofday*(2)) is subsequently made to the **CLOCK_REALTIME** clock; and

(3) the caller once more calls **timerfd_settime**() to rearm the timer (without first doing a *read*(2) on the file descriptor).

In this case the following occurs:

- The **timerfd_settime**() returns −1 with *errno* set to **ECANCELED**. (This enables the caller to know that the previous timer was affected by a discontinuous change to the clock.)
- The timer *is successfully rearmed* with the settings provided in the second **timerfd_settime**() call. (This was probably an implementation accident, but won't be fixed now, in case there are applications that depend on this behaviour.)

BUGS
Currently, **timerfd_create**() supports fewer types of clock IDs than *timer_create*(2).

EXAMPLES
The following program creates a timer and then monitors its progress. The program accepts up to three command-line arguments. The first argument specifies the number of seconds for the initial expiration of the timer. The second argument specifies the interval for the timer, in seconds. The third argument specifies the number of times the program should allow the timer to expire before terminating. The second and third command-line arguments are optional.

The following shell session demonstrates the use of the program:

```
$ a.out 3 1 100
0.000: timer started
3.000: read: 1; total=1
4.000: read: 1; total=2
^Z                  # type control-Z to suspend the program
[1]+  Stopped                 ./timerfd3_demo 3 1 100
$ fg                # Resume execution after a few seconds
a.out 3 1 100
9.660: read: 5; total=7
10.000: read: 1; total=8
11.000: read: 1; total=9
^C                  # type control-C to suspend the program
```

Program source

```
#include <err.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/timerfd.h>
#include <time.h>
#include <unistd.h>

static void
print_elapsed_time(void)
{
    int                     secs, nsecs;
    static int              first_call = 1;
    struct timespec         curr;
    static struct timespec  start;

    if (first_call) {
        first_call = 0;
        if (clock_gettime(CLOCK_MONOTONIC, &start) == -1)
            err(EXIT_FAILURE, "clock_gettime");
    }

    if (clock_gettime(CLOCK_MONOTONIC, &curr) == -1)
        err(EXIT_FAILURE, "clock_gettime");

    secs = curr.tv_sec - start.tv_sec;
    nsecs = curr.tv_nsec - start.tv_nsec;
    if (nsecs < 0) {
        secs--;
        nsecs += 1000000000;
    }
    printf("%d.%03d: ", secs, (nsecs + 500000) / 1000000);
}

int
main(int argc, char *argv[])
{
    int                fd;
    ssize_t            s;
    uint64_t           exp, tot_exp, max_exp;
    struct timespec    now;
    struct itimerspec  new_value;

    if (argc != 2 && argc != 4) {
```

```
            fprintf(stderr, "%s init-secs [interval-secs max-exp]\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }

        if (clock_gettime(CLOCK_REALTIME, &now) == -1)
            err(EXIT_FAILURE, "clock_gettime");

        /* Create a CLOCK_REALTIME absolute timer with initial
           expiration and interval as specified in command line. */

        new_value.it_value.tv_sec = now.tv_sec + atoi(argv[1]);
        new_value.it_value.tv_nsec = now.tv_nsec;
        if (argc == 2) {
            new_value.it_interval.tv_sec = 0;
            max_exp = 1;
        } else {
            new_value.it_interval.tv_sec = atoi(argv[2]);
            max_exp = atoi(argv[3]);
        }
        new_value.it_interval.tv_nsec = 0;

        fd = timerfd_create(CLOCK_REALTIME, 0);
        if (fd == -1)
            err(EXIT_FAILURE, "timerfd_create");

        if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new_value, NULL) == -1)
            err(EXIT_FAILURE, "timerfd_settime");

        print_elapsed_time();
        printf("timer started\n");

        for (tot_exp = 0; tot_exp < max_exp;) {
            s = read(fd, &exp, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
                err(EXIT_FAILURE, "read");

            tot_exp += exp;
            print_elapsed_time();
            printf("read: %" PRIu64 "; total=%" PRIu64 "\n", exp, tot_exp);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

eventfd(2), *poll*(2), *read*(2), *select*(2), *setitimer*(2), *signalfd*(2), *timer_create*(2), *timer_gettime*(2), *timer_settime*(2), *timespec*(3), *epoll*(7), *time*(7)

NAME

times − get process times

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/times.h>

clock_t times(struct tms * *buf* **);**

DESCRIPTION

times() stores the current process times in the *struct tms* that *buf* points to. The *struct tms* is as defined in <*sys/times.h*>:

```
struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};
```

The *tms_utime* field contains the CPU time spent executing instructions of the calling process. The *tms_stime* field contains the CPU time spent executing inside the kernel while performing tasks on behalf of the calling process.

The *tms_cutime* field contains the sum of the *tms_utime* and *tms_cutime* values for all waited-for terminated children. The *tms_cstime* field contains the sum of the *tms_stime* and *tms_cstime* values for all waited-for terminated children.

Times for terminated children (and their descendants) are added in at the moment *wait*(2) or *waitpid*(2) returns their process ID. In particular, times of grandchildren that the children did not wait for are never seen.

All times reported are in clock ticks.

RETURN VALUE

times() returns the number of clock ticks that have elapsed since an arbitrary point in the past. The return value may overflow the possible range of type *clock_t*. On error, *(clock_t) −1* is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT

tms points outside the process's address space.

VERSIONS

On Linux, the *buf* argument can be specified as NULL, with the result that **times**() just returns a function result. However, POSIX does not specify this behavior, and most other UNIX implementations require a non-NULL value for *buf*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

In POSIX.1-1996 the symbol **CLK_TCK** (defined in <*time.h*>) is mentioned as obsolescent. It is obsolete now.

Before Linux 2.6.9, if the disposition of **SIGCHLD** is set to **SIG_IGN**, then the times of terminated children are automatically included in the *tms_cstime* and *tms_cutime* fields, although POSIX.1-2001 says that this should happen only if the calling process *wait*(2)s on its children. This nonconformance is rectified in Linux 2.6.9 and later.

On Linux, the "arbitrary point in the past" from which the return value of **times**() is measured has varied across kernel versions. On Linux 2.4 and earlier, this point is the moment the system was booted. Since Linux 2.6, this point is *(2^32/HZ) − 300* seconds before system boot time. This variability across kernel versions (and across UNIX implementations), combined with the fact that the returned value may overflow the range of *clock_t*, means that a portable application would be wise to avoid using this

value. To measure changes in elapsed time, use *clock_gettime*(2) instead.

SVr1-3 returns *long* and the struct members are of type *time_t* although they store clock ticks, not seconds since the Epoch. V7 used *long* for the struct members, because it had no type *time_t* yet.

NOTES

The number of clock ticks per second can be obtained using:

```
sysconf(_SC_CLK_TCK);
```

Note that *clock*(3) also returns a value of type *clock_t*, but this value is measured in units of **CLOCKS_PER_SEC**, not the clock ticks used by **times**().

BUGS

A limitation of the Linux system call conventions on some architectures (notably i386) means that on Linux 2.6 there is a small time window (41 seconds) soon after boot when **times**() can return −1, falsely indicating that an error occurred. The same problem can occur when the return value wraps past the maximum value that can be stored in **clock_t**.

SEE ALSO

time(1), *getrusage*(2), *wait*(2), *clock*(3), *sysconf*(3), *time*(7)

NAME

tkill, tgkill − send a signal to a thread

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>          /* Definition of SIG* constants */
#include <sys/syscall.h>     /* Definition of SYS_* constants */
#include <unistd.h>

[[deprecated]] int syscall(SYS_tkill, pid_t tid, int sig);

#include <signal.h>

int tgkill(pid_t tgid, pid_t tid, int sig);
```

Note: glibc provides no wrapper for **tkill**(), necessitating the use of *syscall*(2).

DESCRIPTION

tgkill() sends the signal *sig* to the thread with the thread ID *tid* in the thread group *tgid*. (By contrast, *kill*(2) can be used to send a signal only to a process (i.e., thread group) as a whole, and the signal will be delivered to an arbitrary thread within that process.)

tkill() is an obsolete predecessor to **tgkill**(). It allows only the target thread ID to be specified, which may result in the wrong thread being signaled if a thread terminates and its thread ID is recycled. Avoid using this system call.

These are the raw system call interfaces, meant for internal thread library use.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: The **RLIMIT_SIGPENDING** resource limit was reached and *sig* is a real-time signal.

EAGAIN
: Insufficient kernel memory was available and *sig* is a real-time signal.

EINVAL
: An invalid thread ID, thread group ID, or signal was specified.

EPERM
: Permission denied. For the required permissions, see *kill*(2).

ESRCH
: No process with the specified thread ID (and thread group ID) exists.

STANDARDS

Linux.

HISTORY

tkill() Linux 2.4.19 / 2.5.4.

tgkill() Linux 2.5.75, glibc 2.30.

NOTES

See the description of **CLONE_THREAD** in *clone*(2) for an explanation of thread groups.

SEE ALSO

clone(2), *gettid*(2), *kill*(2), *rt_sigqueueinfo*(2)

NAME

truncate, ftruncate − truncate a file to a specified length

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int truncate(const char *path, off_t length);
int ftruncate(int  fd, off_t length);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

truncate():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE

ftruncate():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.3.5: */ _POSIX_C_SOURCE >= 200112L
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

The **truncate**() and **ftruncate**() functions cause the regular file named by *path* or referenced by *fd* to be truncated to a size of precisely *length* bytes.

If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0').

The file offset is not changed.

If the size changed, then the st_ctime and st_mtime fields (respectively, time of last status change and time of last modification; see *inode*(7)) for the file are updated, and the set-user-ID and set-group-ID mode bits may be cleared.

With **ftruncate**(), the file must be open for writing; with **truncate**(), the file must be writable.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

For **truncate**():

EACCES
Search permission is denied for a component of the path prefix, or the named file is not writable by the user. (See also *path_resolution*(7).)

EFAULT
The argument *path* points outside the process's allocated address space.

EFBIG
The argument *length* is larger than the maximum file size. (XSI)

EINTR
While blocked waiting to complete, the call was interrupted by a signal handler; see *fcntl*(2) and *signal*(7).

EINVAL
The argument *length* is negative or larger than the maximum file size.

EIO An I/O error occurred updating the inode.

EISDIR
The named file is a directory.

ELOOP
Too many symbolic links were encountered in translating the pathname.

ENAMETOOLONG
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

ENOENT
The named file does not exist.

ENOTDIR
A component of the path prefix is not a directory.

EPERM
The underlying filesystem does not support extending a file beyond its current size.

EPERM
The operation was prevented by a file seal; see *fcntl*(2).

EROFS
The named file resides on a read-only filesystem.

ETXTBSY
The file is an executable file that is being executed.

For **ftruncate**() the same errors apply, but instead of things that can be wrong with *path*, we now have things that can be wrong with the file descriptor, *fd*:

EBADF
fd is not a valid file descriptor.

EBADF or **EINVAL**
fd is not open for writing.

EINVAL
fd does not reference a regular file or a POSIX shared memory object.

EINVAL or **EBADF**
The file descriptor *fd* is not open for writing. POSIX permits, and portable applications should handle, either error for this case. (Linux produces **EINVAL**.)

VERSIONS

The details in DESCRIPTION are for XSI-compliant systems. For non-XSI-compliant systems, the POSIX standard allows two behaviors for **ftruncate**() when *length* exceeds the file length (note that **truncate**() is not specified at all in such an environment): either returning an error, or extending the file. Like most UNIX implementations, Linux follows the XSI requirement when dealing with native filesystems. However, some nonnative filesystems do not permit **truncate**() and **ftruncate**() to be used to extend a file beyond its current length: a notable example on Linux is VFAT.

On some 32-bit architectures, the calling signature for these system calls differ, for the reasons described in *syscall*(2).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.4BSD, SVr4 (first appeared in 4.2BSD).

The original Linux **truncate**() and **ftruncate**() system calls were not designed to handle large file offsets. Consequently, Linux 2.4 added **truncate64**() and **ftruncate64**() system calls that handle large files. However, these details can be ignored by applications using glibc, whose wrapper functions transparently employ the more recent system calls where they are available.

NOTES

ftruncate() can also be used to set the size of a POSIX shared memory object; see *shm_open*(3).

BUGS

A header file bug in glibc 2.12 meant that the minimum value of **_POSIX_C_SOURCE** required to expose the declaration of **ftruncate**() was 200809L instead of 200112L. This has been fixed in later glibc versions.

SEE ALSO

truncate(1), *open*(2), *stat*(2), *path_resolution*(7)

NAME

umask − set file mode creation mask

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/stat.h>

mode_t umask(mode_t *mask***);**

DESCRIPTION

umask() sets the calling process's file mode creation mask (umask) to *mask* & 0777 (i.e., only the file permission bits of *mask* are used), and returns the previous value of the mask.

The umask is used by *open*(2), *mkdir*(2), and other system calls that create files to modify the permissions placed on newly created files or directories. Specifically, permissions in the umask are turned off from the *mode* argument to *open*(2) and *mkdir*(2).

Alternatively, if the parent directory has a default ACL (see *acl*(5)), the umask is ignored, the default ACL is inherited, the permission bits are set based on the inherited ACL, and permission bits absent in the *mode* argument are turned off. For example, the following default ACL is equivalent to a umask of 022:

```
u::rwx,g::r-x,o::r-x
```

Combining the effect of this default ACL with a *mode* argument of 0666 (rw-rw-rw-), the resulting file permissions would be 0644 (rw-r--r--).

The constants that should be used to specify *mask* are described in *inode*(7).

The typical default value for the process umask is **S_IWGRP** | **S_IWOTH** (octal 022). In the usual case where the *mode* argument to *open*(2) is specified as:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

(octal 0666) when creating a new file, the permissions on the resulting file will be:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

(because 0666 & ~022 = 0644; i.e. rw−r−−r−−).

RETURN VALUE

This system call always succeeds and the previous value of the mask is returned.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

A child process created via *fork*(2) inherits its parent's umask. The umask is left unchanged by *execve*(2).

It is impossible to use **umask**() to fetch a process's umask without at the same time changing it. A second call to **umask**() would then be needed to restore the umask. The nonatomicity of these two steps provides the potential for races in multithreaded programs.

Since Linux 4.7, the umask of any process can be viewed via the *Umask* field of */proc/*pid*/status*. Inspecting this field in */proc/self/status* allows a process to retrieve its umask without at the same time changing it.

The umask setting also affects the permissions assigned to POSIX IPC objects (**mq_open**(3), *sem_open*(3), *shm_open*(3)), FIFOs (**mkfifo**(3)), and UNIX domain sockets (**unix**(7)) created by the process. The umask does not affect the permissions assigned to System V IPC objects created by the process (using *msgget*(2), *semget*(2), *shmget*(2)).

SEE ALSO

chmod(2), *mkdir*(2), *open*(2), *stat*(2), *acl*(5)

NAME

umount, umount2 − unmount filesystem

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/mount.h>

int umount(const char *target);
int umount2(const char *target, int  flags);
```

DESCRIPTION

umount() and **umount2**() remove the attachment of the (topmost) filesystem mounted on *target*.

Appropriate privilege (Linux: the **CAP_SYS_ADMIN** capability) is required to unmount filesystems.

Linux 2.1.116 added the **umount2**() system call, which, like **umount**(), unmounts a target, but allows additional *flags* controlling the behavior of the operation:

MNT_FORCE (since Linux 2.1.116)
: Ask the filesystem to abort pending requests before attempting the unmount. This may allow the unmount to complete without waiting for an inaccessible server, but could cause data loss. If, after aborting requests, some processes still have active references to the filesystem, the unmount will still fail. As at Linux 4.12, **MNT_FORCE** is supported only on the following filesystems: 9p (since Linux 2.6.16), ceph (since Linux 2.6.34), cifs (since Linux 2.6.12), fuse (since Linux 2.6.16), lustre (since Linux 3.11), and NFS (since Linux 2.1.116).

MNT_DETACH (since Linux 2.4.11)
: Perform a lazy unmount: make the mount unavailable for new accesses, immediately disconnect the filesystem and all filesystems mounted below it from each other and from the mount table, and actually perform the unmount when the mount ceases to be busy.

MNT_EXPIRE (since Linux 2.6.8)
: Mark the mount as expired. If a mount is not currently in use, then an initial call to **umount2**() with this flag fails with the error **EAGAIN**, but marks the mount as expired. The mount remains expired as long as it isn't accessed by any process. A second **umount2**() call specifying **MNT_EXPIRE** unmounts an expired mount. This flag cannot be specified with either **MNT_FORCE** or **MNT_DETACH**.

UMOUNT_NOFOLLOW (since Linux 2.6.34)
: Don't dereference *target* if it is a symbolic link. This flag allows security problems to be avoided in set-user-ID-*root* programs that allow unprivileged users to unmount filesystems.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The error values given below result from filesystem type independent errors. Each filesystem type may have its own special errors and its own special behavior. See the Linux kernel source code for details.

EAGAIN
: A call to **umount2**() specifying **MNT_EXPIRE** successfully marked an unbusy filesystem as expired.

EBUSY
: *target* could not be unmounted because it is busy.

EFAULT
: *target* points outside the user address space.

EINVAL
: *target* is not a mount point.

EINVAL
: *target* is locked; see *mount_namespaces*(7).

EINVAL
umount2() was called with **MNT_EXPIRE** and either **MNT_DETACH** or **MNT_FORCE**.

EINVAL (since Linux 2.6.34)
umount2() was called with an invalid flag value in *flags*.

ENAMETOOLONG
A pathname was longer than **MAXPATHLEN**.

ENOENT
A pathname was empty or had a nonexistent component.

ENOMEM
The kernel could not allocate a free page to copy filenames or data into.

EPERM
The caller does not have the required privileges.

STANDARDS
Linux.

HISTORY
MNT_DETACH and **MNT_EXPIRE** are available since glibc 2.11.

The original **umount**() function was called as *umount(device)* and would return **ENOTBLK** when called with something other than a block device. In Linux 0.98p4, a call *umount(dir)* was added, in order to support anonymous devices. In Linux 2.3.99-pre7, the call *umount(device)* was removed, leaving only *umount(dir)* (since now devices can be mounted in more than one place, so specifying the device does not suffice).

NOTES
umount() and shared mounts
Shared mounts cause any mount activity on a mount, including **umount**() operations, to be forwarded to every shared mount in the peer group and every slave mount of that peer group. This means that **umount**() of any peer in a set of shared mounts will cause all of its peers to be unmounted and all of their slaves to be unmounted as well.

This propagation of unmount activity can be particularly surprising on systems where every mount is shared by default. On such systems, recursively bind mounting the root directory of the filesystem onto a subdirectory and then later unmounting that subdirectory with **MNT_DETACH** will cause every mount in the mount namespace to be lazily unmounted.

To ensure **umount**() does not propagate in this fashion, the mount may be remounted using a *mount*(2) call with a *mount_flags* argument that includes both **MS_REC** and **MS_PRIVATE** prior to **umount**() being called.

SEE ALSO
mount(2), *mount_namespaces*(7), *path_resolution*(7), *mount*(8), *umount*(8)

NAME

uname − get name and information about current kernel

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/utsname.h>

int uname(struct utsname *****buf* **);

DESCRIPTION

uname() returns system information in the structure pointed to by *buf*. The *utsname* struct is defined in <*sys/utsname.h*>:

```
struct utsname {
    char sysname[];    /* Operating system name (e.g., "Linux") */
    char nodename[];   /* Name within communications network
                          to which the node is attached, if any */
    char release[];    /* Operating system release
                          (e.g., "2.6.28") */
    char version[];    /* Operating system version */
    char machine[];    /* Hardware type identifier */
#ifdef _GNU_SOURCE
    char domainname[]; /* NIS or YP domain name */
#endif
};
```

The length of the arrays in a *struct utsname* is unspecified (see NOTES); the fields are terminated by a null byte ('\0').

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT

buf is not valid.

VERSIONS

The *domainname* member (the NIS or YP domain name) is a GNU extension.

The length of the fields in the struct varies. Some operating systems or libraries use a hardcoded 9 or 33 or 65 or 257. Other systems use **SYS_NMLN** or **_SYS_NMLN** or **UTSLEN** or **_UTSNAME_LENGTH**. Clearly, it is a bad idea to use any of these constants; just use sizeof(...). SVr4 uses 257, "to support Internet hostnames" — this is the largest value likely to be encountered in the wild.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD.

C library/kernel differences

Over time, increases in the size of the *utsname* structure have led to three successive versions of **uname**(): *sys_olduname*() (slot *__NR_oldolduname*), *sys_uname*() (slot *__NR_olduname*), and *sys_newuname*() (slot *__NR_uname)*. The first one used length 9 for all fields; the second used 65; the third also uses 65 but adds the *domainname* field. The glibc **uname**() wrapper function hides these details from applications, invoking the most recent version of the system call provided by the kernel.

NOTES

The kernel has the name, release, version, and supported machine type built in. Conversely, the *nodename* field is configured by the administrator to match the network (this is what the BSD historically calls the "hostname", and is set via *sethostname*(2)). Similarly, the *domainname* field is set via *setdomainname*(2).

Part of the utsname information is also accessible via */proc/sys/kernel/*{*ostype*, *hostname*, *osrelease*, *version*, *domainname*}.

SEE ALSO

uname(1), *getdomainname*(2), *gethostname*(2), *uts_namespaces*(7)

NAME

afs_syscall, break, fattach, fdetach, ftime, getmsg, getpmsg, gtty, isastream, lock, madvise1, mpx, prof, profil, putmsg, putpmsg, security, stty, tuxcall, ulimit, vserver − unimplemented system calls

SYNOPSIS

Unimplemented system calls.

DESCRIPTION

These system calls are not implemented in the Linux kernel.

RETURN VALUE

These system calls always return −1 and set *errno* to **ENOSYS**.

NOTES

Note that *ftime*(3), *profil*(3), and *ulimit*(3) are implemented as library functions.

Some system calls, like *alloc_hugepages*(2), *free_hugepages*(2), *ioperm*(2), *iopl*(2), and *vm86*(2) exist only on certain architectures.

Some system calls, like *ipc*(2), *create_module*(2), *init_module*(2), and *delete_module*(2) exist only when the Linux kernel was built with support for them.

SEE ALSO

syscalls(2)

NAME

unlink, unlinkat − delete a name and possibly the file it refers to

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *pathname);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int unlinkat(int dirfd, const char *pathname, int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

unlinkat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

unlinkat()

The **unlinkat**() system call operates in exactly the same way as either **unlink**() or *rmdir*(2) (depending on whether or not *flags* includes the **AT_REMOVEDIR** flag) except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **unlink**() and *rmdir*(2) for a relative pathname).

If the pathname given in *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **unlink**() and *rmdir*(2)).

If the pathname given in *pathname* is absolute, then *dirfd* is ignored.

flags is a bit mask that can either be specified as 0, or by ORing together flag values that control the operation of **unlinkat**(). Currently, only one such flag is defined:

AT_REMOVEDIR
By default, **unlinkat**() performs the equivalent of **unlink**() on *pathname*. If the **AT_REMOVEDIR** flag is specified, it performs the equivalent of *rmdir*(2) on *pathname*.

See *openat*(2) for an explanation of the need for **unlinkat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Write access to the directory containing *pathname* is not allowed for the process's effective UID, or one of the directories in *pathname* did not allow search permission. (See also *path_resolution*(7).)

EBUSY
The file *pathname* cannot be unlinked because it is being used by the system or another process; for example, it is a mount point or the NFS client software created it to represent an active but otherwise nameless inode ("NFS silly renamed").

EFAULT
pathname points outside your accessible address space.

EIO An I/O error occurred.

EISDIR
pathname refers to a directory. (This is the non-POSIX value returned since Linux 2.1.132.)

ELOOP
Too many symbolic links were encountered in translating *pathname*.

ENAMETOOLONG
pathname was too long.

ENOENT
A component in *pathname* does not exist or is a dangling symbolic link, or *pathname* is empty.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

EPERM
The system does not allow unlinking of directories, or unlinking of directories requires privileges that the calling process doesn't have. (This is the POSIX prescribed error return; as noted above, Linux returns **EISDIR** for this case.)

EPERM (Linux only)
The filesystem does not allow unlinking of files.

EPERM or **EACCES**
The directory containing *pathname* has the sticky bit (**S_ISVTX**) set and the process's effective UID is neither the UID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability).

EPERM
The file to be unlinked is marked immutable or append-only. (See *ioctl_iflags*(2).)

EROFS
pathname refers to a file on a read-only filesystem.

The same errors that occur for **unlink**() and *rmdir*(2) can also occur for **unlinkat**(). The following additional errors can occur for **unlinkat**():

EBADF
pathname is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EINVAL
An invalid flag value was specified in *flags*.

EISDIR
pathname refers to a directory, and **AT_REMOVEDIR** was not specified in *flags*.

ENOTDIR
pathname is relative and *dirfd* is a file descriptor referring to a file other than a directory.

STANDARDS
POSIX.1-2008.

HISTORY
unlink()
SVr4, 4.3BSD, POSIX.1-2001.

unlinkat()
POSIX.1-2008. Linux 2.6.16, glibc 2.4.

glibc
On older kernels where **unlinkat**() is unavailable, the glibc wrapper function falls back to the use of **unlink**() or *rmdir*(2). When *pathname* is a relative pathname, glibc constructs a pathname based on the symbolic link in */proc/self/fd* that corresponds to the *dirfd* argument.

BUGS

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of files which are still being used.

SEE ALSO

rm(1), *unlink*(1), *chmod*(2), *link*(2), *mknod*(2), *open*(2), *rename*(2), *rmdir*(2), *mkfifo*(3), *remove*(3), *path_resolution*(7), *symlink*(7)

NAME

unshare − disassociate parts of the process execution context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE
#include <sched.h>

int unshare(int flags);
```

DESCRIPTION

unshare() allows a process (or thread) to disassociate parts of its execution context that are currently being shared with other processes (or threads). Part of the execution context, such as the mount namespace, is shared implicitly when a new process is created using *fork*(2) or *vfork*(2), while other parts, such as virtual memory, may be shared by explicit request when creating a process or thread using *clone*(2).

The main use of **unshare**() is to allow a process to control its shared execution context without creating a new process.

The *flags* argument is a bit mask that specifies which parts of the execution context should be unshared. This argument is specified by ORing together zero or more of the following constants:

CLONE_FILES
: Reverse the effect of the *clone*(2) **CLONE_FILES** flag. Unshare the file descriptor table, so that the calling process no longer shares its file descriptors with any other process.

CLONE_FS
: Reverse the effect of the *clone*(2) **CLONE_FS** flag. Unshare filesystem attributes, so that the calling process no longer shares its root directory (**chroot**(2)), current directory (**chdir**(2)), or umask (**umask**(2)) attributes with any other process.

CLONE_NEWCGROUP (since Linux 4.6)
: This flag has the same effect as the *clone*(2) **CLONE_NEWCGROUP** flag. Unshare the cgroup namespace. Use of **CLONE_NEWCGROUP** requires the **CAP_SYS_ADMIN** capability.

CLONE_NEWIPC (since Linux 2.6.19)
: This flag has the same effect as the *clone*(2) **CLONE_NEWIPC** flag. Unshare the IPC namespace, so that the calling process has a private copy of the IPC namespace which is not shared with any other process. Specifying this flag automatically implies **CLONE_SYSVSEM** as well. Use of **CLONE_NEWIPC** requires the **CAP_SYS_ADMIN** capability.

CLONE_NEWNET (since Linux 2.6.24)
: This flag has the same effect as the *clone*(2) **CLONE_NEWNET** flag. Unshare the network namespace, so that the calling process is moved into a new network namespace which is not shared with any previously existing process. Use of **CLONE_NEWNET** requires the **CAP_SYS_ADMIN** capability.

CLONE_NEWNS
: This flag has the same effect as the *clone*(2) **CLONE_NEWNS** flag. Unshare the mount namespace, so that the calling process has a private copy of its namespace which is not shared with any other process. Specifying this flag automatically implies **CLONE_FS** as well. Use of **CLONE_NEWNS** requires the **CAP_SYS_ADMIN** capability. For further information, see *mount_namespaces*(7).

CLONE_NEWPID (since Linux 3.8)
: This flag has the same effect as the *clone*(2) **CLONE_NEWPID** flag. Unshare the PID namespace, so that the calling process has a new PID namespace for its children which is not shared with any previously existing process. The calling process is *not* moved into the new namespace. The first child created by the calling process will have the process ID 1 and will assume the role of *init*(1) in the new namespace. **CLONE_NEWPID** automatically implies **CLONE_THREAD** as well. Use of **CLONE_NEWPID** requires the **CAP_SYS_ADMIN** capability. For further information, see *pid_namespaces*(7).

CLONE_NEWTIME (since Linux 5.6)
Unshare the time namespace, so that the calling process has a new time namespace for its children which is not shared with any previously existing process. The calling process is *not* moved into the new namespace. Use of **CLONE_NEWTIME** requires the **CAP_SYS_ADMIN** capability. For further information, see *time_namespaces*(7).

CLONE_NEWUSER (since Linux 3.8)
This flag has the same effect as the *clone*(2) **CLONE_NEWUSER** flag. Unshare the user namespace, so that the calling process is moved into a new user namespace which is not shared with any previously existing process. As with the child process created by *clone*(2) with the **CLONE_NEWUSER** flag, the caller obtains a full set of capabilities in the new namespace.

CLONE_NEWUSER requires that the calling process is not threaded; specifying **CLONE_NEWUSER** automatically implies **CLONE_THREAD**. Since Linux 3.9, **CLONE_NEWUSER** also automatically implies **CLONE_FS**. **CLONE_NEWUSER** requires that the user ID and group ID of the calling process are mapped to user IDs and group IDs in the user namespace of the calling process at the time of the call.

For further information on user namespaces, see *user_namespaces*(7).

CLONE_NEWUTS (since Linux 2.6.19)
This flag has the same effect as the *clone*(2) **CLONE_NEWUTS** flag. Unshare the UTS IPC namespace, so that the calling process has a private copy of the UTS namespace which is not shared with any other process. Use of **CLONE_NEWUTS** requires the **CAP_SYS_ADMIN** capability.

CLONE_SYSVSEM (since Linux 2.6.26)
This flag reverses the effect of the *clone*(2) **CLONE_SYSVSEM** flag. Unshare System V semaphore adjustment (*semadj*) values, so that the calling process has a new empty *semadj* list that is not shared with any other process. If this is the last process that has a reference to the process's current *semadj* list, then the adjustments in that list are applied to the corresponding semaphores, as described in *semop*(2).

In addition, **CLONE_THREAD**, **CLONE_SIGHAND**, and **CLONE_VM** can be specified in *flags* if the caller is single threaded (i.e., it is not sharing its address space with another process or thread). In this case, these flags have no effect. (Note also that specifying **CLONE_THREAD** automatically implies **CLONE_VM**, and specifying **CLONE_VM** automatically implies **CLONE_SIGHAND**.) If the process is multithreaded, then the use of these flags results in an error.

If *flags* is specified as zero, then **unshare**() is a no-op; no changes are made to the calling process's execution context.

RETURN VALUE

On success, zero returned. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
An invalid bit was specified in *flags*.

EINVAL
CLONE_THREAD, **CLONE_SIGHAND**, or **CLONE_VM** was specified in *flags*, and the caller is multithreaded.

EINVAL
CLONE_NEWIPC was specified in *flags*, but the kernel was not configured with the **CONFIG_SYSVIPC** and **CONFIG_IPC_NS** options.

EINVAL
CLONE_NEWNET was specified in *flags*, but the kernel was not configured with the **CONFIG_NET_NS** option.

EINVAL
CLONE_NEWPID was specified in *flags*, but the kernel was not configured with the **CONFIG_PID_NS** option.

EINVAL
CLONE_NEWUSER was specified in *flags*, but the kernel was not configured with the **CONFIG_USER_NS** option.

EINVAL
CLONE_NEWUTS was specified in *flags*, but the kernel was not configured with the **CONFIG_UTS_NS** option.

EINVAL
CLONE_NEWPID was specified in *flags*, but the process has previously called **unshare**() with the **CLONE_NEWPID** flag.

ENOMEM
Cannot allocate sufficient memory to copy parts of caller's context that need to be unshared.

ENOSPC (since Linux 3.7)
CLONE_NEWPID was specified in flags, but the limit on the nesting depth of PID namespaces would have been exceeded; see *pid_namespaces*(7).

ENOSPC (since Linux 4.9; beforehand **EUSERS**)
CLONE_NEWUSER was specified in *flags*, and the call would cause the limit on the number of nested user namespaces to be exceeded. See *user_namespaces*(7).

From Linux 3.11 to Linux 4.8, the error diagnosed in this case was **EUSERS**.

ENOSPC (since Linux 4.9)
One of the values in *flags* specified the creation of a new user namespace, but doing so would have caused the limit defined by the corresponding file in */proc/sys/user* to be exceeded. For further details, see *namespaces*(7).

EPERM
The calling process did not have the required privileges for this operation.

EPERM
CLONE_NEWUSER was specified in *flags*, but either the effective user ID or the effective group ID of the caller does not have a mapping in the parent namespace (see *user_namespaces*(7)).

EPERM (since Linux 3.9)
CLONE_NEWUSER was specified in *flags* and the caller is in a chroot environment (i.e., the caller's root directory does not match the root directory of the mount namespace in which it resides).

EUSERS (from Linux 3.11 to Linux 4.8)
CLONE_NEWUSER was specified in *flags*, and the limit on the number of nested user namespaces would be exceeded. See the discussion of the **ENOSPC** error above.

STANDARDS
Linux.

HISTORY
Linux 2.6.16.

NOTES
Not all of the process attributes that can be shared when a new process is created using *clone*(2) can be unshared using **unshare**(). In particular, as at kernel 3.8, **unshare**() does not implement flags that reverse the effects of **CLONE_SIGHAND**, **CLONE_THREAD**, or **CLONE_VM**. Such functionality may be added in the future, if required.

Creating all kinds of namespace, except user namespaces, requires the **CAP_SYS_ADMIN** capability. However, since creating a user namespace automatically confers a full set of capabilities, creating both a user namespace and any other type of namespace in the same **unshare**() call does not require the **CAP_SYS_ADMIN** capability in the original namespace.

EXAMPLES
The program below provides a simple implementation of the *unshare*(1) command, which unshares one or more namespaces and executes the command supplied in its command-line arguments. Here's an example of the use of this program, running a shell in a new mount namespace, and verifying that the

original shell and the new shell are in separate mount namespaces:

```
$ readlink /proc/$$/ns/mnt
mnt:[4026531840]
$ sudo ./unshare −m /bin/bash
# readlink /proc/$$/ns/mnt
mnt:[4026532325]
```

The differing output of the two *readlink*(1) commands shows that the two shells are in different mount namespaces.

Program source

```
/* unshare.c

   A simple implementation of the unshare(1) command: unshare
   namespaces and execute a command.
*/
#define _GNU_SOURCE
#include <err.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
usage(char *pname)
{
    fprintf(stderr, "Usage: %s [options] program [arg...]\n", pname);
    fprintf(stderr, "Options can be:\n");
    fprintf(stderr, "    −C   unshare cgroup namespace\n");
    fprintf(stderr, "    −i   unshare IPC namespace\n");
    fprintf(stderr, "    −m   unshare mount namespace\n");
    fprintf(stderr, "    −n   unshare network namespace\n");
    fprintf(stderr, "    −p   unshare PID namespace\n");
    fprintf(stderr, "    −t   unshare time namespace\n");
    fprintf(stderr, "    −u   unshare UTS namespace\n");
    fprintf(stderr, "    −U   unshare user namespace\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;

    flags = 0;

    while ((opt = getopt(argc, argv, "CimnptuU")) != −1) {
        switch (opt) {
        case 'C': flags |= CLONE_NEWCGROUP;     break;
        case 'i': flags |= CLONE_NEWIPC;        break;
        case 'm': flags |= CLONE_NEWNS;         break;
        case 'n': flags |= CLONE_NEWNET;        break;
        case 'p': flags |= CLONE_NEWPID;        break;
        case 't': flags |= CLONE_NEWTIME;       break;
        case 'u': flags |= CLONE_NEWUTS;        break;
        case 'U': flags |= CLONE_NEWUSER;       break;
        default:  usage(argv[0]);
        }
```

```
        }

        if (optind >= argc)
            usage(argv[0]);

        if (unshare(flags) == -1)
            err(EXIT_FAILURE, "unshare");

        execvp(argv[optind], &argv[optind]);
        err(EXIT_FAILURE, "execvp");
    }
```

SEE ALSO

unshare(1), *clone*(2), *fork*(2), *kcmp*(2), *setns*(2), *vfork*(2), *namespaces*(7)

Documentation/userspace-api/unshare.rst in the Linux kernel source tree (or *Documentation/unshare.txt* before Linux 4.12)

NAME

uselib − load shared library

SYNOPSIS

#include <unistd.h>

[[deprecated]] int uselib(const char ****library*);**

DESCRIPTION

The system call **uselib**() serves to load a shared library to be used by the calling process. It is given a pathname. The address where to load is found in the library itself. The library can have any recognized binary format.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

In addition to all of the error codes returned by *open*(2) and *mmap*(2), the following may also be returned:

EACCES
: The library specified by *library* does not have read or execute permission, or the caller does not have search permission for one of the directories in the path prefix. (See also *path_resolution*(7).)

ENFILE
: The system-wide limit on the total number of open files has been reached.

ENOEXEC
: The file specified by *library* is not an executable of a known type; for example, it does not have the correct magic numbers.

STANDARDS

Linux.

HISTORY

This obsolete system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc before glibc 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it was sufficient to manually declare the interface in your code; alternatively, you could invoke the system call using *syscall*(2).

In ancient libc versions (before glibc 2.0), **uselib**() was used to load the shared libraries with names found in an array of names in the binary.

Since Linux 3.15, this system call is available only when the kernel is configured with the **CONFIG_USELIB** option.

SEE ALSO

ar(1), *gcc*(1), *ld*(1), *ldd*(1), *mmap*(2), *open*(2), *dlopen*(3), *capabilities*(7), *ld.so*(8)

NAME

userfaultfd − create a file descriptor for handling page faults in user space

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>            /* Definition of O_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <linux/userfaultfd.h> /* Definition of UFFD_* constants */
#include <unistd.h>

int syscall(SYS_userfaultfd, int flags);
```

Note: glibc provides no wrapper for **userfaultfd**(), necessitating the use of *syscall*(2).

DESCRIPTION

userfaultfd() creates a new userfaultfd object that can be used for delegation of page-fault handling to a user-space application, and returns a file descriptor that refers to the new object. The new userfaultfd object is configured using *ioctl*(2).

Once the userfaultfd object is configured, the application can use *read*(2) to receive userfaultfd notifications. The reads from userfaultfd may be blocking or non-blocking, depending on the value of *flags* used for the creation of the userfaultfd or subsequent calls to *fcntl*(2).

The following values may be bitwise ORed in *flags* to change the behavior of **userfaultfd**():

O_CLOEXEC
: Enable the close-on-exec flag for the new userfaultfd file descriptor. See the description of the **O_CLOEXEC** flag in *open*(2).

O_NONBLOCK
: Enables non-blocking operation for the userfaultfd object. See the description of the **O_NONBLOCK** flag in *open*(2).

UFFD_USER_MODE_ONLY
: This is an userfaultfd-specific flag that was introduced in Linux 5.11. When set, the userfaultfd object will only be able to handle page faults originated from the user space on the registered regions. When a kernel-originated fault was triggered on the registered range with this userfaultfd, a **SIGBUS** signal will be delivered.

When the last file descriptor referring to a userfaultfd object is closed, all memory ranges that were registered with the object are unregistered and unread events are flushed.

Userfaultfd supports three modes of registration:

UFFDIO_REGISTER_MODE_MISSING (since Linux 4.10)
: When registered with **UFFDIO_REGISTER_MODE_MISSING** mode, user-space will receive a page-fault notification when a missing page is accessed. The faulted thread will be stopped from execution until the page fault is resolved from user-space by either an **UFFDIO_COPY** or an **UFFDIO_ZEROPAGE** ioctl.

UFFDIO_REGISTER_MODE_MINOR (since Linux 5.13)
: When registered with **UFFDIO_REGISTER_MODE_MINOR** mode, user-space will receive a page-fault notification when a minor page fault occurs. That is, when a backing page is in the page cache, but page table entries don't yet exist. The faulted thread will be stopped from execution until the page fault is resolved from user-space by an **UFFDIO_CONTINUE** ioctl.

UFFDIO_REGISTER_MODE_WP (since Linux 5.7)
: When registered with **UFFDIO_REGISTER_MODE_WP** mode, user-space will receive a page-fault notification when a write-protected page is written. The faulted thread will be stopped from execution until user-space write-unprotects the page using an **UFFDIO_WRITEPROTECT** ioctl.

Multiple modes can be enabled at the same time for the same memory range.

Since Linux 4.14, a userfaultfd page-fault notification can selectively embed faulting thread ID information into the notification. One needs to enable this feature explicitly using the

UFFD_FEATURE_THREAD_ID feature bit when initializing the userfaultfd context. By default, thread ID reporting is disabled.

Usage

The userfaultfd mechanism is designed to allow a thread in a multithreaded program to perform user-space paging for the other threads in the process. When a page fault occurs for one of the regions registered to the userfaultfd object, the faulting thread is put to sleep and an event is generated that can be read via the userfaultfd file descriptor. The fault-handling thread reads events from this file descriptor and services them using the operations described in *ioctl_userfaultfd*(2). When servicing the page fault events, the fault-handling thread can trigger a wake-up for the sleeping thread.

It is possible for the faulting threads and the fault-handling threads to run in the context of different processes. In this case, these threads may belong to different programs, and the program that executes the faulting threads will not necessarily cooperate with the program that handles the page faults. In such non-cooperative mode, the process that monitors userfaultfd and handles page faults needs to be aware of the changes in the virtual memory layout of the faulting process to avoid memory corruption.

Since Linux 4.11, userfaultfd can also notify the fault-handling threads about changes in the virtual memory layout of the faulting process. In addition, if the faulting process invokes *fork*(2), the userfaultfd objects associated with the parent may be duplicated into the child process and the userfaultfd monitor will be notified (via the **UFFD_EVENT_FORK** described below) about the file descriptor associated with the userfault objects created for the child process, which allows the userfaultfd monitor to perform user-space paging for the child process. Unlike page faults which have to be synchronous and require an explicit or implicit wakeup, all other events are delivered asynchronously and the non-cooperative process resumes execution as soon as the userfaultfd manager executes *read*(2). The userfaultfd manager should carefully synchronize calls to **UFFDIO_COPY** with the processing of events.

The current asynchronous model of the event delivery is optimal for single threaded non-cooperative userfaultfd manager implementations.

Since Linux 5.7, userfaultfd is able to do synchronous page dirty tracking using the new write-protect register mode. One should check against the feature bit **UFFD_FEATURE_PAGEFAULT_FLAG_WP** before using this feature. Similar to the original userfaultfd missing mode, the write-protect mode will generate a userfaultfd notification when the protected page is written. The user needs to resolve the page fault by unprotecting the faulted page and kicking the faulted thread to continue. For more information, please refer to the "Userfaultfd write-protect mode" section.

Userfaultfd operation

After the userfaultfd object is created with **userfaultfd**(), the application must enable it using the **UFFDIO_API** *ioctl*(2) operation. This operation allows a two-step handshake between the kernel and user space to determine what API version and features the kernel supports, and then to enable those features user space wants. This operation must be performed before any of the other *ioctl*(2) operations described below (or those operations fail with the **EINVAL** error).

After a successful **UFFDIO_API** operation, the application then registers memory address ranges using the **UFFDIO_REGISTER** *ioctl*(2) operation. After successful completion of a **UFFDIO_REGISTER** operation, a page fault occurring in the requested memory range, and satisfying the mode defined at the registration time, will be forwarded by the kernel to the user-space application. The application can then use various (e.g., **UFFDIO_COPY**, **UFFDIO_ZEROPAGE**, or **UFFDIO_CONTINUE**) *ioctl*(2) operations to resolve the page fault.

Since Linux 4.14, if the application sets the **UFFD_FEATURE_SIGBUS** feature bit using the **UFFDIO_API** *ioctl*(2), no page-fault notification will be forwarded to user space. Instead a **SIGBUS** signal is delivered to the faulting process. With this feature, userfaultfd can be used for robustness purposes to simply catch any access to areas within the registered address range that do not have pages allocated, without having to listen to userfaultfd events. No userfaultfd monitor will be required for dealing with such memory accesses. For example, this feature can be useful for applications that want to prevent the kernel from automatically allocating pages and filling holes in sparse files when the hole is accessed through a memory mapping.

The **UFFD_FEATURE_SIGBUS** feature is implicitly inherited through *fork*(2) if used in combination with **UFFD_FEATURE_FORK**.

Details of the various *ioctl*(2) operations can be found in *ioctl_userfaultfd*(2).

Since Linux 4.11, events other than page-fault may enabled during **UFFDIO_API** operation.

Up to Linux 4.11, userfaultfd can be used only with anonymous private memory mappings. Since Linux 4.11, userfaultfd can be also used with hugetlbfs and shared memory mappings.

Userfaultfd write-protect mode (since Linux 5.7)

Since Linux 5.7, userfaultfd supports write-protect mode for anonymous memory. The user needs to first check availability of this feature using **UFFDIO_API** ioctl against the feature bit **UFFD_FEATURE_PAGEFAULT_FLAG_WP** before using this feature.

Since Linux 5.19, the write-protection mode was also supported on shmem and hugetlbfs memory types. It can be detected with the feature bit **UFFD_FEATURE_WP_HUGETLBFS_SHMEM**.

To register with userfaultfd write-protect mode, the user needs to initiate the **UFFDIO_REGISTER** ioctl with mode **UFFDIO_REGISTER_MODE_WP** set. Note that it is legal to monitor the same memory range with multiple modes. For example, the user can do **UFFDIO_REGISTER** with the mode set to **UFFDIO_REGISTER_MODE_MISSING | UFFDIO_REGISTER_MODE_WP**. When there is only **UFFDIO_REGISTER_MODE_WP** registered, user-space will *not* receive any notification when a missing page is written. Instead, user-space will receive a write-protect page-fault notification only when an existing but write-protected page got written.

After the **UFFDIO_REGISTER** ioctl completed with **UFFDIO_REGISTER_MODE_WP** mode set, the user can write-protect any existing memory within the range using the ioctl **UFFDIO_WRITEPROTECT** where *uffdio_writeprotect.mode* should be set to **UFFDIO_WRITEPROTECT_MODE_WP**.

When a write-protect event happens, user-space will receive a page-fault notification whose *uffd_msg.pagefault.flags* will be with **UFFD_PAGEFAULT_FLAG_WP** flag set. Note: since only writes can trigger this kind of fault, write-protect notifications will always have the **UFFD_PAGEFAULT_FLAG_WRITE** bit set along with the **UFFD_PAGEFAULT_FLAG_WP** bit.

To resolve a write-protection page fault, the user should initiate another **UFFDIO_WRITEPROTECT** ioctl, whose *uffd_msg.pagefault.flags* should have the flag **UFFDIO_WRITEPROTECT_MODE_WP** cleared upon the faulted page or range.

Userfaultfd minor fault mode (since Linux 5.13)

Since Linux 5.13, userfaultfd supports minor fault mode. In this mode, fault messages are produced not for major faults (where the page was missing), but rather for minor faults, where a page exists in the page cache, but the page table entries are not yet present. The user needs to first check availability of this feature using the **UFFDIO_API** ioctl with the appropriate feature bits set before using this feature: **UFFD_FEATURE_MINOR_HUGETLBFS** since Linux 5.13, or **UFFD_FEATURE_MINOR_SHMEM** since Linux 5.14.

To register with userfaultfd minor fault mode, the user needs to initiate the **UFFDIO_REGISTER** ioctl with mode **UFFD_REGISTER_MODE_MINOR** set.

When a minor fault occurs, user-space will receive a page-fault notification whose *uffd_msg.pagefault.flags* will have the **UFFD_PAGEFAULT_FLAG_MINOR** flag set.

To resolve a minor page fault, the handler should decide whether or not the existing page contents need to be modified first. If so, this should be done in-place via a second, non-userfaultfd-registered mapping to the same backing page (e.g., by mapping the shmem or hugetlbfs file twice). Once the page is considered "up to date", the fault can be resolved by initiating an **UFFDIO_CONTINUE** ioctl, which installs the page table entries and (by default) wakes up the faulting thread(s).

Minor fault mode supports only hugetlbfs-backed (since Linux 5.13) and shmem-backed (since Linux 5.14) memory.

Reading from the userfaultfd structure

Each *read*(2) from the userfaultfd file descriptor returns one or more *uffd_msg* structures, each of which describes a page-fault event or an event required for the non-cooperative userfaultfd usage:

```
struct uffd_msg {
    __u8  event;            /* Type of event */
    ...
    union {
        struct {
```

```
                __u64 flags;    /* Flags describing fault */
                __u64 address;  /* Faulting address */
                union {
                    __u32 ptid; /* Thread ID of the fault */
                } feat;
            } pagefault;

            struct {            /* Since Linux 4.11 */
                __u32 ufd;      /* Userfault file descriptor
                                   of the child process */
            } fork;

            struct {            /* Since Linux 4.11 */
                __u64 from;     /* Old address of remapped area */
                __u64 to;       /* New address of remapped area */
                __u64 len;      /* Original mapping length */
            } remap;

            struct {            /* Since Linux 4.11 */
                __u64 start;    /* Start address of removed area */
                __u64 end;      /* End address of removed area */
            } remove;
            ...
        } arg;

        /* Padding fields omitted */
    } __packed;
```

If multiple events are available and the supplied buffer is large enough, *read*(2) returns as many events as will fit in the supplied buffer. If the buffer supplied to *read*(2) is smaller than the size of the *uffd_msg* structure, the *read*(2) fails with the error **EINVAL**.

The fields set in the *uffd_msg* structure are as follows:

event The type of event. Depending of the event type, different fields of the *arg* union represent details required for the event processing. The non-page-fault events are generated only when appropriate feature is enabled during API handshake with **UFFDIO_API** *ioctl*(2).

The following values can appear in the *event* field:

UFFD_EVENT_PAGEFAULT (since Linux 4.3)
A page-fault event. The page-fault details are available in the *pagefault* field.

UFFD_EVENT_FORK (since Linux 4.11)
Generated when the faulting process invokes *fork*(2) (or *clone*(2) without the **CLONE_VM** flag). The event details are available in the *fork* field.

UFFD_EVENT_REMAP (since Linux 4.11)
Generated when the faulting process invokes *mremap*(2). The event details are available in the *remap* field.

UFFD_EVENT_REMOVE (since Linux 4.11)
Generated when the faulting process invokes *madvise*(2) with **MADV_DONTNEED** or **MADV_REMOVE** advice. The event details are available in the *remove* field.

UFFD_EVENT_UNMAP (since Linux 4.11)
Generated when the faulting process unmaps a memory range, either explicitly using *munmap*(2) or implicitly during *mmap*(2) or *mremap*(2). The event details are available in the *remove* field.

pagefault.address
The address that triggered the page fault.

pagefault.flags
A bit mask of flags that describe the event. For **UFFD_EVENT_PAGEFAULT**, the following flag may appear:

UFFD_PAGEFAULT_FLAG_WP
If this flag is set, then the fault was a write-protect fault.

UFFD_PAGEFAULT_FLAG_MINOR
If this flag is set, then the fault was a minor fault.

UFFD_PAGEFAULT_FLAG_WRITE
If this flag is set, then the fault was a write fault.

If neither **UFFD_PAGEFAULT_FLAG_WP** nor **UFFD_PAGEFAULT_FLAG_MINOR** are set, then the fault was a missing fault.

pagefault.feat.pid
The thread ID that triggered the page fault.

fork.ufd
The file descriptor associated with the userfault object created for the child created by *fork*(2).

remap.from
The original address of the memory range that was remapped using *mremap*(2).

remap.to
The new address of the memory range that was remapped using *mremap*(2).

remap.len
The original length of the memory range that was remapped using *mremap*(2).

remove.start
The start address of the memory range that was freed using *madvise*(2) or unmapped

remove.end
The end address of the memory range that was freed using *madvise*(2) or unmapped

A *read*(2) on a userfaultfd file descriptor can fail with the following errors:

EINVAL
The userfaultfd object has not yet been enabled using the **UFFDIO_API** *ioctl*(2) operation

If the **O_NONBLOCK** flag is enabled in the associated open file description, the userfaultfd file descriptor can be monitored with *poll*(2), *select*(2), and *epoll*(7). When events are available, the file descriptor indicates as readable. If the **O_NONBLOCK** flag is not enabled, then *poll*(2) (always) indicates the file as having a **POLLERR** condition, and *select*(2) indicates the file descriptor as both readable and writable.

RETURN VALUE

On success, **userfaultfd**() returns a new file descriptor that refers to the userfaultfd object. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
An unsupported value was specified in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
Insufficient kernel memory was available.

EPERM (since Linux 5.2)
The caller is not privileged (does not have the **CAP_SYS_PTRACE** capability in the initial user namespace), and */proc/sys/vm/unprivileged_userfaultfd* has the value 0.

STANDARDS

Linux.

HISTORY

Linux 4.3.

Support for hugetlbfs and shared memory areas and non-page-fault events was added in Linux 4.11

NOTES

The userfaultfd mechanism can be used as an alternative to traditional user-space paging techniques based on the use of the **SIGSEGV** signal and *mmap*(2). It can also be used to implement lazy restore for checkpoint/restore mechanisms, as well as post-copy migration to allow (nearly) uninterrupted execution when transferring virtual machines and Linux containers from one host to another.

BUGS

If the **UFFD_FEATURE_EVENT_FORK** is enabled and a system call from the *fork*(2) family is interrupted by a signal or failed, a stale userfaultfd descriptor might be created. In this case, a spurious **UFFD_EVENT_FORK** will be delivered to the userfaultfd monitor.

EXAMPLES

The program below demonstrates the use of the userfaultfd mechanism. The program creates two threads, one of which acts as the page-fault handler for the process, for the pages in a demand-page zero region created using *mmap*(2).

The program takes one command-line argument, which is the number of pages that will be created in a mapping whose page faults will be handled via userfaultfd. After creating a userfaultfd object, the program then creates an anonymous private mapping of the specified size and registers the address range of that mapping using the **UFFDIO_REGISTER** *ioctl*(2) operation. The program then creates a second thread that will perform the task of handling page faults.

The main thread then walks through the pages of the mapping fetching bytes from successive pages. Because the pages have not yet been accessed, the first access of a byte in each page will trigger a page-fault event on the userfaultfd file descriptor.

Each of the page-fault events is handled by the second thread, which sits in a loop processing input from the userfaultfd file descriptor. In each loop iteration, the second thread first calls *poll*(2) to check the state of the file descriptor, and then reads an event from the file descriptor. All such events should be **UFFD_EVENT_PAGEFAULT** events, which the thread handles by copying a page of data into the faulting region using the **UFFDIO_COPY** *ioctl*(2) operation.

The following is an example of what we see when running the program:

```
$ ./userfaultfd_demo 3
Address returned by mmap() = 0x7fd30106c000

fault_handler_thread():
    poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
    UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106c00f
        (uffdio_copy.copy returned 4096)
Read address 0x7fd30106c00f in main(): A
Read address 0x7fd30106c40f in main(): A
Read address 0x7fd30106c80f in main(): A
Read address 0x7fd30106cc0f in main(): A

fault_handler_thread():
    poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
    UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106d00f
        (uffdio_copy.copy returned 4096)
Read address 0x7fd30106d00f in main(): B
Read address 0x7fd30106d40f in main(): B
Read address 0x7fd30106d80f in main(): B
Read address 0x7fd30106dc0f in main(): B

fault_handler_thread():
    poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
```

```
        UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106e00f
            (uffdio_copy.copy returned 4096)
    Read address 0x7fd30106e00f in main(): C
    Read address 0x7fd30106e40f in main(): C
    Read address 0x7fd30106e80f in main(): C
    Read address 0x7fd30106ec0f in main(): C
```

Program source

```
/* userfaultfd_demo.c

   Licensed under the GNU General Public License version 2 or later.
*/
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <linux/userfaultfd.h>
#include <poll.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <unistd.h>

static int page_size;

static void *
fault_handler_thread(void *arg)
{
    int                 nready;
    long                uffd;   /* userfaultfd file descriptor */
    ssize_t             nread;
    struct pollfd       pollfd;
    struct uffdio_copy  uffdio_copy;

    static int      fault_cnt = 0; /* Number of faults so far handled */
    static char     *page = NULL;
    static struct uffd_msg  msg;  /* Data read from userfaultfd */

    uffd = (long) arg;

    /* Create a page that will be copied into the faulting region. */

    if (page == NULL) {
        page = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (page == MAP_FAILED)
            err(EXIT_FAILURE, "mmap");
    }

    /* Loop, handling incoming events on the userfaultfd
       file descriptor. */

    for (;;) {
```

```
            /* See what poll() tells us about the userfaultfd. */

            pollfd.fd = uffd;
            pollfd.events = POLLIN;
            nready = poll(&pollfd, 1, -1);
            if (nready == -1)
                err(EXIT_FAILURE, "poll");

            printf("\nfault_handler_thread():\n");
            printf("    poll() returns: nready = %d; "
                   "POLLIN = %d; POLLERR = %d\n", nready,
                   (pollfd.revents & POLLIN) != 0,
                   (pollfd.revents & POLLERR) != 0);

            /* Read an event from the userfaultfd. */

            nread = read(uffd, &msg, sizeof(msg));
            if (nread == 0) {
                printf("EOF on userfaultfd!\n");
                exit(EXIT_FAILURE);
            }

            if (nread == -1)
                err(EXIT_FAILURE, "read");

            /* We expect only one kind of event; verify that assumption. */

            if (msg.event != UFFD_EVENT_PAGEFAULT) {
                fprintf(stderr, "Unexpected event on userfaultfd\n");
                exit(EXIT_FAILURE);
            }

            /* Display info about the page-fault event. */

            printf("    UFFD_EVENT_PAGEFAULT event: ");
            printf("flags = %"PRIx64"; ", msg.arg.pagefault.flags);
            printf("address = %"PRIx64"\n", msg.arg.pagefault.address);

            /* Copy the page pointed to by 'page' into the faulting
               region. Vary the contents that are copied in, so that it
               is more obvious that each fault is handled separately. */

            memset(page, 'A' + fault_cnt % 20, page_size);
            fault_cnt++;

            uffdio_copy.src = (unsigned long) page;

            /* We need to handle page faults in units of pages(!).
               So, round faulting address down to page boundary. */

            uffdio_copy.dst = (unsigned long) msg.arg.pagefault.address &
                                              ~(page_size - 1);
            uffdio_copy.len = page_size;
            uffdio_copy.mode = 0;
            uffdio_copy.copy = 0;
            if (ioctl(uffd, UFFDIO_COPY, &uffdio_copy) == -1)
                err(EXIT_FAILURE, "ioctl-UFFDIO_COPY");
```

```
            printf("        (uffdio_copy.copy returned %"PRId64")\n",
                   uffdio_copy.copy);
        }
    }

    int
    main(int argc, char *argv[])
    {
        int        s;
        char       c;
        char       *addr;   /* Start of region handled by userfaultfd */
        long       uffd;    /* userfaultfd file descriptor */
        size_t     len, l;  /* Length of region handled by userfaultfd */
        pthread_t  thr;     /* ID of thread that handles page faults */
        struct uffdio_api       uffdio_api;
        struct uffdio_register  uffdio_register;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s num-pages\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        page_size = sysconf(_SC_PAGE_SIZE);
        len = strtoull(argv[1], NULL, 0) * page_size;

        /* Create and enable userfaultfd object. */

        uffd = syscall(SYS_userfaultfd, O_CLOEXEC | O_NONBLOCK);
        if (uffd == -1)
            err(EXIT_FAILURE, "userfaultfd");

        /* NOTE: Two-step feature handshake is not needed here, since this
           example doesn't require any specific features.

           Programs that *do* should call UFFDIO_API twice: once with
           'features = 0' to detect features supported by this kernel, and
           again with the subset of features the program actually wants to
           enable. */
        uffdio_api.api = UFFD_API;
        uffdio_api.features = 0;
        if (ioctl(uffd, UFFDIO_API, &uffdio_api) == -1)
            err(EXIT_FAILURE, "ioctl-UFFDIO_API");

        /* Create a private anonymous mapping. The memory will be
           demand-zero paged--that is, not yet allocated. When we
           actually touch the memory, it will be allocated via
           the userfaultfd. */

        addr = mmap(NULL, len, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (addr == MAP_FAILED)
            err(EXIT_FAILURE, "mmap");

        printf("Address returned by mmap() = %p\n", addr);

        /* Register the memory range of the mapping we just created for
           handling by the userfaultfd object. In mode, we request to track
           missing pages (i.e., pages that have not yet been faulted in). */
```

```
        uffdio_register.range.start = (unsigned long) addr;
        uffdio_register.range.len = len;
        uffdio_register.mode = UFFDIO_REGISTER_MODE_MISSING;
        if (ioctl(uffd, UFFDIO_REGISTER, &uffdio_register) == -1)
            err(EXIT_FAILURE, "ioctl-UFFDIO_REGISTER");

        /* Create a thread that will process the userfaultfd events. */

        s = pthread_create(&thr, NULL, fault_handler_thread, (void *) uffd);
        if (s != 0) {
            errc(EXIT_FAILURE, s, "pthread_create");
        }

        /* Main thread now touches memory in the mapping, touching
           locations 1024 bytes apart. This will trigger userfaultfd
           events for all pages in the region. */

        l = 0xf;    /* Ensure that faulting address is not on a page
                       boundary, in order to test that we correctly
                       handle that case in fault_handling_thread(). */
        while (l < len) {
            c = addr[l];
            printf("Read address %p in %s(): ", addr + l, __func__);
            printf("%c\n", c);
            l += 1024;
            usleep(100000);         /* Slow things down a little */
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fcntl(2), *ioctl*(2), *ioctl_userfaultfd*(2), *madvise*(2), *mmap*(2)

Documentation/admin-guide/mm/userfaultfd.rst in the Linux kernel source tree

NAME

ustat − get filesystem statistics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>    /* libc[45] */
#include <ustat.h>     /* glibc2 */

[[deprecated]] int ustat(dev_t dev, struct ustat *ubuf);
```

DESCRIPTION

ustat() returns information about a mounted filesystem. *dev* is a device number identifying a device containing a mounted filesystem. *ubuf* is a pointer to a *ustat* structure that contains the following members:

```
daddr_t f_tfree;      /* Total free blocks */
ino_t   f_tinode;     /* Number of free inodes */
char    f_fname[6];   /* Filsys name */
char    f_fpack[6];   /* Filsys pack name */
```

The last two fields, *f_fname* and *f_fpack*, are not implemented and will always be filled with null bytes ('\0').

RETURN VALUE

On success, zero is returned and the *ustat* structure pointed to by *ubuf* will be filled in. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: *ubuf* points outside of your accessible address space.

EINVAL
: *dev* does not refer to a device containing a mounted filesystem.

ENOSYS
: The mounted filesystem referenced by *dev* does not support this operation, or any version of Linux before Linux 1.3.16.

STANDARDS

None.

HISTORY

SVr4. Removed in glibc 2.28.

ustat() is deprecated and has been provided only for compatibility. All new programs should use *statfs*(2) instead.

HP-UX notes

The HP-UX version of the *ustat* structure has an additional field, *f_blksize*, that is unknown elsewhere. HP-UX warns: For some filesystems, the number of free inodes does not change. Such filesystems will return −1 in the field *f_tinode*. For some filesystems, inodes are dynamically allocated. Such filesystems will return the current number of free inodes.

SEE ALSO

stat(2), *statfs*(2)

NAME

utime, utimes − change file last access and modification times

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <utime.h>

int utime(const char *filename,
          const struct utimbuf *_Nullable times);

#include <sys/time.h>

int utimes(const char *filename,
           const struct timeval times[_Nullable 2]);
```

DESCRIPTION

Note: modern applications may prefer to use the interfaces described in *utimensat*(2).

The **utime**() system call changes the access and modification times of the inode specified by *filename* to the *actime* and *modtime* fields of *times* respectively. The status change time (ctime) will be set to the current time, even if the other time stamps don't actually change.

If *times* is NULL, then the access and modification times of the file are set to the current time.

Changing timestamps is permitted when: either the process has appropriate privileges, or the effective user ID equals the user ID of the file, or *times* is NULL and the process has write permission for the file.

The *utimbuf* structure is:

```
struct utimbuf {
    time_t actime;       /* access time */
    time_t modtime;      /* modification time */
};
```

The **utime**() system call allows specification of timestamps with a resolution of 1 second.

The **utimes**() system call is similar, but the *times* argument refers to an array rather than a structure. The elements of this array are *timeval* structures, which allow a precision of 1 microsecond for specifying timestamps. The *timeval* structure is:

```
struct timeval {
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
};
```

times[0] specifies the new access time, and *times[1]* specifies the new modification time. If *times* is NULL, then analogously to **utime**(), the access and modification times of the file are set to the current time.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: Search permission is denied for one of the directories in the path prefix of *path* (see also *path_resolution*(7)).

EACCES
: *times* is NULL, the caller's effective user ID does not match the owner of the file, the caller does not have write access to the file, and the caller is not privileged (Linux: does not have either the **CAP_DAC_OVERRIDE** or the **CAP_FOWNER** capability).

ENOENT
: *filename* does not exist.

EPERM
: *times* is not NULL, the caller's effective UID does not match the owner of the file, and the caller is not privileged (Linux: does not have the **CAP_FOWNER** capability).

EROFS
path resides on a read-only filesystem.

STANDARDS
POSIX.1-2008.

HISTORY
utime()
SVr4, POSIX.1-2001. POSIX.1-2008 marks it as obsolete.

utimes()
4.3BSD, POSIX.1-2001.

NOTES
Linux does not allow changing the timestamps on an immutable file, or setting the timestamps to something other than the current time on an append-only file.

SEE ALSO
chattr(1), *touch*(1), *futimesat*(2), *stat*(2), *utimensat*(2), *futimens*(3), *futimes*(3), *inode*(7)

NAME

utimensat, futimens − change file timestamps with nanosecond precision

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>          /* Definition of AT_* constants */
#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname,
              const struct timespec times[_Nullable 2], int flags);
int futimens(int fd, const struct timespec times[_Nullable 2]);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

utimensat():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _ATFILE_SOURCE

futimens():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _GNU_SOURCE

DESCRIPTION

utimensat() and **futimens**() update the timestamps of a file with nanosecond precision. This contrasts with the historical *utime*(2) and *utimes*(2), which permit only second and microsecond precision, respectively, when setting file timestamps.

With **utimensat**() the file is specified via the pathname given in *pathname*. With **futimens**() the file whose timestamps are to be updated is specified via an open file descriptor, *fd*.

For both calls, the new file timestamps are specified in the array *times*: *times[0]* specifies the new "last access time" (*atime*); *times[1]* specifies the new "last modification time" (*mtime*). Each of the elements of *times* specifies a time as the number of seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). This information is conveyed in a *timespec*(3) structure.

Updated file timestamps are set to the greatest value supported by the filesystem that is not greater than the specified time.

If the *tv_nsec* field of one of the *timespec* structures has the special value **UTIME_NOW**, then the corresponding file timestamp is set to the current time. If the *tv_nsec* field of one of the *timespec* structures has the special value **UTIME_OMIT**, then the corresponding file timestamp is left unchanged. In both of these cases, the value of the corresponding *tv_sec* field is ignored.

If *times* is NULL, then both timestamps are set to the current time.

The status change time (ctime) will be set to the current time, even if the other time stamps don't actually change.

Permissions requirements

To set both file timestamps to the current time (i.e., *times* is NULL, or both *tv_nsec* fields specify **UTIME_NOW**), either:

- the caller must have write access to the file;
- the caller's effective user ID must match the owner of the file; or
- the caller must have appropriate privileges.

To make any change other than setting both timestamps to the current time (i.e., *times* is not NULL, and neither *tv_nsec* field is **UTIME_NOW** and neither *tv_nsec* field is **UTIME_OMIT**), either condition 2 or 3 above must apply.

If both *tv_nsec* fields are specified as **UTIME_OMIT**, then no file ownership or permission checks are performed, and the file timestamps are not modified, but other error conditions may still be detected.

utimensat() specifics

If *pathname* is relative, then by default it is interpreted relative to the directory referred to by the open file descriptor, *dirfd* (rather than relative to the current working directory of the calling process, as is done by *utimes*(2) for a relative pathname). See *openat*(2) for an explanation of why this can be useful.

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *utimes*(2)).

If *pathname* is absolute, then *dirfd* is ignored.

The *flags* argument is a bit mask created by ORing together zero or more of the following values defined in <*fcntl.h*>:

AT_EMPTY_PATH (since Linux 5.8)
: If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the *open*(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory. If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_NOFOLLOW
: If *pathname* specifies a symbolic link, then update the timestamps of the link, rather than the file to which it refers.

RETURN VALUE

On success, **utimensat**() and **futimens**() return 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
: *times* is NULL, or both *tv_nsec* values are **UTIME_NOW**, and the effective user ID of the caller does not match the owner of the file, the caller does not have write access to the file, and the caller is not privileged (Linux: does not have either the **CAP_FOWNER** or the **CAP_DAC_OVERRIDE** capability).

EBADF
: (**futimens**()) *fd* is not a valid file descriptor.

EBADF
: (**utimensat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
: *times* pointed to an invalid address; or, *dirfd* was **AT_FDCWD**, and *pathname* is NULL or an invalid address.

EINVAL
: Invalid value in *flags*.

EINVAL
: Invalid value in one of the *tv_nsec* fields (value outside range [0, 999,999,999], and not **UTIME_NOW** or **UTIME_OMIT**); or an invalid value in one of the *tv_sec* fields.

EINVAL
: *pathname* is NULL, *dirfd* is not **AT_FDCWD**, and *flags* contains **AT_SYMLINK_NOFOLLOW**.

ELOOP
: (**utimensat**()) Too many symbolic links were encountered in resolving *pathname*.

ENAMETOOLONG
: (**utimensat**()) *pathname* is too long.

ENOENT
: (**utimensat**()) A component of *pathname* does not refer to an existing directory or file, or *pathname* is an empty string.

ENOTDIR
: (**utimensat**()) *pathname* is a relative pathname, but *dirfd* is neither **AT_FDCWD** nor a file descriptor referring to a directory; or, one of the prefix components of *pathname* is not a

directory.

EPERM
The caller attempted to change one or both timestamps to a value other than the current time, or to change one of the timestamps to the current time while leaving the other timestamp unchanged, (i.e., *times* is not NULL, neither *tv_nsec* field is **UTIME_NOW**, and neither *tv_nsec* field is **UTIME_OMIT**) and either:

- the caller's effective user ID does not match the owner of file, and the caller is not privileged (Linux: does not have the **CAP_FOWNER** capability); or,
- the file is marked append-only or immutable (see *chattr*(1)).

EROFS
The file is on a read-only filesystem.

ESRCH
(**utimensat**()) Search permission is denied for one of the prefix components of *pathname*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
utimensat(), **futimens**()	Thread safety	MT-Safe

VERSIONS

C library/kernel ABI differences

On Linux, **futimens**() is a library function implemented on top of the **utimensat**() system call. To support this, the Linux **utimensat**() system call implements a nonstandard feature: if *pathname* is NULL, then the call modifies the timestamps of the file referred to by the file descriptor *dirfd* (which may refer to any type of file). Using this feature, the call *futimens(fd, times)* is implemented as:

```
utimensat(fd, NULL, times, 0);
```

Note, however, that the glibc wrapper for **utimensat**() disallows passing NULL as the value for *pathname*: the wrapper function returns the error **EINVAL** in this case.

STANDARDS

POSIX.1-2008.

VERSIONS

utimensat()
Linux 2.6.22, glibc 2.6. POSIX.1-2008.

futimens()
glibc 2.6. POSIX.1-2008.

NOTES

utimensat() obsoletes *futimesat*(2).

On Linux, timestamps cannot be changed for a file marked immutable, and the only change permitted for files marked append-only is to set the timestamps to the current time. (This is consistent with the historical behavior of *utime*(2) and *utimes*(2) on Linux.)

If both *tv_nsec* fields are specified as **UTIME_OMIT**, then the Linux implementation of **utimensat**() succeeds even if the file referred to by *dirfd* and *pathname* does not exist.

BUGS

Several bugs afflict **utimensat**() and **futimens**() before Linux 2.6.26. These bugs are either nonconformances with the POSIX.1 draft specification or inconsistencies with historical Linux behavior.

- POSIX.1 specifies that if one of the *tv_nsec* fields has the value **UTIME_NOW** or **UTIME_OMIT**, then the value of the corresponding *tv_sec* field should be ignored. Instead, the value of the *tv_sec* field is required to be 0 (or the error **EINVAL** results).
- Various bugs mean that for the purposes of permission checking, the case where both *tv_nsec* fields are set to **UTIME_NOW** isn't always treated the same as specifying *times* as NULL, and the case where one *tv_nsec* value is **UTIME_NOW** and the other is **UTIME_OMIT** isn't treated the same as specifying *times* as a pointer to an array of structures containing arbitrary time values. As a result, in some cases: a) file timestamps can be updated by a process that shouldn't have permission

to perform updates; b) file timestamps can't be updated by a process that should have permission to perform updates; and c) the wrong *errno* value is returned in case of an error.

- POSIX.1 says that a process that has *write access to the file* can make a call with *times* as NULL, or with *times* pointing to an array of structures in which both *tv_nsec* fields are **UTIME_NOW**, in order to update both timestamps to the current time. However, **futimens**() instead checks whether the *access mode of the file descriptor allows writing*.

SEE ALSO

chattr(1), *touch*(1), *futimesat*(2), *openat*(2), *stat*(2), *utimes*(2), *futimes*(3), *timespec*(3), *inode*(7), *path_resolution*(7), *symlink*(7)

NAME

vfork − create a child process and block parent

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

pid_t vfork(void);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

vfork():
  Since glibc 2.12:
    (_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
      || /* Since glibc 2.19: */ _DEFAULT_SOURCE
      || /* glibc <= 2.19: */ _BSD_SOURCE
  Before glibc 2.12:
    _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

Standard description

(From POSIX.1) The **vfork**() function has the same effect as *fork*(2), except that the behavior is undefined if the process created by **vfork**() either modifies any data other than a variable of type *pid_t* used to store the return value from **vfork**(), or returns from the function in which **vfork**() was called, or calls any other function before successfully calling *_exit*(2) or one of the *exec*(3) family of functions.

Linux description

vfork(), just like *fork*(2), creates a child process of the calling process. For details and return value and errors, see *fork*(2).

vfork() is a special case of *clone*(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an *execve*(2).

vfork() differs from *fork*(2) in that the calling thread is suspended until the child terminates (either normally, by calling *_exit*(2), or abnormally, after delivery of a fatal signal), or it makes a call to *execve*(2). Until that point, the child shares all memory with its parent, including the stack. The child must not return from the current function or call *exit*(3) (which would have the effect of calling exit handlers established by the parent process and flushing the parent's *stdio*(3) buffers), but may call *_exit*(2).

As with *fork*(2), the child process created by **vfork**() inherits copies of various of the caller's process attributes (e.g., file descriptors, signal dispositions, and current working directory); the **vfork**() call differs only in the treatment of the virtual address space, as described above.

Signals sent to the parent arrive after the child releases the parent's memory (i.e., after the child terminates or calls *execve*(2)).

Historic description

Under Linux, *fork*(2) is implemented using copy-on-write pages, so the only penalty incurred by *fork*(2) is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. However, in the bad old days a *fork*(2) would require making a complete copy of the caller's data space, often needlessly, since usually immediately afterward an *exec*(3) is done. Thus, for greater efficiency, BSD introduced the **vfork**() system call, which did not fully copy the address space of the parent process, but borrowed the parent's memory and thread of control until a call to *execve*(2) or an exit occurred. The parent process was suspended while the child was using its resources. The use of **vfork**() was tricky: for example, not modifying data in the parent process depended on knowing which variables were held in a register.

VERSIONS

The requirements put on **vfork**() by the standards are weaker than those put on *fork*(2), so an implementation where the two are synonymous is compliant. In particular, the programmer cannot rely on the parent remaining blocked until the child either terminates or calls *execve*(2), and cannot rely on any specific behavior with respect to shared memory.

Some consider the semantics of **vfork**() to be an architectural blemish, and the 4.2BSD man page

stated: "This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of *vfork* as it will, in that case, be made synonymous to *fork*." However, even though modern memory management hardware has decreased the performance difference between *fork*(2) and **vfork**(), there are various reasons why Linux and other systems have retained **vfork**():

- Some performance-critical applications require the small performance advantage conferred by **vfork**().
- **vfork**() can be implemented on systems that lack a memory-management unit (MMU), but *fork*(2) can't be implemented on such systems. (POSIX.1-2008 removed **vfork**() from the standard; the POSIX rationale for the *posix_spawn*(3) function notes that that function, which provides functionality equivalent to *fork*(2)+*exec*(3), is designed to be implementable on systems that lack an MMU.)
- On systems where memory is constrained, **vfork**() avoids the need to temporarily commit memory (see the description of */proc/sys/vm/overcommit_memory* in *proc*(5)) in order to execute a new program. (This can be especially beneficial where a large parent process wishes to execute a small helper program in a child process.) By contrast, using *fork*(2) in this scenario requires either committing an amount of memory equal to the size of the parent process (if strict overcommitting is in force) or overcommitting memory with the risk that a process is terminated by the out-of-memory (OOM) killer.

Linux notes

Fork handlers established using *pthread_atfork*(3) are not called when a multithreaded program employing the NPTL threading library calls **vfork**(). Fork handlers are called in this case in a program using the LinuxThreads threading library. (See *pthreads*(7) for a description of Linux threading libraries.)

A call to **vfork**() is equivalent to calling *clone*(2) with *flags* specified as:

```
CLONE_VM | CLONE_VFORK | SIGCHLD
```

STANDARDS

None.

HISTORY

4.3BSD; POSIX.1-2001 (but marked OBSOLETE). POSIX.1-2008 removes the specification of **vfork**().

The **vfork**() system call appeared in 3.0BSD. In 4.4BSD it was made synonymous to *fork*(2) but NetBSD introduced it again; see . In Linux, it has been equivalent to *fork*(2) until Linux 2.2.0-pre6 or so. Since Linux 2.2.0-pre9 (on i386, somewhat later on other architectures) it is an independent system call. Support was added in glibc 2.0.112.

CAVEATS

The child process should take care not to modify the memory in unintended ways, since such changes will be seen by the parent process once the child terminates or executes another program. In this regard, signal handlers can be especially problematic: if a signal handler that is invoked in the child of **vfork**() changes memory, those changes may result in an inconsistent process state from the perspective of the parent process (e.g., memory changes would be visible in the parent, but changes to the state of open file descriptors would not be visible).

When **vfork**() is called in a multithreaded process, only the calling thread is suspended until the child terminates or executes a new program. This means that the child is sharing an address space with other running code. This can be dangerous if another thread in the parent process changes credentials (using *setuid*(2) or similar), since there are now two processes with different privilege levels running in the same address space. As an example of the dangers, suppose that a multithreaded program running as root creates a child using **vfork**(). After the **vfork**(), a thread in the parent process drops the process to an unprivileged user in order to run some untrusted code (e.g., perhaps via plug-in opened with *dlopen*(3)). In this case, attacks are possible where the parent process uses *mmap*(2) to map in code that will be executed by the privileged child process.

BUGS

Details of the signal handling are obscure and differ between systems. The BSD man page states: "To avoid a possible deadlock situation, processes that are children in the middle of a **vfork**() are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or *ioctl*s are allowed and input attempts result in an

end-of-file indication."

SEE ALSO

clone(2), *execve*(2), *_exit*(2), *fork*(2), *unshare*(2), *wait*(2)

NAME

vhangup − virtually hangup the current terminal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int vhangup(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

vhangup():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

vhangup() simulates a hangup on the current terminal. This call arranges for other users to have a "clean" terminal at login time.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EPERM
The calling process has insufficient privilege to call **vhangup**(); the **CAP_SYS_TTY_CONFIG** capability is required.

STANDARDS

Linux.

SEE ALSO

init(1), *capabilities*(7)

NAME

vm86old, vm86 − enter virtual 8086 mode

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/vm86.h>

int vm86old(struct vm86_struct ****info*);**
int vm86(unsigned long *fn***, struct vm86plus_struct ****v86***);**

DESCRIPTION

The system call **vm86**() was introduced in Linux 0.97p2. In Linux 2.1.15 and 2.0.28, it was renamed to **vm86old**(), and a new **vm86**() was introduced. The definition of *struct vm86_struct* was changed in 1.1.8 and 1.1.9.

These calls cause the process to enter VM86 mode (virtual-8086 in Intel literature), and are used by **dosemu**.

VM86 mode is an emulation of real mode within a protected mode task.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
: This return value is specific to i386 and indicates a problem with getting user-space data.

ENOSYS
: This return value indicates the call is not implemented on the present architecture.

EPERM
: Saved kernel stack exists. (This is a kernel sanity check; the saved stack should exist only within vm86 mode itself.)

STANDARDS

Linux on 32-bit Intel processors.

NAME

vmsplice − splice user pages to/from a pipe

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <fcntl.h>

ssize_t vmsplice(int fd, const struct iovec *iov,
                 size_t nr_segs, unsigned int flags);
```

DESCRIPTION

If *fd* is opened for writing, the **vmsplice**() system call maps *nr_segs* ranges of user memory described by *iov* into a pipe. If *fd* is opened for reading, the **vmsplice**() system call fills *nr_segs* ranges of user memory described by *iov* from a pipe. The file descriptor *fd* must refer to a pipe.

The pointer *iov* points to an array of *iovec* structures as described in *iovec*(3type).

The *flags* argument is a bit mask that is composed by ORing together zero or more of the following values:

SPLICE_F_MOVE
: Unused for **vmsplice**(); see *splice*(2).

SPLICE_F_NONBLOCK
: Do not block on I/O; see *splice*(2) for further details.

SPLICE_F_MORE
: Currently has no effect for **vmsplice**(), but may be implemented in the future; see *splice*(2).

SPLICE_F_GIFT
: The user pages are a gift to the kernel. The application may not modify this memory ever, otherwise the page cache and on-disk data may differ. Gifting pages to the kernel means that a subsequent *splice*(2) **SPLICE_F_MOVE** can successfully move the pages; if this flag is not specified, then a subsequent *splice*(2) **SPLICE_F_MOVE** must copy the pages. Data must also be properly page aligned, both in memory and length.

RETURN VALUE

Upon successful completion, **vmsplice**() returns the number of bytes transferred to the pipe. On error, **vmsplice**() returns −1 and *errno* is set to indicate the error.

ERRORS

EAGAIN
: **SPLICE_F_NONBLOCK** was specified in *flags*, and the operation would block.

EBADF
: *fd* either not valid, or doesn't refer to a pipe.

EINVAL
: *nr_segs* is greater than **IOV_MAX**; or memory not aligned if **SPLICE_F_GIFT** set.

ENOMEM
: Out of memory.

STANDARDS

Linux.

HISTORY

Linux 2.6.17, glibc 2.5.

NOTES

vmsplice() follows the other vectorized read/write type functions when it comes to limitations on the number of segments being passed in. This limit is **IOV_MAX** as defined in <*limits.h*>. Currently, this limit is 1024.

vmsplice() really supports true splicing only from user memory to a pipe. In the opposite direction, it actually just copies the data to user space. But this makes the interface nice and symmetric and enables people to build on **vmsplice**() with room for future improvement in performance.

SEE ALSO

splice(2), *tee*(2), *pipe*(7)

NAME

wait, waitpid, waitid − wait for process to change state

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                /* This is the glibc and POSIX interface; see
                   NOTES for information on the raw system call. */
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

waitid():
Since glibc 2.26:
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
glibc 2.25 and earlier:
_XOPEN_SOURCE
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of *sigaction*(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(−1, &wstatus, 0);
```

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< **−1** meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.

−1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid**().

> **0** meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG
return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via *ptrace*(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

(For Linux-only options, see below.)

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and *waitpid*()!):

WIFEXITED(*wstatus***)**
returns true if the child terminated normally, that is, by calling *exit*(3) or *_exit*(2), or by returning from main().

WEXITSTATUS(*wstatus***)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to *exit*(3) or *_exit*(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus***)**
returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus***)**
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

WCOREDUMP(*wstatus***)**
returns true if the child produced a core dump (see *core*(5)). This macro should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Therefore, enclose its use inside *#ifdef WCOREDUMP ... #endif*.

WIFSTOPPED(*wstatus***)**
returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using **WUNTRACED** or when the child is being traced (see *ptrace*(2)).

WSTOPSIG(*wstatus***)**
returns the number of the signal which caused the child to stop. This macro should be employed only if **WIFSTOPPED** returned true.

WIFCONTINUED(*wstatus***)**
(since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitid()
The **waitid**() system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as follows:

idtype == **P_PID**
Wait for the child whose process ID matches *id*.

idtype == **P_PIDFD** (since Linux 5.4)
Wait for the child referred to by the PID file descriptor specified in *id*. (See *pidfd_open*(2) for further information on PID file descriptors.)

idtype == **P_PGID**
Wait for any child whose process group ID matches *id*. Since Linux 5.4, if *id* is zero, then wait for any child that is in the same process group as the caller's process group at the time of the call.

idtype == **P_ALL**
Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in *options*:

WEXITED
Wait for children that have terminated.

WSTOPPED
Wait for children that have been stopped by delivery of a signal.

WCONTINUED
Wait for (previously stopped) children that have been resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

WNOHANG
As for **waitpid**().

WNOWAIT
Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, **waitid**() fills in the following fields of the *siginfo_t* structure pointed to by *infop*:

si_pid The process ID of the child.

si_uid The real user ID of the child. (This field is not set on most other implementations.)

si_signo
Always set to **SIGCHLD**.

si_status
Either the exit status of the child, as given to *_exit*(2) (or *exit*(3)), or the signal that caused the child to terminate, stop, or continue. The *si_code* field can be used to determine how to interpret this field.

si_code Set to one of: **CLD_EXITED** (child called *_exit*(2)); **CLD_KILLED** (child killed by signal); **CLD_DUMPED** (child killed by signal, and dumped core); **CLD_STOPPED** (child stopped by signal); **CLD_TRAPPED** (traced child has trapped); or **CLD_CONTINUED** (child continued by **SIGCONT**).

If **WNOHANG** was specified in *options* and there were no children in a waitable state, then **waitid**() returns 0 immediately and the state of the *siginfo_t* structure pointed to by *infop* depends on the implementation. To (portably) distinguish this case from that where a child was in a waitable state, zero out the *si_pid* field before the call and check for a nonzero value in this field after the call returns.

POSIX.1-2008 Technical Corrigendum 1 (2013) adds the requirement that when **WNOHANG** is specified in *options* and there were no children in a waitable state, then **waitid**() should zero out the *si_pid* and *si_signo* fields of the structure. On Linux and other implementations that adhere to this requirement, it is not necessary to zero out the *si_pid* field before calling **waitid**(). However, not all implementations follow the POSIX.1 specification on this point.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on failure, −1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On failure, −1 is returned.

waitid(): returns 0 on success or if **WNOHANG** was specified and no child(ren) specified by *id* has yet changed state; on failure, −1 is returned.

On failure, each of these calls sets *errno* to indicate the error.

ERRORS

EAGAIN
The PID file descriptor specified in *id* is nonblocking and the process that it refers to has not terminated.

ECHILD
(for *wait*()) The calling process does not have any unwaited-for children.

ECHILD
(for **waitpid**() or *waitid*()) The process specified by *pid* (**waitpid**()) or *idtype* and *id* (**waitid**()) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for **SIGCHLD** is set to **SIG_IGN**. See also the *Linux Notes* section

about threads.)

EINTR
WNOHANG was not set and an unblocked signal or a **SIGCHLD** was caught; see *signal*(7).

EINVAL
The *options* argument was invalid.

ESRCH
(for **wait**() or *waitpid*()) *pid* is equal to **INT_MIN**.

VERSIONS

C library/kernel differences

wait() is actually a library function that (in glibc) is implemented as a call to *wait4*(2).

On some architectures, there is no **waitpid**() system call; instead, this interface is implemented via a C library wrapper function that calls *wait4*(2).

The raw **waitid**() system call takes a fifth argument, of type *struct rusage* *. If this argument is non-NULL, then it is used to return resource usage information about the child, in the same manner as *wait4*(2). See *getrusage*(2) for details.

STANDARDS

POSIX.1-2008.

HISTORY

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by *init*(1), (or by the nearest "subreaper" process as defined through the use of the *prctl*(2) **PR_SET_CHILD_SUBREAPER** operation); *init*(1) automatically performs a wait to remove the zombies.

POSIX.1-2001 specifies that if the disposition of **SIGCHLD** is set to **SIG_IGN** or the **SA_NOCLDWAIT** flag is set for **SIGCHLD** (see *sigaction*(2)), then children that terminate do not become zombies and a call to **wait**() or **waitpid**() will block until all children have terminated, and then fail with *errno* set to **ECHILD**. (The original POSIX standard left the behavior of setting **SIGCHLD** to **SIG_IGN** unspecified. Note that even though the default disposition of **SIGCHLD** is "ignore", explicitly setting the disposition to **SIG_IGN** results in different treatment of zombie process children.)

Linux 2.6 conforms to the POSIX requirements. However, Linux 2.4 (and earlier) does not: if a **wait**() or **waitpid**() call is made while **SIGCHLD** is being ignored, the call behaves just as though **SIGCHLD** were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

Linux notes

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique *clone*(2) system call; other routines such as the portable *pthread_create*(3) call are implemented using *clone*(2). Before Linux 2.4, a thread was just a special case of a process, and as a consequence one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.

The following Linux-specific *options* are for use with children created using *clone*(2); they can also, since Linux 4.7, be used with **waitid**():

__WCLONE
Wait for "clone" children only. If omitted, then wait for "non-clone" children only. (A "clone" child is one which delivers no signal, or a signal other than **SIGCHLD** to its parent upon termination.) This option is ignored if **__WALL** is also specified.

__WALL (since Linux 2.4)
Wait for all children, regardless of type ("clone" or "non-clone").

__WNOTHREAD (since Linux 2.4)
Do not wait for children of other threads in the same thread group. This was the default before Linux 2.4.

Since Linux 4.7, the **__WALL** flag is automatically implied if the child is being ptraced.

BUGS

According to POSIX.1-2008, an application calling **waitid**() must ensure that *infop* points to a *siginfo_t* structure (i.e., that it is a non-null pointer). On Linux, if *infop* is NULL, **waitid**() succeeds, and returns the process ID of the waited-for child. Applications should avoid relying on this inconsistent, nonstandard, and unnecessary feature.

EXAMPLES

The following program demonstrates the use of *fork*(2) and **waitpid**(). The program creates a child process. If no command-line argument is supplied to the program, then the child suspends its execution using *pause*(2), to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(), and uses the W*() macros described above to analyze the wait status value.

The following shell session demonstrates the use of the program:

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

Program source

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int    wstatus;
    pid_t  cpid, w;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));
```

```
        } else {                    /* Code executed by parent */
            do {
                w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
                if (w == -1) {
                    perror("waitpid");
                    exit(EXIT_FAILURE);
                }

                if (WIFEXITED(wstatus)) {
                    printf("exited, status=%d\n", WEXITSTATUS(wstatus));
                } else if (WIFSIGNALED(wstatus)) {
                    printf("killed by signal %d\n", WTERMSIG(wstatus));
                } else if (WIFSTOPPED(wstatus)) {
                    printf("stopped by signal %d\n", WSTOPSIG(wstatus));
                } else if (WIFCONTINUED(wstatus)) {
                    printf("continued\n");
                }
            } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
            exit(EXIT_SUCCESS);
        }
    }
```

SEE ALSO

_exit(2), *clone*(2), *fork*(2), *kill*(2), *ptrace*(2), *sigaction*(2), *signal*(2), *wait4*(2), *pthread_create*(3), *core*(5), *credentials*(7), *signal*(7)

NAME

wait3, wait4 − wait for process to change state, BSD style

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait3(int *_Nullable wstatus, int options,
            struct rusage *_Nullable rusage);
pid_t wait4(pid_t pid, int *_Nullable wstatus, int options,
            struct rusage *_Nullable rusage);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wait3():
Since glibc 2.26:
_DEFAULT_SOURCE
|| (_XOPEN_SOURCE >= 500 &&
! (_POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 600))
From glibc 2.19 to glibc 2.25:
_DEFAULT_SOURCE || _XOPEN_SOURCE >= 500
glibc 2.19 and earlier:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

wait4():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

These functions are nonstandard; in new programs, the use of *waitpid*(2) or *waitid*(2) is preferable.

The **wait3**() and **wait4**() system calls are similar to *waitpid*(2), but additionally return resource usage information about the child in the structure pointed to by *rusage*.

Other than the use of the *rusage* argument, the following **wait3**() call:

```
wait3(wstatus, options, rusage);
```

is equivalent to:

```
waitpid(-1, wstatus, options);
```

Similarly, the following **wait4**() call:

```
wait4(pid, wstatus, options, rusage);
```

is equivalent to:

```
waitpid(pid, wstatus, options);
```

In other words, **wait3**() waits of any child, while **wait4**() can be used to select a specific child, or children, on which to wait. See *wait*(2) for further details.

If *rusage* is not NULL, the *struct rusage* to which it points will be filled with accounting information about the child. See *getrusage*(2) for details.

RETURN VALUE

As for *waitpid*(2).

ERRORS

As for *waitpid*(2).

STANDARDS

None.

HISTORY

4.3BSD.

SUSv1 included a specification of **wait3**(); SUSv2 included **wait3**(), but marked it LEGACY; SUSv3 removed it.

Including <*sys/time.h*> is not required these days, but increases portability. (Indeed, <*sys/resource.h*> defines the *rusage* structure with fields of type *struct timeval* defined in <*sys/time.h*>.)

C library/kernel differences

On Linux, **wait3**() is a library function implemented on top of the **wait4**() system call.

SEE ALSO

fork(2), *getrusage*(2), *sigaction*(2), *signal*(2), *wait*(2), *signal*(7)

NAME

write − write to a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd***, const void** *buf***[.***count***], size_t** *count***);**

DESCRIPTION

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see *setrlimit*(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also *pipe*(7).)

For a seekable file (i.e., one to which *lseek*(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was *open*(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a *read*(2) that can be proved to occur after a **write**() has returned will return the new data. Note that not all filesystems are POSIX conforming.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE

On success, the number of bytes written is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note that a successful **write**() may transfer fewer than *count* bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked **write**() to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes. In the event of a partial write, the caller can make another **write**() call to transfer the remaining bytes. The subsequent call will either transfer further bytes or may result in an error (e.g., if the disk is now full).

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if one of the errors below is detected. If no errors are detected, or error detection is not performed, 0 is returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

ERRORS

EAGAIN
: The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. See *open*(2) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**
: The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF
: *fd* is not a valid file descriptor or is not open for writing.

EDESTADDRREQ
: *fd* refers to a datagram socket for which a peer address has not been set using *connect*(2).

EDQUOT
: The user's quota of disk blocks on the filesystem containing the file referred to by *fd* has been exhausted.

EFAULT
buf is outside your accessible address space.

EFBIG
An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process's file size limit, or to write at a position past the maximum allowed offset.

EINTR
The call was interrupted by a signal before any data was written; see *signal*(7).

EINVAL
fd is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EIO A low-level I/O error occurred while modifying the inode. This error may relate to the write-back of data written by an earlier **write**(), which may have been issued to a different file descriptor on the same file. Since Linux 4.13, errors from write-back come with a promise that they *may* be reported by subsequent. **write**() requests, and *will* be reported by a subsequent *fsync*(2) (whether or not they were also reported by *write*())An alternate cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost locks* section of *fcntl*(2) for further details.

ENOSPC
The device containing the file referred to by *fd* has no room for the data.

EPERM
The operation was prevented by a file seal; see *fcntl*(2).

EPIPE *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a **SIGPIPE** signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)

Other errors may occur, depending on the object connected to *fd*.

STANDARDS
POSIX.1-2008.

HISTORY
SVr4, 4.3BSD, POSIX.1-2001.

Under SVr4 a write may be interrupted and return **EINTR** at any point, not just before any data is written.

NOTES
A successful return from **write**() does not make any guarantee that data has been committed to disk. On some filesystems, including NFS, it does not even guarantee that space has successfully been reserved for the data. In this case, some errors might be delayed until a future **write**(), *fsync*(2), or even *close*(2). The only way to be sure is to call *fsync*(2) after you are done writing all your data.

If a **write**() is interrupted by a signal handler before any bytes are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.

On Linux, **write**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

An error return value while performing **write**() using direct I/O does not mean the entire write has failed. Partial data may be written and the data at the file offset on which the **write**() was attempted should be considered inconsistent.

BUGS
According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

> All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **write**() and *writev*(2). And among the effects that should be

atomic across threads (and processes) are updates of the file offset. However, before Linux 3.14, this was not the case: if two processes that share an open file description (see *open*(2)) perform a **write**() (or *writev*(2)) at the same time, then the I/O operations were not atomic with respect to updating the file offset, with the result that the blocks of data output by the two processes might (incorrectly) overlap. This problem was fixed in Linux 3.14.

SEE ALSO

close(2), *fcntl*(2), *fsync*(2), *ioctl*(2), *lseek*(2), *open*(2), *pwrite*(2), *read*(2), *select*(2), *writev*(2), *fwrite*(3)

NAME

open_how − how to open a pathname

LIBRARY

Linux kernel headers

SYNOPSIS

```
#include <linux/openat2.h>

struct open_how {
    u64  flags;    /* O_* flags */
    u64  mode;     /* Mode for O_{CREAT,TMPFILE} */
    u64  resolve;  /* RESOLVE_* flags */
    /* ... */
};
```

DESCRIPTION

Specifies how a pathname should be opened.

The fields are as follows:

flags This field specifies the file creation and file status flags to use when opening the file.

mode This field specifies the mode for the new file.

resolve This is a bit mask of flags that modify the way in which *all* components of a pathname will be resolved (see *path_resolution*(7) for background information).

VERSIONS

Extra fields may be appended to the structure, with a zero value in a new field resulting in the kernel behaving as though that extension field was not present. Therefore, a user *must* zero-fill this structure on initialization.

STANDARDS

Linux.

SEE ALSO

openat2(2)

NAME

intro − introduction to library functions

DESCRIPTION

Section 3 of the manual describes all library functions excluding the library functions (system call wrappers) described in Section 2, which implement system calls.

Many of the functions described in the section are part of the Standard C Library (*libc*). Some functions are part of other libraries (e.g., the math library, *libm*, or the real-time library, *librt*) in which case the manual page will indicate the linker option needed to link against the required library (e.g., *−lm* and *−lrt*, respectively, for the aforementioned libraries).

In some cases, the programmer must define a feature test macro in order to obtain the declaration of a function from the header file specified in the man page SYNOPSIS section. (Where required, these *feature test macros* must be defined before including *any* header files.) In such cases, the required macro is described in the man page. For further information on feature test macros, see *feature_test_macros*(7).

Subsections

Section 3 of this manual is organized into subsections that reflect the complex structure of the standard C library and its many implementations:

- 3const
- 3head
- 3type

This difficult history frequently makes it a poor example to follow in design, implementation, and presentation.

Ideally, a library for the C language is designed such that each header file presents the interface to a coherent software module. It provides a small number of function declarations and exposes only data types and constants that are required for use of those functions. Together, these are termed an API or *application program interface*. Types and constants to be shared among multiple APIs should be placed in header files that declare no functions. This organization permits a C library module to be documented concisely with one header file per manual page. Such an approach improves the readability and accessibility of library documentation, and thereby the usability of the software.

STANDARDS

Certain terms and abbreviations are used to indicate UNIX variants and standards to which calls in this section conform. See *standards*(7).

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

SEE ALSO

intro(2), *errno*(3), *capabilities*(7), *credentials*(7), *environ*(7), *feature_test_macros*(7), *libc*(7), *math_error*(7), *path_resolution*(7), *pthreads*(7), *signal*(7), *standards*(7), *system_data_types*(7)

NAME

a64l, l64a − convert between long and base-64

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

long a64l(const char *str64);
char *l64a(long value);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

a64l(), **l64a**():
_XOPEN_SOURCE >= 500
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

DESCRIPTION

These functions provide a conversion between 32-bit long integers and little-endian base-64 ASCII strings (of length zero to six). If the string used as argument for **a64l**() has length greater than six, only the first six bytes are used. If the type *long* has more than 32 bits, then **l64a**() uses only the low order 32 bits of *value*, and **a64l**() sign-extends its 32-bit result.

The 64 digits in the base-64 system are:

'.'	represents a 0
'/'	represents a 1
0-9	represent 2-11
A-Z	represent 12-37
a-z	represent 38-63

So 123 = 59*64^0 + 1*64^1 = "v/".

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
l64a()	Thread safety	MT-Unsafe race:l64a
a64l()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The value returned by **l64a**() may be a pointer to a static buffer, possibly overwritten by later calls.

The behavior of **l64a**() is undefined when *value* is negative. If *value* is zero, it returns an empty string.

These functions are broken before glibc 2.2.5 (puts most significant digit first).

This is not the encoding used by *uuencode*(1)

SEE ALSO

uuencode(1), *strtoul*(3)

NAME

abort − cause abnormal process termination

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

[[noreturn]] void abort(void);

DESCRIPTION

The **abort**() function first unblocks the **SIGABRT** signal, and then raises that signal for the calling process (as though *raise*(3) was called). This results in the abnormal termination of the process unless the **SIGABRT** signal is caught and the signal handler does not return (see *longjmp*(3)).

If the **SIGABRT** signal is ignored, or caught by a handler that returns, the **abort**() function will still terminate the process. It does this by restoring the default disposition for **SIGABRT** and then raising the signal for a second time.

As with other cases of abnormal termination the functions registered with *atexit*(3) and *on_exit*(3) are not called.

RETURN VALUE

The **abort**() function never returns.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
abort()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

SVr4, POSIX.1-2001, 4.3BSD, C89.

Up until glibc 2.26, if the **abort**() function caused process termination, all open streams were closed and flushed (as with *fclose*(3)). However, in some cases this could result in deadlocks and data corruption. Therefore, starting with glibc 2.27, **abort**() terminates the process without flushing streams. POSIX.1 permits either possible behavior, saying that **abort**() "may include an attempt to effect fclose() on all open streams".

SEE ALSO

gdb(1), *sigaction*(2), *assert*(3), *exit*(3), *longjmp*(3), *raise*(3)

NAME

abs, labs, llabs, imaxabs − compute the absolute value of an integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int abs(int j);
long labs(long j);
long long llabs(long long j);

#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

llabs():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **abs**() function computes the absolute value of the integer argument *j*. The **labs**(), **llabs**(), and **imaxabs**() functions compute the absolute value of the argument *j* of the appropriate integer type for the function.

RETURN VALUE

Returns the absolute value of the integer argument, of the appropriate integer type for the function.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
abs(), **labs**(), **llabs**(), **imaxabs**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99, SVr4, 4.3BSD.

C89 only includes the **abs**() and **labs**() functions; the functions **llabs**() and **imaxabs**() were added in C99.

NOTES

Trying to take the absolute value of the most negative integer is not defined.

The **llabs**() function is included since glibc 2.0. The **imaxabs**() function is included since glibc 2.1.1.

For **llabs**() to be declared, it may be necessary to define **_ISOC99_SOURCE** or **_ISOC9X_SOURCE** (depending on the version of glibc) before including any standard headers.

By default, GCC handles **abs**(), **labs**(), and (since GCC 3.0) **llabs**() and **imaxabs**() as built-in functions.

SEE ALSO

cabs(3), *ceil*(3), *fabs*(3), *floor*(3), *rint*(3)

NAME

acos, acosf, acosl − arc cosine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

acosf(), **acosl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the arc cosine of *x*; that is the value whose cosine is *x*.

RETURN VALUE

On success, these functions return the arc cosine of *x* in radians; the return value is in the range [0, pi].

If *x* is a NaN, a NaN is returned.

If *x* is +1, +0 is returned.

If *x* is positive infinity or negative infinity, a domain error occurs, and a NaN is returned.

If *x* is outside the range [−1, 1], a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is outside the range [−1, 1]
errno is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
acos(), **acosf**(), **acosl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to C89, SVr4, 4.3BSD.

SEE ALSO

asin(3), *atan*(3), *atan2*(3), *cacos*(3), *cos*(3), *sin*(3), *tan*(3)

NAME

acosh, acoshf, acoshl − inverse hyperbolic cosine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

acosh():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

acoshf(), **acoshl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the inverse hyperbolic cosine of *x*; that is the value whose hyperbolic cosine is *x*.

RETURN VALUE

On success, these functions return the inverse hyperbolic cosine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +1, +0 is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is less than 1, a domain error occurs, and the functions return a NaN.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is less than 1
errno is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
acosh(), **acoshf**(), **acoshl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

asinh(3), *atanh*(3), *cacosh*(3), *cosh*(3), *sinh*(3), *tanh*(3)

NAME

addseverity − introduce new severity classes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fmtmsg.h>

int addseverity(int *severity***, const char *****s***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

addseverity():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

This function allows the introduction of new severity classes which can be addressed by the *severity* argument of the *fmtmsg*(3) function. By default, that function knows only how to print messages for severity 0-4 (with strings (none), HALT, ERROR, WARNING, INFO). This call attaches the given string *s* to the given value *severity*. If *s* is NULL, the severity class with the numeric value *severity* is removed. It is not possible to overwrite or remove one of the default severity classes. The severity value must be nonnegative.

RETURN VALUE

Upon success, the value **MM_OK** is returned. Upon error, the return value is **MM_NOTOK**. Possible errors include: out of memory, attempt to remove a nonexistent or default severity class.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
addseverity()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1. System V.

NOTES

New severity classes can also be added by setting the environment variable **SEV_LEVEL**.

SEE ALSO

fmtmsg(3)

NAME

adjtime − correct the time to synchronize the system clock

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/time.h>

int adjtime(const struct timeval ****delta*, struct timeval ****olddelta***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

adjtime():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **adjtime**() function gradually adjusts the system clock (as returned by *gettimeofday*(2)). The amount of time by which the clock is to be adjusted is specified in the structure pointed to by *delta*. This structure has the following form:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

If the adjustment in *delta* is positive, then the system clock is speeded up by some small percentage (i.e., by adding a small amount of time to the clock value in each second) until the adjustment has been completed. If the adjustment in *delta* is negative, then the clock is slowed down in a similar fashion.

If a clock adjustment from an earlier **adjtime**() call is already in progress at the time of a later **adjtime**() call, and *delta* is not NULL for the later call, then the earlier adjustment is stopped, but any already completed part of that adjustment is not undone.

If *olddelta* is not NULL, then the buffer that it points to is used to return the amount of time remaining from any previous adjustment that has not yet been completed.

RETURN VALUE

On success, **adjtime**() returns 0. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
The adjustment in *delta* is outside the permitted range.

EPERM
The caller does not have sufficient privilege to adjust the time. Under Linux, the **CAP_SYS_TIME** capability is required.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
adjtime()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD, System V.

NOTES

The adjustment that **adjtime**() makes to the clock is carried out in such a manner that the clock is always monotonically increasing. Using **adjtime**() to adjust the time prevents the problems that could be caused for certain applications (e.g., *make*(1)) by abrupt positive or negative jumps in the system time.

adjtime() is intended to be used to make small adjustments to the system time. Most systems impose a limit on the adjustment that can be specified in *delta*. In the glibc implementation, *delta* must be less

than or equal to (INT_MAX / 1000000 − 2) and greater than or equal to (INT_MIN / 1000000 + 2) (respectively 2145 and −2145 seconds on i386).

BUGS

A longstanding bug meant that if *delta* was specified as NULL, no valid information about the outstanding clock adjustment was returned in *olddelta*. (In this circumstance, **adjtime**() should return the outstanding clock adjustment, without changing it.) This bug is fixed on systems with glibc 2.8 or later and Linux kernel 2.6.26 or later.

SEE ALSO

adjtimex(2), *gettimeofday*(2), *time*(7)

NAME

aio_cancel − cancel an outstanding asynchronous I/O request

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

int aio_cancel(int *fd***, struct aiocb ****aiocbp***);**

DESCRIPTION

The **aio_cancel**() function attempts to cancel outstanding asynchronous I/O requests for the file descriptor *fd*. If *aiocbp* is NULL, all such requests are canceled. Otherwise, only the request described by the control block pointed to by *aiocbp* is canceled. (See *aio*(7) for a description of the *aiocb* structure.)

Normal asynchronous notification occurs for canceled requests (see *aio*(7) and *sigevent*(3type)). The request return status (**aio_return**(3)) is set to −1, and the request error status (**aio_error**(3)) is set to **ECANCELED**. The control block of requests that cannot be canceled is not changed.

If the request could not be canceled, then it will terminate in the usual way after performing the I/O operation. (In this case, *aio_error*(3) will return the status **EINPROGRESSS**.)

If *aiocbp* is not NULL, and *fd* differs from the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Which operations are cancelable is implementation-defined.

RETURN VALUE

The **aio_cancel**() function returns one of the following values:

AIO_CANCELED
: All requests were successfully canceled.

AIO_NOTCANCELED
: At least one of the requests specified was not canceled because it was in progress. In this case, one may check the status of individual requests using *aio_error*(3).

AIO_ALLDONE
: All requests had already been completed before the call.

−1
: An error occurred. The cause of the error can be found by inspecting *errno*.

ERRORS

EBADF
: *fd* is not a valid file descriptor.

ENOSYS
: **aio_cancel**() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_cancel()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

EXAMPLES

See *aio*(7).

SEE ALSO

aio_error(3), *aio_fsync*(3), *aio_read*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3), *aio*(7)

NAME

aio_error − get error status of asynchronous I/O operation

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

int aio_error(const struct aiocb ****aiocbp*);**

DESCRIPTION

The **aio_error**() function returns the error status for the asynchronous I/O request with control block pointed to by *aiocbp*. (See *aio*(7) for a description of the *aiocb* structure.)

RETURN VALUE

This function returns one of the following:

EINPROGRESS
: if the request has not been completed yet.

ECANCELED
: if the request was canceled.

0
: if the request completed successfully.

> **0**
: A positive error number, if the asynchronous I/O operation failed. This is the same value that would have been stored in the *errno* variable in the case of a synchronous *read*(2), *write*(2), *fsync*(2), or *fdatasync*(2) call.

ERRORS

EINVAL
: *aiocbp* does not point at a control block for an asynchronous I/O request of which the return status (see *aio_return*(3)) has not been retrieved yet.

ENOSYS
: **aio_error**() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_error()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

EXAMPLES

See *aio*(7).

SEE ALSO

aio_cancel(3), *aio_fsync*(3), *aio_read*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3), *aio*(7)

NAME

aio_fsync − asynchronous file synchronization

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

int aio_fsync(int *op***, struct aiocb ***_aiocbp_**);**

DESCRIPTION

The **aio_fsync**() function does a sync on all outstanding asynchronous I/O operations associated with *aiocbp−>aio_fildes*. (See *aio*(7) for a description of the *aiocb* structure.)

More precisely, if *op* is **O_SYNC**, then all currently queued I/O operations shall be completed as if by a call of *fsync*(2), and if *op* is **O_DSYNC**, this call is the asynchronous analog of *fdatasync*(2).

Note that this is a request only; it does not wait for I/O completion.

Apart from *aio_fildes*, the only field in the structure pointed to by *aiocbp* that is used by this call is the *aio_sigevent* field (a *sigevent* structure, described in *sigevent*(3type)), which indicates the desired type of asynchronous notification at completion. All other fields are ignored.

RETURN VALUE

On success (the sync request was successfully queued) this function returns 0. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: Out of resources.

EBADF
: *aio_fildes* is not a valid file descriptor open for writing.

EINVAL
: Synchronized I/O is not supported for this file, or *op* is not **O_SYNC** or **O_DSYNC**.

ENOSYS
: **aio_fsync**() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_fsync()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_read*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3), *aio*(7), *sigevent*(3type)

NAME

aio_init − asynchronous I/O initialization

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <aio.h>

void aio_init(const struct aioinit **init*);

DESCRIPTION

The GNU-specific **aio_init**() function allows the caller to provide tuning hints to the glibc POSIX AIO implementation. Use of this function is optional, but to be effective, it must be called before employing any other functions in the POSIX AIO API.

The tuning information is provided in the buffer pointed to by the argument *init*. This buffer is a structure of the following form:

```
struct aioinit {
    int aio_threads;    /* Maximum number of threads */
    int aio_num;        /* Number of expected simultaneous
                           requests */
    int aio_locks;      /* Not used */
    int aio_usedba;     /* Not used */
    int aio_debug;      /* Not used */
    int aio_numusers;   /* Not used */
    int aio_idle_time;  /* Number of seconds before idle thread
                           terminates (since glibc 2.2) */
    int aio_reserved;
};
```

The following fields are used in the *aioinit* structure:

aio_threads
: This field specifies the maximum number of worker threads that may be used by the implementation. If the number of outstanding I/O operations exceeds this limit, then excess operations will be queued until a worker thread becomes free. If this field is specified with a value less than 1, the value 1 is used. The default value is 20.

aio_num
: This field should specify the maximum number of simultaneous I/O requests that the caller expects to enqueue. If a value less than 32 is specified for this field, it is rounded up to 32. The default value is 64.

aio_idle_time
: This field specifies the amount of time in seconds that a worker thread should wait for further requests before terminating, after having completed a previous request. The default value is 1.

STANDARDS

GNU.

HISTORY

glibc 2.1.

SEE ALSO

aio(7)

NAME

aio_read − asynchronous read

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

int aio_read(struct aiocb **aiocbp*);

DESCRIPTION

The **aio_read**() function queues the I/O request described by the buffer pointed to by *aiocbp*. This function is the asynchronous analog of *read*(2). The arguments of the call

```
read(fd, buf, count)
```

correspond (in order) to the fields *aio_fildes*, *aio_buf*, and *aio_nbytes* of the structure pointed to by *aiocbp*. (See *aio*(7) for a description of the *aiocb* structure.)

The data is read starting at the absolute position *aiocbp−>aio_offset*, regardless of the file offset. After the call, the value of the file offset is unspecified.

The "asynchronous" means that this call returns as soon as the request has been enqueued; the read may or may not have completed when the call returns. One tests for completion using *aio_error*(3). The return status of a completed I/O operation can be obtained by *aio_return*(3). Asynchronous notification of I/O completion can be obtained by setting *aiocbp−>aio_sigevent* appropriately; see *sigevent*(3type) for details.

If **_POSIX_PRIORITIZED_IO** is defined, and this file supports it, then the asynchronous operation is submitted at a priority equal to that of the calling process minus *aiocbp−>aio_reqprio*.

The field *aiocbp−>aio_lio_opcode* is ignored.

No data is read from a regular file beyond its maximum offset.

RETURN VALUE

On success, 0 is returned. On error, the request is not enqueued, −1 is returned, and *errno* is set to indicate the error. If an error is detected only later, it will be reported via *aio_return*(3) (returns status −1) and *aio_error*(3) (error status—whatever one would have gotten in *errno*, such as **EBADF**).

ERRORS

EAGAIN
: Out of resources.

EBADF
: *aio_fildes* is not a valid file descriptor open for reading.

EINVAL
: One or more of *aio_offset*, *aio_reqprio*, or *aio_nbytes* are invalid.

ENOSYS
: **aio_read**() is not implemented.

EOVERFLOW
: The file is a regular file, we start reading before end-of-file and want at least one byte, but the starting position is past the maximum offset for this file.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_read()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

It is a good idea to zero out the control block before use. The control block must not be changed while the read operation is in progress. The buffer area being read into must not be accessed during the operation or undefined results may occur. The memory areas involved must remain valid.

Simultaneous I/O operations specifying the same *aiocb* structure produce undefined results.

EXAMPLES

See *aio*(7).

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3), *aio*(7)

NAME

aio_return − get return status of asynchronous I/O operation

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

ssize_t aio_return(struct aiocb *****aiocbp*);**

DESCRIPTION

The **aio_return**() function returns the final return status for the asynchronous I/O request with control block pointed to by *aiocbp*. (See *aio*(7) for a description of the *aiocb* structure.)

This function should be called only once for any given request, after *aio_error*(3) returns something other than **EINPROGRESS**.

RETURN VALUE

If the asynchronous I/O operation has completed, this function returns the value that would have been returned in case of a synchronous *read*(2), *write*(2), *fsync*(2), or *fdatasync*(2), call. On error, −1 is returned, and *errno* is set to indicate the error.

If the asynchronous I/O operation has not yet completed, the return value and effect of **aio_return**() are undefined.

ERRORS

EINVAL

aiocbp does not point at a control block for an asynchronous I/O request of which the return status has not been retrieved yet.

ENOSYS

aio_return() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_return()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

EXAMPLES

See *aio*(7).

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_read*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3), *aio*(7)

NAME

aio_suspend − wait for asynchronous I/O operation or timeout

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <aio.h>

int aio_suspend(const struct aiocb *const aiocb_list[], int nitems,
                const struct timespec *restrict timeout);
```

DESCRIPTION

The **aio_suspend**() function suspends the calling thread until one of the following occurs:

- One or more of the asynchronous I/O requests in the list *aiocb_list* has completed.
- A signal is delivered.
- *timeout* is not NULL and the specified time interval has passed. (For details of the *timespec* structure, see *nanosleep*(2).)

The *nitems* argument specifies the number of items in *aiocb_list*. Each item in the list pointed to by *aiocb_list* must be either NULL (and then is ignored), or a pointer to a control block on which I/O was initiated using *aio_read*(3), *aio_write*(3), or *lio_listio*(3). (See *aio*(7) for a description of the *aiocb* structure.)

If **CLOCK_MONOTONIC** is supported, this clock is used to measure the timeout interval (see *clock_gettime*(2)).

RETURN VALUE

If this function returns after completion of one of the I/O requests specified in *aiocb_list*, 0 is returned. Otherwise, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: The call timed out before any of the indicated operations had completed.

EINTR
: The call was ended by signal (possibly the completion signal of one of the operations we were waiting for); see *signal*(7).

ENOSYS
: **aio_suspend**() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_suspend()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

POSIX doesn't specify the parameters to be *restrict*; that is specific to glibc.

NOTES

One can achieve polling by using a non-NULL *timeout* that specifies a zero time interval.

If one or more of the asynchronous I/O operations specified in *aiocb_list* has already completed at the time of the call to **aio_suspend**(), then the call returns immediately.

To determine which I/O operations have completed after a successful return from **aio_suspend**(), use *aio_error*(3) to scan the list of *aiocb* structures pointed to by *aiocb_list*.

BUGS

The glibc implementation of **aio_suspend**() is not async-signal-safe, in violation of the requirements of POSIX.1.

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_read*(3), *aio_return*(3), *aio_write*(3), *lio_listio*(3), *aio*(7), *time*(7)

NAME

aio_write − asynchronous write

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <aio.h>

int aio_write(struct aiocb ****aiocbp*);**

DESCRIPTION

The **aio_write**() function queues the I/O request described by the buffer pointed to by *aiocbp*. This function is the asynchronous analog of *write*(2). The arguments of the call

```
write(fd, buf, count)
```

correspond (in order) to the fields *aio_fildes*, *aio_buf*, and *aio_nbytes* of the structure pointed to by *aiocbp*. (See *aio*(7) for a description of the *aiocb* structure.)

If **O_APPEND** is not set, the data is written starting at the absolute position *aiocbp−>aio_offset*, regardless of the file offset. If **O_APPEND** is set, data is written at the end of the file in the same order as **aio_write**() calls are made. After the call, the value of the file offset is unspecified.

The "asynchronous" means that this call returns as soon as the request has been enqueued; the write may or may not have completed when the call returns. One tests for completion using *aio_error*(3). The return status of a completed I/O operation can be obtained *aio_return*(3). Asynchronous notification of I/O completion can be obtained by setting *aiocbp−>aio_sigevent* appropriately; see *sigevent*(3type) for details.

If **_POSIX_PRIORITIZED_IO** is defined, and this file supports it, then the asynchronous operation is submitted at a priority equal to that of the calling process minus *aiocbp−>aio_reqprio*.

The field *aiocbp−>aio_lio_opcode* is ignored.

No data is written to a regular file beyond its maximum offset.

RETURN VALUE

On success, 0 is returned. On error, the request is not enqueued, −1 is returned, and *errno* is set to indicate the error. If an error is detected only later, it will be reported via *aio_return*(3) (returns status −1) and *aio_error*(3) (error status—whatever one would have gotten in *errno*, such as **EBADF**).

ERRORS

EAGAIN
: Out of resources.

EBADF
: *aio_fildes* is not a valid file descriptor open for writing.

EFBIG
: The file is a regular file, we want to write at least one byte, but the starting position is at or beyond the maximum offset for this file.

EINVAL
: One or more of *aio_offset*, *aio_reqprio*, *aio_nbytes* are invalid.

ENOSYS
: **aio_write**() is not implemented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aio_write()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

It is a good idea to zero out the control block before use. The control block must not be changed while the write operation is in progress. The buffer area being written out must not be accessed during the operation or undefined results may occur. The memory areas involved must remain valid.

Simultaneous I/O operations specifying the same *aiocb* structure produce undefined results.

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_read*(3), *aio_return*(3), *aio_suspend*(3), *lio_listio*(3), *aio*(7)

NAME

alloca − allocate memory that is automatically freed

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <alloca.h>

void *alloca(size_t *size***);**

DESCRIPTION

The **alloca**() function allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called **alloca**() returns to its caller.

RETURN VALUE

The **alloca**() function returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behavior is undefined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
alloca()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

PWB, 32V.

NOTES

The **alloca**() function is machine- and compiler-dependent. Because it allocates from the stack, it's faster than *malloc*(3) and *free*(3). In certain cases, it can also simplify memory deallocation in applications that use *longjmp*(3) or *siglongjmp*(3). Otherwise, its use is discouraged.

Because the space allocated by **alloca**() is allocated within the stack frame, that space is automatically freed if the function return is jumped over by a call to *longjmp*(3) or *siglongjmp*(3).

The space allocated by **alloca**() is *not* automatically deallocated if the pointer that refers to it simply goes out of scope.

Do not attempt to *free*(3) space allocated by **alloca**()!

By necessity, **alloca**() is a compiler built-in, also known as **__builtin_alloca**(). By default, modern compilers automatically translate all uses of **alloca**() into the built-in, but this is forbidden if standards conformance is requested (*−ansi*, *−std=c**), in which case *<alloca.h>* is required, lest a symbol dependency be emitted.

The fact that **alloca**() is a built-in means it is impossible to take its address or to change its behavior by linking with a different library.

Variable length arrays (VLAs) are part of the C99 standard, optional since C11, and can be used for a similar purpose. However, they do not port to standard C++, and, being variables, live in their block scope and don't have an allocator-like interface, making them unfit for implementing functionality like *strdupa*(3).

BUGS

Due to the nature of the stack, it is impossible to check if the allocation would overflow the space available, and, hence, neither is indicating an error. (However, the program is likely to receive a **SIGSEGV** signal if it attempts to access unavailable space.)

On many systems **alloca**() cannot be used inside the list of arguments of a function call, because the stack space reserved by **alloca**() would appear on the stack in the middle of the space for the function arguments.

SEE ALSO

brk(2), *longjmp*(3), *malloc*(3)

NAME

arc4random, arc4random_uniform, arc4random_buf − cryptographically-secure pseudorandom number generator

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

uint32_t arc4random(void);
uint32_t arc4random_uniform(uint32_t *upper_bound***);**
void arc4random_buf(void *buf* **[.***n***], size_t** *n***);**

DESCRIPTION

These functions give cryptographically-secure pseudorandom numbers.

arc4random() returns a uniformly-distributed value.

arc4random_uniform() returns a uniformly-distributed value less than *upper_bound* (see BUGS).

arc4random_buf() fills the memory pointed to by *buf* , with *n* bytes of pseudorandom data.

The *rand*(3) and *drand48*(3) families of functions should only be used where the quality of the pseudo-random numbers is not a concern *and* there's a need for repeatability of the results. Unless you meet both of those conditions, use the **arc4random**() functions.

RETURN VALUE

arc4random() returns a pseudorandom number.

arc4random_uniform() returns a pseudorandom number less than *upper_bound* for valid input, or **0** when *upper_bound* is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
arc4random(), **arc4random_uniform**(), **arc4random_buf**()	Thread safety	MT-Safe

STANDARDS

BSD.

HISTORY

OpenBSD 2.1, FreeBSD 3.0, NetBSD 1.6, DragonFly 1.0, libbsd, glibc 2.36.

BUGS

An *upper_bound* of **0** doesn't make sense in a call to **arc4random_uniform**(). Such a call will fail, and return **0**. Be careful, since that value is *not* less than *upper_bound*. In some cases, such as accessing an array, using that value could result in Undefined Behavior.

SEE ALSO

getrandom(3), *rand*(3), *drand48*(3), *random*(7)

NAME

argz_add, argz_add_sep, argz_append, argz_count, argz_create, argz_create_sep, argz_delete, argz_extract, argz_insert, argz_next, argz_replace, argz_stringify − functions to handle an argz list

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <argz.h>

error_t argz_add(char **restrict *argz***, size_t *restrict** *argz_len***,**
const char *restrict *str***);**

error_t argz_add_sep(char **restrict *argz***, size_t *restrict** *argz_len***,**
const char *restrict *str***, int** *delim***);**

error_t argz_append(char **restrict *argz***, size_t *restrict** *argz_len***,**
const char *restrict *buf***, size_t** *buf_len***);**

size_t argz_count(const char ***_argz_, size_t** *argz_len***);**

error_t argz_create(char *const *argv***[], char **restrict** *argz***,**
size_t *restrict *argz_len***);**

error_t argz_create_sep(const char *restrict *str***, int** *sep***,**
char **restrict *argz***, size_t *restrict** *argz_len***);**

void argz_delete(char **restrict *argz***, size_t *restrict** *argz_len***,**
char *restrict *entry***);**

void argz_extract(const char *restrict *argz***, size_t** *argz_len***,**
char **restrict *argv***);**

error_t argz_insert(char **restrict *argz***, size_t *restrict** *argz_len***,**
char *restrict *before***, const char *restrict** *entry***);**

char *argz_next(const char *restrict *argz***, size_t** *argz_len***,**
const char *restrict *entry***);**

error_t argz_replace(char **restrict *argz***, size_t *restrict** *argz_len***,**
const char *restrict *str***, const char *restrict** *with***,**
unsigned int *restrict *replace_count***);**

void argz_stringify(char ***_argz_, size_t** *len***, int** *sep***);**

DESCRIPTION

These functions are glibc-specific.

An argz vector is a pointer to a character buffer together with a length. The intended interpretation of the character buffer is an array of strings, where the strings are separated by null bytes ('\0'). If the length is nonzero, the last byte of the buffer must be a null byte.

These functions are for handling argz vectors. The pair (NULL,0) is an argz vector, and, conversely, argz vectors of length 0 must have null pointer. Allocation of nonempty argz vectors is done using *malloc*(3), so that *free*(3) can be used to dispose of them again.

argz_add() adds the string *str* at the end of the array **argz*, and updates **argz* and **argz_len*.

argz_add_sep() is similar, but splits the string *str* into substrings separated by the delimiter *delim*. For example, one might use this on a UNIX search path with delimiter ':'.

argz_append() appends the argz vector (*buf*, *buf_len*) after (**argz*, **argz_len*) and updates **argz* and **argz_len*. (Thus, **argz_len* will be increased by *buf_len*.)

argz_count() counts the number of strings, that is, the number of null bytes ('\0'), in (*argz*, *argz_len*).

argz_create() converts a UNIX-style argument vector *argv*, terminated by *(char *) 0*, into an argz vector (**argz*, **argz_len*).

argz_create_sep() converts the null-terminated string *str* into an argz vector (**argz*, **argz_len*) by breaking it up at every occurrence of the separator *sep*.

argz_delete() removes the substring pointed to by *entry* from the argz vector (**argz*, **argz_len*) and

updates **argz* and **argz_len*.

argz_extract() is the opposite of **argz_create**(). It takes the argz vector (*argz*, *argz_len*) and fills the array starting at *argv* with pointers to the substrings, and a final NULL, making a UNIX-style argv vector. The array *argv* must have room for *argz_count*(*argz*, *argz_len*) + 1 pointers.

argz_insert() is the opposite of **argz_delete**(). It inserts the argument *entry* at position *before* into the argz vector (**argz*, **argz_len*) and updates **argz* and **argz_len*. If *before* is NULL, then *entry* will inserted at the end.

argz_next() is a function to step through the argz vector. If *entry* is NULL, the first entry is returned. Otherwise, the entry following is returned. It returns NULL if there is no following entry.

argz_replace() replaces each occurrence of *str* with *with*, reallocating argz as necessary. If *replace_count* is non-NULL, **replace_count* will be incremented by the number of replacements.

argz_stringify() is the opposite of **argz_create_sep**(). It transforms the argz vector into a normal string by replacing all null bytes ('\0') except the last by *sep*.

RETURN VALUE

All argz functions that do memory allocation have a return type of *error_t* (an integer type), and return 0 for success, and **ENOMEM** if an allocation error occurs.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
argz_add(), **argz_add_sep**(), **argz_append**(), **argz_count**(), **argz_create**(), **argz_create_sep**(), **argz_delete**(), **argz_extract**(), **argz_insert**(), **argz_next**(), **argz_replace**(), **argz_stringify**()	Thread safety	MT-Safe

STANDARDS

GNU.

BUGS

Argz vectors without a terminating null byte may lead to Segmentation Faults.

SEE ALSO

envz_add(3)

NAME

asin, asinf, asinl − arc sine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

asinf(), **asinl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the principal value of the arc sine of *x*; that is the value whose sine is *x*.

RETURN VALUE

On success, these functions return the principal value of the arc sine of *x* in radians; the return value is in the range [−pi/2, pi/2].

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is outside the range [−1, 1], a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is outside the range [−1, 1]
errno is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
asin(), **asinf**(), **asinl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

acos(3), *atan*(3), *atan2*(3), *casin*(3), *cos*(3), *sin*(3), *tan*(3)

NAME

asinh, asinhf, asinhl − inverse hyperbolic sine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

asinh():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

asinhf(), **asinhl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the inverse hyperbolic sine of *x*; that is the value whose hyperbolic sine is *x*.

RETURN VALUE

On success, these functions return the inverse hyperbolic sine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity (negative infinity), positive infinity (negative infinity) is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
asinh(), **asinhf**(), **asinhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

acosh(3), *atanh*(3), *casinh*(3), *cosh*(3), *sinh*(3), *tanh*(3)

NAME

asprintf, vasprintf − print to allocated string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <stdio.h>

int asprintf(char **restrict strp, const char *restrict fmt, ...);
int vasprintf(char **restrict strp, const char *restrict fmt,
              va_list ap);
```

DESCRIPTION

The functions **asprintf**() and **vasprintf**() are analogs of *sprintf*(3) and *vsprintf*(3), except that they allocate a string large enough to hold the output including the terminating null byte ('\0'), and return a pointer to it via the first argument. This pointer should be passed to *free*(3) to release the allocated storage when it is no longer needed.

RETURN VALUE

When successful, these functions return the number of bytes printed, just like *sprintf*(3). If memory allocation wasn't possible, or some other error occurs, these functions will return −1, and the contents of *strp* are undefined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
asprintf(), **vasprintf**()	Thread safety	MT-Safe locale

VERSIONS

The FreeBSD implementation sets *strp* to NULL on error.

STANDARDS

GNU, BSD.

SEE ALSO

free(3), *malloc*(3), *printf*(3)

NAME

assert − abort the program if assertion is false

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <assert.h>

void assert(scalar *expression***);**

DESCRIPTION

This macro can help programmers find bugs in their programs, or handle exceptional cases via a crash that will produce limited debugging output.

If *expression* is false (i.e., compares equal to zero), **assert**() prints an error message to standard error and terminates the program by calling *abort*(3). The error message includes the name of the file and function containing the **assert**() call, the source code line number of the call, and the text of the argument; something like:

```
prog: some_file.c:16: some_func: Assertion `val == 0' failed.
```

If the macro **NDEBUG** is defined at the moment <*assert.h*> was last included, the macro **assert**() generates no code, and hence does nothing at all. It is not recommended to define **NDEBUG** if using **assert**() to detect error conditions since the software may behave non-deterministically.

RETURN VALUE

No value is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
assert()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, C99, POSIX.1-2001.

In C89, *expression* is required to be of type *int* and undefined behavior results if it is not, but in C99 it may have any scalar type.

BUGS

assert() is implemented as a macro; if the expression tested has side-effects, program behavior will be different depending on whether **NDEBUG** is defined. This may create Heisenbugs which go away when debugging is turned on.

SEE ALSO

abort(3), *assert_perror*(3), *exit*(3)

NAME

assert_perror − test errnum and abort

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <assert.h>

void assert_perror(int errnum);
```

DESCRIPTION

If the macro **NDEBUG** was defined at the moment <*assert.h*> was last included, the macro **assert_perror**() generates no code, and hence does nothing at all. Otherwise, the macro **assert_perror**() prints an error message to standard error and terminates the program by calling *abort*(3) if *errnum* is nonzero. The message contains the filename, function name and line number of the macro call, and the output of *strerror(errnum)*.

RETURN VALUE

No value is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
assert_perror()	Thread safety	MT-Safe

STANDARDS

GNU.

BUGS

The purpose of the assert macros is to help programmers find bugs in their programs, things that cannot happen unless there was a coding mistake. However, with system or library calls the situation is rather different, and error returns can happen, and will happen, and should be tested for. Not by an assert, where the test goes away when **NDEBUG** is defined, but by proper error handling code. Never use this macro.

SEE ALSO

abort(3), *assert*(3), *exit*(3), *strerror*(3)

NAME

atan, atanf, atanl − arc tangent function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

atanf(), **atanl**():

```
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

These functions calculate the principal value of the arc tangent of *x*; that is the value whose tangent is *x*.

RETURN VALUE

On success, these functions return the principal value of the arc tangent of *x* in radians; the return value is in the range [−pi/2, pi/2].

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity (negative infinity), +pi/2 (−pi/2) is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atan(), **atanf**(), **atanl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

acos(3), *asin*(3), *atan2*(3), *carg*(3), *catan*(3), *cos*(3), *sin*(3), *tan*(3)

NAME

atan2, atan2f, atan2l − arc tangent function of two variables

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

atan2f(), **atan2l**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the principal value of the arc tangent of *y/x*, using the signs of the two arguments to determine the quadrant of the result.

RETURN VALUE

On success, these functions return the principal value of the arc tangent of *y/x* in radians; the return value is in the range [−pi, pi].

If *y* is +0 (−0) and *x* is less than 0, +pi (−pi) is returned.

If *y* is +0 (−0) and *x* is greater than 0, +0 (−0) is returned.

If *y* is less than 0 and *x* is +0 or −0, −pi/2 is returned.

If *y* is greater than 0 and *x* is +0 or −0, pi/2 is returned.

If either *x* or *y* is NaN, a NaN is returned.

If *y* is +0 (−0) and *x* is −0, +pi (−pi) is returned.

If *y* is +0 (−0) and *x* is +0, +0 (−0) is returned.

If *y* is a finite value greater (less) than 0, and *x* is negative infinity, +pi (−pi) is returned.

If *y* is a finite value greater (less) than 0, and *x* is positive infinity, +0 (−0) is returned.

If *y* is positive infinity (negative infinity), and *x* is finite, pi/2 (−pi/2) is returned.

If *y* is positive infinity (negative infinity) and *x* is negative infinity, +3*pi/4 (−3*pi/4) is returned.

If *y* is positive infinity (negative infinity) and *x* is positive infinity, +pi/4 (−pi/4) is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atan2(), **atan2f**(), **atan2l**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

acos(3), *asin*(3), *atan*(3), *carg*(3), *cos*(3), *sin*(3), *tan*(3)

NAME

atanh, atanhf, atanhl − inverse hyperbolic tangent function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

atanh():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

atanhf(), **atanhl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions calculate the inverse hyperbolic tangent of *x*; that is the value whose hyperbolic tangent is *x*.

RETURN VALUE

On success, these functions return the inverse hyperbolic tangent of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is +1 or −1, a pole error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the mathematically correct sign.

If the absolute value of *x* is greater than 1, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* less than −1 or greater than +1
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

Pole error: *x* is +1 or −1
: *errno* is set to **ERANGE** (but see BUGS). A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atanh(), **atanhf**(), **atanhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

BUGS

In glibc 2.9 and earlier, when a pole error occurs, *errno* is set to **EDOM** instead of the POSIX-mandated **ERANGE**. Since glibc 2.10, glibc does the right thing.

SEE ALSO

acosh(3), *asinh*(3), *catanh*(3), *cosh*(3), *sinh*(3), *tanh*(3)

NAME

atexit − register a function to be called at normal process termination

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int atexit(void (* *function***)(void));**

DESCRIPTION

The **atexit**() function registers the given *function* to be called at normal process termination, either via *exit*(3) or via return from the program's *main*(). Functions so registered are called in the reverse order of their registration; no arguments are passed.

The same function may be registered multiple times: it is called once for each registration.

POSIX.1 requires that an implementation allow at least **ATEXIT_MAX** (32) such functions to be registered. The actual limit supported by an implementation can be obtained using *sysconf*(3).

When a child process is created via *fork*(2), it inherits copies of its parent's registrations. Upon a successful call to one of the *exec*(3) functions, all registrations are removed.

RETURN VALUE

The **atexit**() function returns the value 0 if successful; otherwise it returns a nonzero value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atexit()	Thread safety	MT-Safe

VERSIONS

POSIX.1 says that the result of calling *exit*(3) more than once (i.e., calling *exit*(3) within a function registered using *atexit*()) is undefined. On some systems (but not Linux), this can result in an infinite recursion; portable programs should not invoke *exit*(3) inside a function registered using **atexit**().

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

NOTES

Functions registered using **atexit**() (and *on_exit*(3)) are not called if a process terminates abnormally because of the delivery of a signal.

If one of the registered functions calls *_exit*(2), then any remaining functions are not invoked, and the other process termination steps performed by *exit*(3) are not performed.

The **atexit**() and *on_exit*(3) functions register functions on the same list: at normal process termination, the registered functions are invoked in reverse order of their registration by these two functions.

According to POSIX.1, the result is undefined if *longjmp*(3) is used to terminate execution of one of the functions registered using **atexit**().

Linux notes

Since glibc 2.2.3, **atexit**() (and *on_exit*(3)) can be used within a shared library to establish functions that are called when the shared library is unloaded.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
bye(void)
{
    printf("That was all, folks\n");
```

```
}

int
main(void)
{
    long a;
    int i;

    a = sysconf(_SC_ATEXIT_MAX);
    printf("ATEXIT_MAX = %ld\n", a);

    i = atexit(bye);
    if (i != 0) {
        fprintf(stderr, "cannot set exit function\n");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

_exit(2), *dlopen*(3), *exit*(3), *on_exit*(3)

NAME

atof − convert a string to a double

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

double atof(const char ****nptr*);**

DESCRIPTION

The **atof**() function converts the initial portion of the string pointed to by *nptr* to *double*. The behavior is the same as

```
strtod(nptr, NULL);
```

except that **atof**() does not detect errors.

RETURN VALUE

The converted value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atof()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

SEE ALSO

atoi(3), *atol*(3), *strfromd*(3), *strtod*(3), *strtol*(3), *strtoul*(3)

NAME

atoi, atol, atoll − convert a string to an integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

atoll():
_ISOC99_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **atoi**() function converts the initial portion of the string pointed to by *nptr* to *int*. The behavior is the same as

```
strtol(nptr, NULL, 10);
```

except that **atoi**() does not detect errors.

The **atol**() and **atoll**() functions behave the same as **atoi**(), except that they convert the initial portion of the string to their return type of *long* or *long long*.

RETURN VALUE

The converted value or 0 on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
atoi(), **atol**(), **atoll**()	Thread safety	MT-Safe locale

VERSIONS

POSIX.1 leaves the return value of **atoi**() on error unspecified. On glibc, musl libc, and uClibc, 0 is returned on error.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001, SVr4, 4.3BSD.

C89 and POSIX.1-1996 include the functions **atoi**() and **atol**() only.

BUGS

errno is not set on error so there is no way to distinguish between 0 as an error and as the converted value. No checks for overflow or underflow are done. Only base-10 input can be converted. It is recommended to instead use the **strtol**() and **strtoul**() family of functions in new programs.

SEE ALSO

atof(3), *strtod*(3), *strtol*(3), *strtoul*(3)

NAME

backtrace, backtrace_symbols, backtrace_symbols_fd − support for application self-debugging

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <execinfo.h>

int backtrace(void *buffer[.size], int size);

char **backtrace_symbols(void *const buffer[.size], int size);
void backtrace_symbols_fd(void *const buffer[.size], int size, int  fd);
```

DESCRIPTION

backtrace() returns a backtrace for the calling program, in the array pointed to by *buffer*. A backtrace is the series of currently active function calls for the program. Each item in the array pointed to by *buffer* is of type *void **, and is the return address from the corresponding stack frame. The *size* argument specifies the maximum number of addresses that can be stored in *buffer*. If the backtrace is larger than *size*, then the addresses corresponding to the *size* most recent function calls are returned; to obtain the complete backtrace, make sure that *buffer* and *size* are large enough.

Given the set of addresses returned by **backtrace**() in *buffer*, **backtrace_symbols**() translates the addresses into an array of strings that describe the addresses symbolically. The *size* argument specifies the number of addresses in *buffer*. The symbolic representation of each address consists of the function name (if this can be determined), a hexadecimal offset into the function, and the actual return address (in hexadecimal). The address of the array of string pointers is returned as the function result of **backtrace_symbols**(). This array is *malloc*(3)ed by **backtrace_symbols**(), and must be freed by the caller. (The strings pointed to by the array of pointers need not and should not be freed.)

backtrace_symbols_fd() takes the same *buffer* and *size* arguments as **backtrace_symbols**(), but instead of returning an array of strings to the caller, it writes the strings, one per line, to the file descriptor *fd*. **backtrace_symbols_fd**() does not call *malloc*(3), and so can be employed in situations where the latter function might fail, but see NOTES.

RETURN VALUE

backtrace() returns the number of addresses returned in *buffer*, which is not greater than *size*. If the return value is less than *size*, then the full backtrace was stored; if it is equal to *size*, then it may have been truncated, in which case the addresses of the oldest stack frames are not returned.

On success, **backtrace_symbols**() returns a pointer to the array *malloc*(3)ed by the call; on error, NULL is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
backtrace(), **backtrace_symbols**(), **backtrace_symbols_fd**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

NOTES

These functions make some assumptions about how a function's return address is stored on the stack. Note the following:

- Omission of the frame pointers (as implied by any of *gcc*(1)nonzero optimization levels) may cause these assumptions to be violated.
- Inlined functions do not have stack frames.
- Tail-call optimization causes one stack frame to replace another.
- **backtrace**() and **backtrace_symbols_fd**() don't call **malloc**() explicitly, but they are part of *libgcc*, which gets loaded dynamically when first used. Dynamic loading usually triggers a call to *malloc*(3). If you need certain calls to these two functions to not allocate memory (in signal handlers, for example), you need to make sure *libgcc* is loaded beforehand.

The symbol names may be unavailable without the use of special linker options. For systems using the GNU linker, it is necessary to use the −*rdynamic* linker option. Note that names of "static" functions are not exposed, and won't be available in the backtrace.

EXAMPLES

The program below demonstrates the use of **backtrace**() and **backtrace_symbols**(). The following shell session shows what we might see when running the program:

```
$ cc -rdynamic prog.c -o prog
$ ./prog 3
backtrace() returned 8 addresses
./prog(myfunc3+0x5c) [0x80487f0]
./prog [0x8048871]
./prog(myfunc+0x21) [0x8048894]
./prog(myfunc+0x1a) [0x804888d]
./prog(myfunc+0x1a) [0x804888d]
./prog(main+0x65) [0x80488fb]
/lib/libc.so.6(__libc_start_main+0xdc) [0xb7e38f9c]
./prog [0x8048711]
```

Program source

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BT_BUF_SIZE 100

void
myfunc3(void)
{
    int nptrs;
    void *buffer[BT_BUF_SIZE];
    char **strings;

    nptrs = backtrace(buffer, BT_BUF_SIZE);
    printf("backtrace() returned %d addresses\n", nptrs);

    /* The call backtrace_symbols_fd(buffer, nptrs, STDOUT_FILENO)
       would produce similar output to the following: */

    strings = backtrace_symbols(buffer, nptrs);
    if (strings == NULL) {
        perror("backtrace_symbols");
        exit(EXIT_FAILURE);
    }

    for (size_t j = 0; j < nptrs; j++)
        printf("%s\n", strings[j]);

    free(strings);
}

static void   /* "static" means don't export the symbol... */
myfunc2(void)
{
    myfunc3();
}
```

```
void
myfunc(int ncalls)
{
    if (ncalls > 1)
        myfunc(ncalls - 1);
    else
        myfunc2();
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "%s num-calls\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    myfunc(atoi(argv[1]));
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

addr2line(1), *gcc*(1), *gdb*(1), *ld*(1), *dlopen*(3), *malloc*(3)

NAME

basename, dirname − parse pathname components

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <libgen.h>

char *dirname(char *path);
char *basename(char *path);
```

DESCRIPTION

Warning: there are two different functions **basename**(); see below.

The functions **dirname**() and **basename**() break a null-terminated pathname string into directory and filename components. In the usual case, **dirname**() returns the string up to, but not including, the final '/', and **basename**() returns the component following the final '/'. Trailing '/' characters are not counted as part of the pathname.

If *path* does not contain a slash, **dirname**() returns the string "." while **basename**() returns a copy of *path*. If *path* is the string "/", then both **dirname**() and **basename**() return the string "/". If *path* is a null pointer or points to an empty string, then both **dirname**() and **basename**() return the string ".".

Concatenating the string returned by **dirname**(), a "/", and the string returned by **basename**() yields a complete pathname.

Both **dirname**() and **basename**() may modify the contents of *path*, so it may be desirable to pass a copy when calling one of these functions.

These functions may return pointers to statically allocated memory which may be overwritten by subsequent calls. Alternatively, they may return a pointer to some part of *path*, so that the string referred to by *path* should not be modified or freed until the pointer returned by the function is no longer required.

The following list of examples (taken from SUSv2) shows the strings returned by **dirname**() and **basename**() for different paths:

path	**dirname**	**basename**
/usr/lib	/usr	lib
/usr/	/	usr
usr	.	usr
/	/	/
.	.	.
..	.	..

RETURN VALUE

Both **dirname**() and **basename**() return pointers to null-terminated strings. (Do not pass these pointers to *free*(3).)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	**Attribute**	**Value**
basename(), **dirname**()	Thread safety	MT-Safe

VERSIONS

There are two different versions of **basename**() - the POSIX version described above, and the GNU version, which one gets after

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>
```

The GNU version never modifies its argument, and returns the empty string when *path* has a trailing slash, and in particular also when it is "/". There is no GNU version of **dirname**().

With glibc, one gets the POSIX version of **basename**() when <*libgen.h*> is included, and the GNU version otherwise.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

BUGS

In the glibc implementation, the POSIX versions of these functions modify the *path* argument, and segfault when called with a static string such as "/usr/".

Before glibc 2.2.1, the glibc version of **dirname**() did not correctly handle pathnames with trailing '/' characters, and generated a segfault if given a NULL argument.

EXAMPLES

The following code snippet demonstrates the use of **basename**() and **dirname**():

```
char *dirc, *basec, *bname, *dname;
char *path = "/etc/passwd";

dirc = strdup(path);
basec = strdup(path);
dname = dirname(dirc);
bname = basename(basec);
printf("dirname=%s, basename=%s\n", dname, bname);
```

SEE ALSO

basename(1), *dirname*(1)

NAME

bcmp − compare byte sequences

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <strings.h>

[[deprecated]] int bcmp(const void *s1***[.**n**], const void** *s2***[.**n**], size_t** *n***);**

DESCRIPTION

bcmp() is identical to *memcmp*(3); use the latter instead.

STANDARDS

None.

HISTORY

4.3BSD. Marked as LEGACY in POSIX.1-2001; removed in POSIX.1-2008.

SEE ALSO

memcmp(3)

NAME

bcopy − copy byte sequence

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <strings.h>

[[deprecated]] void bcopy(const void *src***[.***n***], void** *dest***[.***n***], size_t** *n***);**

DESCRIPTION

The **bcopy**() function copies *n* bytes from *src* to *dest*. The result is correct, even when both areas overlap.

RETURN VALUE

None.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
bcopy()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD.

Marked as LEGACY in POSIX.1-2001: use *memcpy*(3) or *memmove*(3) in new programs. Note that the first two arguments are interchanged for *memcpy*(3) and *memmove*(3). POSIX.1-2008 removes the specification of **bcopy**().

SEE ALSO

bstring(3), *memccpy*(3), *memcpy*(3), *memmove*(3), *strcpy*(3), *strncpy*(3)

NAME

bindresvport − bind a socket to a privileged IP port

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport(int sockfd, struct sockaddr_in *sin);
```

DESCRIPTION

bindresvport() is used to bind the socket referred to by the file descriptor *sockfd* to a privileged anonymous IP port, that is, a port number arbitrarily selected from the range 512 to 1023.

If the *bind*(2) performed by **bindresvport**() is successful, and *sin* is not NULL, then *sin−>sin_port* returns the port number actually allocated.

sin can be NULL, in which case *sin−>sin_family* is implicitly taken to be **AF_INET**. However, in this case, **bindresvport**() has no way to return the port number actually allocated. (This information can later be obtained using *getsockname*(2).)

RETURN VALUE

bindresvport() returns 0 on success; otherwise −1 is returned and *errno* is set to indicate the error.

ERRORS

bindresvport() can fail for any of the same reasons as *bind*(2). In addition, the following errors may occur:

EACCES
: The calling process was not privileged (on Linux: the calling process did not have the **CAP_NET_BIND_SERVICE** capability in the user namespace governing its network namespace).

EADDRINUSE
: All privileged ports are in use.

EAFNOSUPPORT (**EPFNOSUPPORT** in glibc 2.7 and earlier)
: *sin* is not NULL and *sin−>sin_family* is not **AF_INET**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
bindresvport()	Thread safety	glibc >= 2.17: MT-Safe; glibc < 2.17: MT-Unsafe

The **bindresvport**() function uses a static variable that was not protected by a lock before glibc 2.17, rendering the function MT-Unsafe.

VERSIONS

Present on the BSDs, Solaris, and many other systems.

NOTES

Unlike some **bindresvport**() implementations, the glibc implementation ignores any value that the caller supplies in *sin−>sin_port*.

STANDARDS

BSD.

SEE ALSO

bind(2), *getsockname*(2)

NAME

bsd_signal − signal handling with BSD semantics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t bsd_signal(int *signum***, sighandler_t** *handler***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

bsd_signal():
Since glibc 2.26:
_XOPEN_SOURCE >= 500
&& ! (_POSIX_C_SOURCE >= 200809L)
glibc 2.25 and earlier:
_XOPEN_SOURCE

DESCRIPTION

The **bsd_signal**() function takes the same arguments, and performs the same task, as *signal*(2).

The difference between the two is that **bsd_signal**() is guaranteed to provide reliable signal semantics, that is: a) the disposition of the signal is not reset to the default when the handler is invoked; b) delivery of further instances of the signal is blocked while the signal handler is executing; and c) if the handler interrupts a blocking system call, then the system call is automatically restarted. A portable application cannot rely on *signal*(2) to provide these guarantees.

RETURN VALUE

The **bsd_signal**() function returns the previous value of the signal handler, or **SIG_ERR** on error.

ERRORS

As for *signal*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
bsd_signal()	Thread safety	MT-Safe

VERSIONS

Use of **bsd_signal**() should be avoided; use *sigaction*(2) instead.

On modern Linux systems, **bsd_signal**() and *signal*(2) are equivalent. But on older systems, *signal*(2) provided unreliable signal semantics; see *signal*(2) for details.

The use of *sighandler_t* is a GNU extension; this type is defined only if the **_GNU_SOURCE** feature test macro is defined.

STANDARDS

None.

HISTORY

4.2BSD, POSIX.1-2001. Removed in POSIX.1-2008, recommending the use of *sigaction*(2) instead.

SEE ALSO

sigaction(2), *signal*(2), *sysv_signal*(3), *signal*(7)

NAME

bsearch − binary search of a sorted array

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void key[.size], const void base[.size * .nmemb],
              size_t nmemb, size_t size,
              int (*compar)(const void [.size], const void [.size]));
```

DESCRIPTION

The **bsearch**() function searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array should be in ascending sorted order according to the comparison function referenced by *compar*. The *compar* routine is expected to have two arguments which point to the *key* object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the *key* object is found, respectively, to be less than, to match, or be greater than the array member.

RETURN VALUE

The **bsearch**() function returns a pointer to a matching member of the array, or NULL if no match is found. If there are multiple elements that match the key, the element returned is unspecified.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
bsearch()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

EXAMPLES

The example below first sorts an array of structures using *qsort*(3), then retrieves desired elements using **bsearch**().

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(arr)  (sizeof((arr)) / sizeof((arr)[0]))

struct mi {
    int         nr;
    const char  *name;
};

static struct mi  months[] = {
    { 1, "jan" }, { 2, "feb" }, { 3, "mar" }, { 4, "apr" },
    { 5, "may" }, { 6, "jun" }, { 7, "jul" }, { 8, "aug" },
    { 9, "sep" }, {10, "oct" }, {11, "nov" }, {12, "dec" }
};

static int
compmi(const void *m1, const void *m2)
{
    const struct mi *mi1 = m1;
    const struct mi *mi2 = m2;
```

```
    return strcmp(mi1->name, mi2->name);
}

int
main(int argc, char *argv[])
{
    qsort(months, ARRAY_SIZE(months), sizeof(months[0]), compmi);
    for (size_t i = 1; i < argc; i++) {
        struct mi key;
        struct mi *res;

        key.name = argv[i];
        res = bsearch(&key, months, ARRAY_SIZE(months),
                      sizeof(months[0]), compmi);
        if (res == NULL)
            printf("'%s': unknown month\n", argv[i]);
        else
            printf("%s: month #%d\n", res->name, res->nr);
    }
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

hsearch(3), *lsearch*(3), *qsort*(3), *tsearch*(3)

NAME

bcmp, bcopy, bzero, memccpy, memchr, memcmp, memcpy, memfrob, memmem, memmove, memset − byte string operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

int bcmp(const void s1[.n], const void s2[.n], size_t n);

void bcopy(const void src[.n], void dest[.n], size_t n);

void bzero(void s[.n], size_t n);

void *memccpy(void dest[.n], const void src[.n], int c, size_t n);

void *memchr(const void s[.n], int c, size_t n);

int memcmp(const void s1[.n], const void s2[.n], size_t n);

void *memcpy(void dest[.n], const void src[.n], size_t n);

void *memfrob(void s[.n], size_t n);

void *memmem(const void haystack[.haystacklen], size_t haystacklen,
       const void needle[.needlelen], size_t needlelen);

void *memmove(void dest[.n], const void src[.n], size_t n);

void *memset(void s[.n], int c, size_t n);
```

DESCRIPTION

The byte string functions perform operations on strings (byte arrays) that are not necessarily null-terminated. See the individual man pages for descriptions of each function.

NOTES

The functions **bcmp**() and **bcopy**() are obsolete. Use **memcmp**() and **memmove**() instead.

SEE ALSO

bcmp(3), *bcopy*(3), *bzero*(3), *memccpy*(3), *memchr*(3), *memcmp*(3), *memcpy*(3), *memfrob*(3), *memmem*(3), *memmove*(3), *memset*(3), *string*(3)

NAME

bswap_16, bswap_32, bswap_64 − reverse order of bytes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <byteswap.h>

uint16_t bswap_16(uint16_t x);
uint32_t bswap_32(uint32_t x);
uint64_t bswap_64(uint64_t x);
```

DESCRIPTION

These functions return a value in which the order of the bytes in their 2-, 4-, or 8-byte arguments is reversed.

RETURN VALUE

These functions return the value of their argument with the bytes reversed.

ERRORS

These functions always succeed.

STANDARDS

GNU.

EXAMPLES

The program below swaps the bytes of the 8-byte integer supplied as its command-line argument. The following shell session demonstrates the use of the program:

```
$ ./a.out 0x0123456789abcdef
0x123456789abcdef ==> 0xefcdab8967452301
```

Program source

```
#include <byteswap.h>
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    uint64_t x;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <num>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    x = strtoull(argv[1], NULL, 0);
    printf("%#" PRIx64 " ==> %#" PRIx64 "\n", x, bswap_64(x));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

byteorder(3), *endian*(3)

NAME

btowc − convert single byte to wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wint_t btowc(int *c***);**

DESCRIPTION

The **btowc**() function converts *c*, interpreted as a multibyte sequence of length 1, starting in the initial shift state, to a wide character and returns it. If *c* is **EOF** or not a valid multibyte sequence of length 1, the **btowc**() function returns **WEOF**.

RETURN VALUE

The **btowc**() function returns the wide character converted from the single byte *c*. If *c* is **EOF** or not a valid multibyte sequence of length 1, it returns **WEOF**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
btowc()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The behavior of **btowc**() depends on the **LC_CTYPE** category of the current locale.

This function should never be used. It does not work for encodings which have state, and unnecessarily treats single bytes differently from multibyte sequences. Use either *mbtowc*(3) or the thread-safe *mbrtowc*(3) instead.

SEE ALSO

mbrtowc(3), *mbtowc*(3), *wctob*(3)

NAME

btree − btree database access method

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <db.h>

DESCRIPTION

Note well: This page documents interfaces provided up until glibc 2.1. Since glibc 2.2, glibc no longer provides these interfaces. Probably, you are looking for the APIs provided by the *libdb* library instead.

The routine *dbopen*(3) is the library interface to database files. One of the supported file formats is btree files. The general description of the database access methods is in *dbopen*(3), this manual page describes only the btree-specific information.

The btree data structure is a sorted, balanced tree structure storing associated key/data pairs.

The btree access-method-specific data structure provided to *dbopen*(3) is defined in the <*db.h*> include file as follows:

```
typedef struct {
    unsigned long flags;
    unsigned int  cachesize;
    int           maxkeypage;
    int           minkeypage;
    unsigned int  psize;
    int         (*compare)(const DBT *key1, const DBT *key2);
    size_t      (*prefix)(const DBT *key1, const DBT *key2);
    int           lorder;
} BTREEINFO;
```

The elements of this structure are as follows:

flags The flag value is specified by ORing any of the following values:

R_DUP

Permit duplicate keys in the tree, that is, permit insertion if the key to be inserted already exists in the tree. The default behavior, as described in *dbopen*(3), is to overwrite a matching key when inserting a new key or to fail if the **R_NOOVERWRITE** flag is specified. The **R_DUP** flag is overridden by the **R_NOOVERWRITE** flag, and if the **R_NOOVERWRITE** flag is specified, attempts to insert duplicate keys into the tree will fail.

If the database contains duplicate keys, the order of retrieval of key/data pairs is undefined if the *get* routine is used, however, *seq* routine calls with the **R_CURSOR** flag set will always return the logical "first" of any group of duplicate keys.

cachesize

A suggested maximum size (in bytes) of the memory cache. This value is *only* advisory, and the access method will allocate more memory rather than fail. Since every search examines the root page of the tree, caching the most recently used pages substantially improves access time. In addition, physical writes are delayed as long as possible, so a moderate cache can reduce the number of I/O operations significantly. Obviously, using a cache increases (but only increases) the likelihood of corruption or lost data if the system crashes while a tree is being modified. If *cachesize* is 0 (no size is specified), a default cache is used.

maxkeypage

The maximum number of keys which will be stored on any single page. Not currently implemented.

minkeypage

The minimum number of keys which will be stored on any single page. This value is used to determine which keys will be stored on overflow pages, that is, if a key or data item is longer than the pagesize divided by the minkeypage value, it will be stored on overflow pages instead

of in the page itself. If *minkeypage* is 0 (no minimum number of keys is specified), a value of 2 is used.

psize Page size is the size (in bytes) of the pages used for nodes in the tree. The minimum page size is 512 bytes and the maximum page size is 64 KiB. If *psize* is 0 (no page size is specified), a page size is chosen based on the underlying filesystem I/O block size.

compare
Compare is the key comparison function. It must return an integer less than, equal to, or greater than zero if the first key argument is considered to be respectively less than, equal to, or greater than the second key argument. The same comparison function must be used on a given tree every time it is opened. If *compare* is NULL (no comparison function is specified), the keys are compared lexically, with shorter keys considered less than longer keys.

prefix Prefix is the prefix comparison function. If specified, this routine must return the number of bytes of the second key argument which are necessary to determine that it is greater than the first key argument. If the keys are equal, the key length should be returned. Note, the usefulness of this routine is very data-dependent, but, in some data sets can produce significantly reduced tree sizes and search times. If *prefix* is NULL (no prefix function is specified), *and* no comparison function is specified, a default lexical comparison routine is used. If *prefix* is NULL and a comparison routine is specified, no prefix comparison is done.

lorder The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified), the current host order is used.

If the file already exists (and the **O_TRUNC** flag is not specified), the values specified for the arguments *flags*, *lorder*, and *psize* are ignored in favor of the values used when the tree was created.

Forward sequential scans of a tree are from the least key to the greatest.

Space freed up by deleting key/data pairs from the tree is never reclaimed, although it is normally made available for reuse. This means that the btree storage structure is grow-only. The only solutions are to avoid excessive deletions, or to create a fresh tree periodically from a scan of an existing one.

Searches, insertions, and deletions in a btree will all complete in O lg base N where base is the average fill factor. Often, inserting ordered data into btrees results in a low fill factor. This implementation has been modified to make ordered insertion the best case, resulting in a much better than normal page fill factor.

ERRORS

The *btree* access method routines may fail and set *errno* for any of the errors specified for the library routine *dbopen*(3).

BUGS

Only big and little endian byte order is supported.

SEE ALSO

dbopen(3), *hash*(3), *mpool*(3), *recno*(3)

The Ubiquitous B-tree, Douglas Comer, ACM Comput. Surv. 11, 2 (June 1979), 121-138.

Prefix B-trees, Bayer and Unterauer, ACM Transactions on Database Systems, Vol. 2, 1 (March 1977), 11-26.

The Art of Computer Programming Vol. 3: Sorting and Searching, D.E. Knuth, 1968, pp 471-480.

NAME

htonl, htons, ntohl, ntohs − convert values between host and network byte order

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

The **htonl**() function converts the unsigned integer *hostlong* from host byte order to network byte order.

The **htons**() function converts the unsigned short integer *hostshort* from host byte order to network byte order.

The **ntohl**() function converts the unsigned integer *netlong* from network byte order to host byte order.

The **ntohs**() function converts the unsigned short integer *netshort* from network byte order to host byte order.

On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
htonl(), **htons**(), **ntohl**(), **ntohs**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

bswap(3), *endian*(3), *gethostbyname*(3), *getservent*(3)

NAME

bzero, explicit_bzero − zero a byte string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <strings.h>

void bzero(void *s***[.***n***], size_t** *n***);**

#include <string.h>

void explicit_bzero(void *s***[.***n***], size_t** *n***);**

DESCRIPTION

The **bzero**() function erases the data in the *n* bytes of the memory starting at the location pointed to by *s*, by writing zeros (bytes containing '\0') to that area.

The **explicit_bzero**() function performs the same task as **bzero**(). It differs from **bzero**() in that it guarantees that compiler optimizations will not remove the erase operation if the compiler deduces that the operation is "unnecessary".

RETURN VALUE

None.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
bzero(), **explicit_bzero**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

explicit_bzero()

glibc 2.25.

The **explicit_bzero**() function is a nonstandard extension that is also present on some of the BSDs. Some other implementations have a similar function, such as **memset_explicit**() or **memset_s**().

bzero() 4.3BSD.

Marked as LEGACY in POSIX.1-2001. Removed in POSIX.1-2008.

NOTES

The **explicit_bzero**() function addresses a problem that security-conscious applications may run into when using **bzero**(): if the compiler can deduce that the location to be zeroed will never again be touched by a *correct* program, then it may remove the **bzero**() call altogether. This is a problem if the intent of the **bzero**() call was to erase sensitive data (e.g., passwords) to prevent the possibility that the data was leaked by an incorrect or compromised program. Calls to **explicit_bzero**() are never optimized away by the compiler.

The **explicit_bzero**() function does not solve all problems associated with erasing sensitive data:

- The **explicit_bzero**() function does *not* guarantee that sensitive data is completely erased from memory. (The same is true of *bzero*().) For example, there may be copies of the sensitive data in a register and in "scratch" stack areas. The **explicit_bzero**() function is not aware of these copies, and can't erase them.
- In some circumstances, **explicit_bzero**() can *decrease* security. If the compiler determined that the variable containing the sensitive data could be optimized to be stored in a register (because it is small enough to fit in a register, and no operation other than the **explicit_bzero**() call would need to take the address of the variable), then the **explicit_bzero**() call will force the data to be copied from the register to a location in RAM that is then immediately erased (while the copy in the register remains unaffected). The problem here is that data in RAM is more likely to be exposed by a bug than data in a register, and thus the **explicit_bzero**() call creates a brief time window where the sensitive data is more vulnerable than it would otherwise have been if no attempt had been made to

erase the data.

Note that declaring the sensitive variable with the **volatile** qualifier does *not* eliminate the above problems. Indeed, it will make them worse, since, for example, it may force a variable that would otherwise have been optimized into a register to instead be maintained in (more vulnerable) RAM for its entire lifetime.

Notwithstanding the above details, for security-conscious applications, using **explicit_bzero**() is generally preferable to not using it. The developers of **explicit_bzero**() anticipate that future compilers will recognize calls to **explicit_bzero**() and take steps to ensure that all copies of the sensitive data are erased, including copies in registers or in "scratch" stack areas.

SEE ALSO

bstring(3), *memset*(3), *swab*(3)

NAME

cabs, cabsf, cabsl − absolute value of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double cabs(double complex *z***);**
float cabsf(float complex *z***);**
long double cabsl(long double complex *z***);**

DESCRIPTION

These functions return the absolute value of the complex number *z*. The result is a real number.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cabs(), **cabsf**(), **cabsl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

NOTES

The function is actually an alias for *hypot(a, b)* (or, equivalently, *sqrt(a*a + b*b)*).

SEE ALSO

abs(3), *cimag*(3), *hypot*(3), *complex*(7)

NAME

cacos, cacosf, cacosl − complex arc cosine

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex cacos(double complex *z***);**
float complex cacosf(float complex *z***);**
long double complex cacosl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex arc cosine of *z*. If *y* = *cacos(z)*, then *z* = *ccos(y)*. The real part of *y* is chosen in the interval [0,pi].

One has:

cacos(z) = −i * clog(z + i * csqrt(1 − z * z))

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cacos(), **cacosf**(), **cacosl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

EXAMPLES

```
/* Link with "-lm" */

#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    double complex z, c, f;
    double complex i = I;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <real> <imag>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    z = atof(argv[1]) + atof(argv[2]) * I;

    c = cacos(z);

    printf("cacos() = %6.3f %6.3f*i\n", creal(c), cimag(c));

    f = -i * clog(z + i * csqrt(1 - z * z));

    printf("formula = %6.3f %6.3f*i\n", creal(f), cimag(f));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

ccos(3), *clog*(3), *complex*(7)

NAME

cacosh, cacoshf, cacoshl − complex arc hyperbolic cosine

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex cacosh(double complex *z***);**
float complex cacoshf(float complex *z***);**
long double complex cacoshl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex arc hyperbolic cosine of *z*. If *y* = *cacosh(z)*, then *z* = *ccosh(y)*. The imaginary part of *y* is chosen in the interval [−pi,pi]. The real part of *y* is chosen nonnegative.

One has:

cacosh(z) = 2 * clog(csqrt((z + 1) / 2) + csqrt((z − 1) / 2))

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cacosh(), **cacoshf**(), **cacoshl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001. glibc 2.1.

EXAMPLES

```
/* Link with "-lm" */

#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    double complex z, c, f;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <real> <imag>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    z = atof(argv[1]) + atof(argv[2]) * I;

    c = cacosh(z);
    printf("cacosh() = %6.3f %6.3f*i\n", creal(c), cimag(c));

    f = 2 * clog(csqrt((z + 1)/2) + csqrt((z - 1)/2));
    printf("formula  = %6.3f %6.3f*i\n", creal(f), cimag(f));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

acosh(3), *cabs*(3), *ccosh*(3), *cimag*(3), *complex*(7)

NAME

canonicalize_file_name − return the canonicalized absolute pathname

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <stdlib.h>

char *canonicalize_file_name(const char *path);
```

DESCRIPTION

The **canonicalize_file_name**() function returns a null-terminated string containing the canonicalized absolute pathname corresponding to *path*. In the returned string, symbolic links are resolved, as are . and .. pathname components. Consecutive slash (/) characters are replaced by a single slash.

The returned string is dynamically allocated by **canonicalize_file_name**() and the caller should deallocate it with *free*(3) when it is no longer required.

The call *canonicalize_file_name(path)* is equivalent to the call:

```
realpath(path, NULL);
```

RETURN VALUE

On success, **canonicalize_file_name**() returns a null-terminated string. On error (e.g., a pathname component is unreadable or does not exist), **canonicalize_file_name**() returns NULL and sets *errno* to indicate the error.

ERRORS

See *realpath*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
canonicalize_file_name()	Thread safety	MT-Safe

STANDARDS

GNU.

SEE ALSO

readlink(2), *realpath*(3)

NAME

carg, cargf, cargl − calculate the complex argument

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double carg(double complex *z***);**
float cargf(float complex *z***);**
long double cargl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

A complex number can be described by two real coordinates. One may use rectangular coordinates and gets

```
z = x + I * y
```

where *x = creal(z)* and *y = cimag(z)*.

Or one may use polar coordinates and gets

```
z = r * cexp(I * a)
```

where *r = cabs(z)* is the "radius", the "modulus", the absolute value of *z*, and *a = carg(z)* is the "phase angle", the argument of *z*.

One has:

```
tan(carg(z)) = cimag(z) / creal(z)
```

RETURN VALUE

The return value is in the range of [−pi,pi].

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
carg(), **cargf**(), **cargl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *complex*(7)

NAME

casin, casinf, casinl − complex arc sine

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex casin(double complex *z***);**
float complex casinf(float complex *z***);**
long double complex casinl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex arc sine of *z*. If *y* = *casin(z)*, then *z* = *csin(y)*. The real part of *y* is chosen in the interval [−pi/2,pi/2].

One has:

casin(z) = −i clog(iz + csqrt(1 − z * z))

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
casin(), **casinf**(), **casinl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

clog(3), *csin*(3), *complex*(7)

NAME

casinh, casinhf, casinhl − complex arc sine hyperbolic

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex casinh(double complex *z***);**
float complex casinhf(float complex *z***);**
long double complex casinhl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex arc hyperbolic sine of *z*. If *y = casinh(z)*, then *z = csinh(y)*. The imaginary part of *y* is chosen in the interval [−pi/2,pi/2].

One has:

```
casinh(z) = clog(z + csqrt(z * z + 1))
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
casinh(), **casinhf**(), **casinhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

asinh(3), *cabs*(3), *cimag*(3), *csinh*(3), *complex*(7)

NAME

catan, catanf, catanl − complex arc tangents

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex catan(double complex *z***);**
float complex catanf(float complex *z***);**
long double complex catanl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex arc tangent of *z*. If *y* = *catan(z)*, then *z* = *ctan(y)*. The real part of *y* is chosen in the interval [−pi/2, pi/2].

One has:

```
catan(z) = (clog(1 + i * z) − clog(1 − i * z)) / (2 * i)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
catan(), **catanf**(), **catanl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

EXAMPLES

```
/* Link with "−lm" */

#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    double complex z, c, f;
    double complex i = I;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <real> <imag>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    z = atof(argv[1]) + atof(argv[2]) * I;

    c = catan(z);
    printf("catan() = %6.3f %6.3f*i\n", creal(c), cimag(c));

    f = (clog(1 + i * z) − clog(1 − i * z)) / (2 * i);
    printf("formula = %6.3f %6.3f*i\n", creal(f), cimag(f));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

ccos(3), *clog*(3), *ctan*(3), *complex*(7)

NAME

catanh, catanhf, catanhl − complex arc tangents hyperbolic

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

DESCRIPTION

These functions calculate the complex arc hyperbolic tangent of *z*. If *y = catanh(z)*, then *z = ctanh(y)*. The imaginary part of *y* is chosen in the interval [−pi/2,pi/2].

One has:

```
catanh(z) = 0.5 * (clog(1 + z) − clog(1 − z))
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
catanh(), **catanhf**(), **catanhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

EXAMPLES

```
/* Link with "−lm" */

#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    double complex z, c, f;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <real> <imag>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    z = atof(argv[1]) + atof(argv[2]) * I;

    c = catanh(z);
    printf("catanh() = %6.3f %6.3f*i\n", creal(c), cimag(c));

    f = 0.5 * (clog(1 + z) − clog(1 − z));
    printf("formula  = %6.3f %6.3f*i\n", creal(f), cimag(f));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

atanh(3), *cabs*(3), *cimag*(3), *ctanh*(3), *complex*(7)

NAME

catgets − get message from a message catalog

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <nl_types.h>

char *catgets(nl_catd *catalog***, int** *set_number***, int** *message_number***,**
const char * *message***);**

DESCRIPTION

catgets() reads the message *message_number*, in set *set_number*, from the message catalog identified by *catalog*, where *catalog* is a catalog descriptor returned from an earlier call to *catopen*(3). The fourth argument, *message*, points to a default message string which will be returned by **catgets**() if the identified message catalog is not currently available. The message-text is contained in an internal buffer area and should be copied by the application if it is to be saved or modified. The return string is always terminated with a null byte ('\0').

RETURN VALUE

On success, **catgets**() returns a pointer to an internal buffer area containing the null-terminated message string. On failure, **catgets**() returns the value *message*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
catgets()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

The **catgets**() function is available only in libc.so.4.4.4c and above.

The Jan 1987 X/Open Portability Guide specifies a more subtle error return: *message* is returned if the message catalog specified by *catalog* is not available, while an empty string is returned when the message catalog is available but does not contain the specified message. These two possible error returns seem to be discarded in SUSv2 in favor of always returning *message*.

SEE ALSO

catopen(3), *setlocale*(3)

NAME

catopen, catclose − open/close a message catalog

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <nl_types.h>

nl_catd catopen(const char *name, int flag);
int catclose(nl_catd catalog);
```

DESCRIPTION

The function **catopen**() opens a message catalog and returns a catalog descriptor. The descriptor remains valid until **catclose**() or *execve*(2). If a file descriptor is used to implement catalog descriptors, then the **FD_CLOEXEC** flag will be set.

The argument *name* specifies the name of the message catalog to be opened. If *name* specifies an absolute path (i.e., contains a '/'), then *name* specifies a pathname for the message catalog. Otherwise, the environment variable **NLSPATH** is used with *name* substituted for **%N** (see *locale*(7)). It is unspecified whether **NLSPATH** will be used when the process has root privileges. If **NLSPATH** does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by it, then an implementation defined path is used. This latter default path may depend on the **LC_MESSAGES** locale setting when the *flag* argument is **NL_CAT_LOCALE** and on the **LANG** environment variable when the *flag* argument is 0. Changing the **LC_MESSAGES** part of the locale may invalidate open catalog descriptors.

The *flag* argument to **catopen**() is used to indicate the source for the language to use. If it is set to **NL_CAT_LOCALE**, then it will use the current locale setting for **LC_MESSAGES**. Otherwise, it will use the **LANG** environment variable.

The function **catclose**() closes the message catalog identified by *catalog*. It invalidates any subsequent references to the message catalog defined by *catalog*.

RETURN VALUE

The function **catopen**() returns a message catalog descriptor of type *nl_catd* on success. On failure, it returns *(nl_catd) −1* and sets *errno* to indicate the error. The possible error values include all possible values for the *open*(2) call.

The function **catclose**() returns 0 on success, or −1 on failure.

ENVIRONMENT

LC_MESSAGES
: May be the source of the **LC_MESSAGES** locale setting, and thus determine the language to use if *flag* is set to **NL_CAT_LOCALE**.

LANG The language to use if *flag* is 0.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
catopen()	Thread safety	MT-Safe env
catclose()	Thread safety	MT-Safe

VERSIONS

The above is the POSIX.1 description. The glibc value for **NL_CAT_LOCALE** is 1. The default path varies, but usually looks at a number of places below */usr/share/locale*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

catgets(3), *setlocale*(3)

NAME

cbrt, cbrtf, cbrtl − cube root function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

cbrt():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

cbrtf(), **cbrtl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the (real) cube root of *x*. This function cannot fail; every representable real value has a real cube root, and rounding it to a representable value never causes overflow nor underflow.

RETURN VALUE

These functions return the cube root of *x*.

If *x* is +0, −0, positive infinity, negative infinity, or NaN, *x* is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cbrt(), **cbrtf**(), **cbrtl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

SEE ALSO

pow(3), *sqrt*(3)

NAME

ccos, ccosf, ccosl − complex cosine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex ccos(double complex *z***);**
float complex ccosf(float complex *z***);**
long double complex ccosl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex cosine of *z*.

The complex cosine function is defined as:

```
ccos(z) = (exp(i * z) + exp(−i * z)) / 2
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ccos(), **ccosf**(), **ccosl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cacos*(3), *csin*(3), *ctan*(3), *complex*(7)

NAME

ccosh, ccoshf, ccoshl − complex hyperbolic cosine

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex ccosh(double complex *z***);**
float complex ccoshf(float complex *z***);**
long double complex ccoshl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex hyperbolic cosine of *z*.

The complex hyperbolic cosine function is defined as:

```
ccosh(z) = (exp(z)+exp(−z))/2
```

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cacosh*(3), *csinh*(3), *ctanh*(3), *complex*(7)

NAME

ceil, ceilf, ceill − ceiling function: smallest integral value not less than argument

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ceilf(), **ceill**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the smallest integral value that is not less than *x*.

For example, *ceil(0.5)* is 1.0, and *ceil(−0.5)* is 0.0.

RETURN VALUE

These functions return the ceiling of *x*.

If *x* is integral, +0, −0, NaN, or infinite, *x* itself is returned.

ERRORS

No errors occur. POSIX.1-2001 documents a range error for overflows, but see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ceil(), **ceilf**(), **ceill**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

NOTES

SUSv2 and POSIX.1-2001 contain text about overflow (which might set *errno* to **ERANGE**, or raise an **FE_OVERFLOW** exception). In practice, the result cannot overflow on any current machine, so this error-handling stuff is just nonsense. (More precisely, overflow can happen only when the maximum value of the exponent is smaller than the number of mantissa bits. For the IEEE-754 standard 32-bit and 64-bit floating-point numbers the maximum value of the exponent is 127 (respectively, 1023), and the number of mantissa bits including the implicit bit is 24 (respectively, 53).)

The integral value returned by these functions may be too large to store in an integer type (*int*, *long*, etc.). To avoid an overflow, which will produce undefined results, an application should perform a range check on the returned value before assigning it to an integer type.

SEE ALSO

floor(3), *lrint*(3), *nearbyint*(3), *rint*(3), *round*(3), *trunc*(3)

NAME

cexp, cexpf, cexpl − complex exponential function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex cexp(double complex *z***);**
float complex cexpf(float complex *z***);**
long double complex cexpl(long double complex *z***);**

DESCRIPTION

These functions calculate e (2.71828..., the base of natural logarithms) raised to the power of *z*.

One has:

```
cexp(I * z) = ccos(z) + I * csin(z)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cexp(), **cexpf**(), **cexpl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cexp2*(3), *clog*(3), *cpow*(3), *complex*(7)

NAME

cexp2, cexp2f, cexp2l − base-2 exponent of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <complex.h>

double complex cexp2(double complex z);
float complex cexp2f(float complex z);
long double complex cexp2l(long double complex z);
```

DESCRIPTION

The function returns 2 raised to the power of *z*.

STANDARDS

These function names are reserved for future use in C99.

As at glibc 2.31, these functions are not provided in glibc.

SEE ALSO

cabs(3), *cexp*(3), *clog10*(3), *complex*(7)

NAME

cfree − free allocated memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

/* In SunOS 4 */
int cfree(void ****ptr*);**

/* In glibc or FreeBSD libcompat */
void cfree(void ****ptr*);**

/* In SCO OpenServer */
void cfree(char *ptr***[.***size* *** .***num***], unsigned int** *num***, unsigned int** *size***);**

/* In Solaris watchmalloc.so.1 */
void cfree(void *ptr***[.***elsize* *** .***nelem***], size_t** *nelem***, size_t** *elsize***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

cfree():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

This function should never be used. Use *free*(3) instead. Starting with glibc 2.26, it has been removed from glibc.

1-arg cfree

In glibc, the function **cfree**() is a synonym for *free*(3), "added for compatibility with SunOS".

Other systems have other functions with this name. The declaration is sometimes in <*stdlib.h*> and sometimes in <*malloc.h*>.

3-arg cfree

Some SCO and Solaris versions have malloc libraries with a 3-argument **cfree**(), apparently as an analog to *calloc*(3).

If you need it while porting something, add

```
#define cfree(p, n, s) free((p))
```

to your file.

A frequently asked question is "Can I use *free*(3) to free memory allocated with *calloc*(3), or do I need **cfree**()?" Answer: use *free*(3).

An SCO manual writes: "The cfree routine is provided for compliance to the iBCSe2 standard and simply calls free. The num and size arguments to cfree are not used."

RETURN VALUE

The SunOS version of **cfree**() (which is a synonym for *free*(3)) returns 1 on success and 0 on failure. In case of error, *errno* is set to **EINVAL**: the value of *ptr* was not a pointer to a block previously allocated by one of the routines in the *malloc*(3) family.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cfree()	Thread safety	MT-Safe /* In glibc */

VERSIONS

The 3-argument version of **cfree**() as used by SCO conforms to the iBCSe2 standard: Intel386 Binary Compatibility Specification, Edition 2.

STANDARDS

None.

HISTORY

Removed in glibc 2.26.

SEE ALSO

malloc(3)

NAME

cimag, cimagf, cimagl − get imaginary part of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double cimag(double complex *z***);**
float cimagf(float complex *z***);**
long double cimagl(long double complex *z***);**

DESCRIPTION

These functions return the imaginary part of the complex number *z*.

One has:

```
z = creal(z) + I * cimag(z)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cimag(), **cimagf**(), **cimagl**()	Thread safety	MT-Safe

VERSIONS

GCC also supports __imag__. That is a GNU extension.

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *creal*(3), *complex*(7)

NAME

CIRCLEQ_EMPTY, CIRCLEQ_ENTRY, CIRCLEQ_FIRST, CIRCLEQ_FOREACH, CIRCLEQ_FOREACH_REVERSE, CIRCLEQ_HEAD, CIRCLEQ_HEAD_INITIALIZER, CIRCLEQ_INIT, CIRCLEQ_INSERT_AFTER, CIRCLEQ_INSERT_BEFORE, CIRCLEQ_INSERT_HEAD, CIRCLEQ_INSERT_TAIL, CIRCLEQ_LAST, CIRCLEQ_LOOP_NEXT, CIRCLEQ_LOOP_PREV, CIRCLEQ_NEXT, CIRCLEQ_PREV, CIRCLEQ_REMOVE − implementation of a doubly linked circular queue

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/queue.h>

CIRCLEQ_ENTRY(TYPE);

CIRCLEQ_HEAD(HEADNAME, TYPE);
CIRCLEQ_HEAD CIRCLEQ_HEAD_INITIALIZER(CIRCLEQ_HEAD *head***);**
void CIRCLEQ_INIT(CIRCLEQ_HEAD ***_head_);**

int CIRCLEQ_EMPTY(CIRCLEQ_HEAD ***_head_);**

void CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD ***_head_,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
void CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD ***_head_,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
void CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD ***_head_, struct TYPE ***_listelm_**,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
void CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD ***_head_, struct TYPE ***_listelm_**,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**

struct TYPE *CIRCLEQ_FIRST(CIRCLEQ_HEAD ***_head_);**
struct TYPE *CIRCLEQ_LAST(CIRCLEQ_HEAD ***_head_);**
struct TYPE *CIRCLEQ_PREV(struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
struct TYPE *CIRCLEQ_NEXT(struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
struct TYPE *CIRCLEQ_LOOP_PREV(CIRCLEQ_HEAD ***_head_,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**
struct TYPE *CIRCLEQ_LOOP_NEXT(CIRCLEQ_HEAD ***_head_,**
struct TYPE ***_elm_, CIRCLEQ_ENTRY** *NAME***);**

CIRCLEQ_FOREACH(struct TYPE ***_var_, CIRCLEQ_HEAD ***_head_**,**
CIRCLEQ_ENTRY *NAME***);**
CIRCLEQ_FOREACH_REVERSE(struct TYPE ***_var_, CIRCLEQ_HEAD ***_head_**,**
CIRCLEQ_ENTRY *NAME***);**

void CIRCLEQ_REMOVE(CIRCLEQ_HEAD ***_head_, struct TYPE ***_elm_**,**
CIRCLEQ_ENTRY *NAME***);**

DESCRIPTION

These macros define and operate on doubly linked circular queues.

In the macro definitions, *TYPE* is the name of a user-defined structure, that must contain a field of type *CIRCLEQ_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user-defined structure that must be declared using the macro **CIRCLEQ_HEAD**().

Creation

A circular queue is headed by a structure defined by the **CIRCLEQ_HEAD**() macro. This structure contains a pair of pointers, one to the first element in the queue and the other to the last element in the queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A *CIRCLEQ_HEAD* structure is declared as follows:

```
CIRCLEQ_HEAD(HEADNAME, TYPE) head;
```

where *struct HEADNAME* is the structure to be defined, and *struct TYPE* is the type of the elements to be linked into the queue. A pointer to the head of the queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

CIRCLEQ_ENTRY() declares a structure that connects the elements in the queue.

CIRCLEQ_HEAD_INITIALIZER() evaluates to an initializer for the queue *head*.

CIRCLEQ_INIT() initializes the queue referenced by *head*.

CIRCLEQ_EMPTY() evaluates to true if there are no items on the queue.

Insertion

CIRCLEQ_INSERT_HEAD() inserts the new element *elm* at the head of the queue.

CIRCLEQ_INSERT_TAIL() inserts the new element *elm* at the end of the queue.

CIRCLEQ_INSERT_BEFORE() inserts the new element *elm* before the element *listelm*.

CIRCLEQ_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

Traversal

CIRCLEQ_FIRST() returns the first item on the queue.

CIRCLEQ_LAST() returns the last item on the queue.

CIRCLEQ_PREV() returns the previous item on the queue, or *&head* if this item is the first one.

CIRCLEQ_NEXT() returns the next item on the queue, or *&head* if this item is the last one.

CIRCLEQ_LOOP_PREV() returns the previous item on the queue. If *elm* is the first element on the queue, the last element is returned.

CIRCLEQ_LOOP_NEXT() returns the next item on the queue. If *elm* is the last element on the queue, the first element is returned.

CIRCLEQ_FOREACH() traverses the queue referenced by *head* in the forward direction, assigning each element in turn to *var*. *var* is set to *&head* if the loop completes normally, or if there were no elements.

CIRCLEQ_FOREACH_REVERSE() traverses the queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

Removal

CIRCLEQ_REMOVE() removes the element *elm* from the queue.

RETURN VALUE

CIRCLEQ_EMPTY() returns nonzero if the queue is empty, and zero if the queue contains at least one entry.

CIRCLEQ_FIRST(), **CIRCLEQ_LAST**(), **CIRCLEQ_LOOP_PREV**(), and **CIRCLEQ_LOOP_NEXT**() return a pointer to the first, last, previous, or next *TYPE* structure, respectively.

CIRCLEQ_PREV(), and **CIRCLEQ_NEXT**() are similar to their **CIRCLEQ_LOOP_***() counterparts, except that if the argument is the first or last element, respectively, they return *&head*.

CIRCLEQ_HEAD_INITIALIZER() returns an initializer that can be assigned to the queue *head*.

STANDARDS

BSD.

BUGS

CIRCLEQ_FOREACH() and **CIRCLEQ_FOREACH_REVERSE**() don't allow *var* to be removed or freed within the loop, as it would interfere with the traversal. **CIRCLEQ_FOREACH_SAFE**() and **CIRCLEQ_FOREACH_REVERSE_SAFE**(), which are present on the BSDs but are not present in glibc, fix this limitation by allowing *var* to safely be removed from the list and freed from within the loop without interfering with the traversal.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>
```

```
struct entry {
    int data;
    CIRCLEQ_ENTRY(entry) entries;          /* Queue */
};

CIRCLEQ_HEAD(circlehead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct circlehead head;                /* Queue head */
    int i;

    CIRCLEQ_INIT(&head);                   /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));     /* Insert at the head */
    CIRCLEQ_INSERT_HEAD(&head, n1, entries);

    n1 = malloc(sizeof(struct entry));     /* Insert at the tail */
    CIRCLEQ_INSERT_TAIL(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));     /* Insert after */
    CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

    n3 = malloc(sizeof(struct entry));     /* Insert before */
    CIRCLEQ_INSERT_BEFORE(&head, n2, n3, entries);

    CIRCLEQ_REMOVE(&head, n2, entries);    /* Deletion */
    free(n2);
                                           /* Forward traversal */
    i = 0;
    CIRCLEQ_FOREACH(np, &head, entries)
        np->data = i++;
                                           /* Reverse traversal */
    CIRCLEQ_FOREACH_REVERSE(np, &head, entries)
        printf("%i\n", np->data);
                                           /* Queue deletion */
    n1 = CIRCLEQ_FIRST(&head);
    while (n1 != (void *)&head) {
        n2 = CIRCLEQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    CIRCLEQ_INIT(&head);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

insque(3), *queue*(7)

NAME

clearenv − clear the environment

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int clearenv(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

clearenv():
/* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

The **clearenv**() function clears the environment of all name-value pairs and sets the value of the external variable *environ* to NULL. After this call, new variables can be added to the environment using *putenv*(3) and *setenv*(3).

RETURN VALUE

The **clearenv**() function returns zero on success, and a nonzero value on failure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clearenv()	Thread safety	MT-Unsafe const:env

STANDARDS

putenv()
POSIX.1-2008.

clearenv()
None.

HISTORY

putenv()
glibc 2.0. POSIX.1-2001.

clearenv()
glibc 2.0.

Various UNIX variants (DG/UX, HP-UX, QNX, ...). POSIX.9 (bindings for FORTRAN77). POSIX.1-1996 did not accept **clearenv**() and *putenv*(3), but changed its mind and scheduled these functions for some later issue of this standard (see §B.4.6.1). However, POSIX.1-2001 adds only *putenv*(3), and rejected **clearenv**().

NOTES

On systems where **clearenv**() is unavailable, the assignment

```
environ = NULL;
```

will probably do.

The **clearenv**() function may be useful in security-conscious applications that want to precisely control the environment that is passed to programs executed using *exec*(3). The application would do this by first clearing the environment and then adding select environment variables.

Note that the main effect of **clearenv**() is to adjust the value of the pointer *environ*(7); this function does not erase the contents of the buffers containing the environment definitions.

The DG/UX and Tru64 man pages write: If *environ* has been modified by anything other than the *putenv*(3), *getenv*(3), or **clearenv**() functions, then **clearenv**() will return an error and the process environment will remain unchanged.

SEE ALSO

getenv(3), *putenv*(3), *setenv*(3), *unsetenv*(3), *environ*(7)

NAME

clock − determine processor time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <time.h>

clock_t clock(void);

DESCRIPTION

The **clock**() function returns an approximation of processor time used by the program.

RETURN VALUE

The value returned is the CPU time used so far as a *clock_t*; to get the number of seconds used, divide by **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value *(clock_t) −1*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clock()	Thread safety	MT-Safe

VERSIONS

XSI requires that **CLOCKS_PER_SEC** equals 1000000 independent of the actual resolution.

On several other implementations, the value returned by **clock**() also includes the times of any children whose status has been collected via *wait*(2) (or another wait-type call). Linux does not include the times of waited-for children in the value returned by **clock**(). The *times*(2) function, which explicitly returns (separate) information about the caller and its children, may be preferable.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

In glibc 2.17 and earlier, **clock**() was implemented on top of *times*(2). For improved accuracy, since glibc 2.18, it is implemented on top of *clock_gettime*(2) (using the **CLOCK_PROCESS_CPUTIME_ID** clock).

NOTES

The C standard allows for arbitrary values at the start of the program; subtract the value returned from a call to **clock**() at the start of the program to get maximum portability.

Note that the time can wrap around. On a 32-bit system where **CLOCKS_PER_SEC** equals 1000000 this function will return the same value approximately every 72 minutes.

SEE ALSO

clock_gettime(2), *getrusage*(2), *times*(2)

NAME

clock_getcpuclockid − obtain ID of a process CPU-time clock

LIBRARY

Standard C library (*libc*, *−lc*), since glibc 2.17

Before glibc 2.17, Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <time.h>

int clock_getcpuclockid(pid_t *pid***, clockid_t ****clockid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

clock_getcpuclockid():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **clock_getcpuclockid**() function obtains the ID of the CPU-time clock of the process whose ID is *pid*, and returns it in the location pointed to by *clockid*. If *pid* is zero, then the clock ID of the CPU-time clock of the calling process is returned.

RETURN VALUE

On success, **clock_getcpuclockid**() returns 0; on error, it returns one of the positive error numbers listed in ERRORS.

ERRORS

ENOSYS
The kernel does not support obtaining the per-process CPU-time clock of another process, and *pid* does not specify the calling process.

EPERM
The caller does not have permission to access the CPU-time clock of the process specified by *pid*. (Specified in POSIX.1-2001; does not occur on Linux unless the kernel does not support obtaining the per-process CPU-time clock of another process.)

ESRCH
There is no process with the ID *pid*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clock_getcpuclockid()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

NOTES

Calling *clock_gettime*(2) with the clock ID obtained by a call to **clock_getcpuclockid**() with a *pid* of 0, is the same as using the clock ID **CLOCK_PROCESS_CPUTIME_ID**.

EXAMPLES

The example program below obtains the CPU-time clock ID of the process whose ID is given on the command line, and then uses *clock_gettime*(2) to obtain the time on that clock. An example run is the following:

```
$ ./a.out 1                 # Show CPU clock of init process
CPU-time clock for PID 1 is 2.213466748 seconds
```

Program source

```
#define _XOPEN_SOURCE 600
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    clockid_t clockid;
    struct timespec ts;

    if (argc != 2) {
        fprintf(stderr, "%s <process-ID>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (clock_getcpuclockid(atoi(argv[1]), &clockid) != 0) {
        perror("clock_getcpuclockid");
        exit(EXIT_FAILURE);
    }

    if (clock_gettime(clockid, &ts) == -1) {
        perror("clock_gettime");
        exit(EXIT_FAILURE);
    }

    printf("CPU-time clock for PID %s is %jd.%09ld seconds\n",
           argv[1], (intmax_t) ts.tv_sec, ts.tv_nsec);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

clock_getres(2), *timer_create*(2), *pthread_getcpuclockid*(3), *time*(7)

NAME

clog, clogf, clogl − natural logarithm of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex clog(double complex *z***);**
float complex clogf(float complex *z***);**
long double complex clogl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex natural logarithm of *z*, with a branch cut along the negative real axis.

The logarithm **clog**() is the inverse function of the exponential *cexp*(3). Thus, if *y* = *clog(z)*, then *z* = *cexp(y)*. The imaginary part of *y* is chosen in the interval [−pi,pi].

One has:

```
clog(z) = log(cabs(z)) + I * carg(z)
```

Note that *z* close to zero will cause an overflow.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clog(), **clogf**(), **clogl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cexp*(3), *clog10*(3), *clog2*(3), *complex*(7)

NAME

clog2, clog2f, clog2l − base-2 logarithm of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex clog2(double complex *z***);**
float complex clog2f(float complex *z***);**
long double complex clog2l(long double complex *z***);**

DESCRIPTION

The call *clog2(z)* is equivalent to *clog(z)/log(2)*.

The other functions perform the same task for *float* and *long double*.

Note that *z* close to zero will cause an overflow.

STANDARDS

None.

HISTORY

These function names are reserved for future use in C99.

Not yet in glibc, as at glibc 2.19.

SEE ALSO

cabs(3), *cexp*(3), *clog*(3), *clog10*(3), *complex*(7)

NAME

clog10, clog10f, clog10l − base-10 logarithm of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <complex.h>

double complex clog10(double complex z);
float complex clog10f(float complex z);
long double complex clog10l(long double complex z);
```

DESCRIPTION

The call *clog10(z)* is equivalent to:

```
clog(z)/log(10)
```

or equally:

```
log10(cabs(c)) + I * carg(c) / log(10)
```

The other functions perform the same task for *float* and *long double*.

Note that *z* close to zero will cause an overflow.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clog10(), **clog10f**(), **clog10l**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

The identifiers are reserved for future use in C99 and C11.

SEE ALSO

cabs(3), *cexp*(3), *clog*(3), *clog2*(3), *complex*(7)

NAME

closedir − close a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

DESCRIPTION

The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.

RETURN VALUE

The **closedir**() function returns 0 on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
: Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
closedir()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

close(2), *opendir*(3), *readdir*(3), *rewinddir*(3), *scandir*(3), *seekdir*(3), *telldir*(3)

NAME

CMSG_ALIGN, CMSG_SPACE, CMSG_NXTHDR, CMSG_FIRSTHDR − access ancillary data

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr **msgh*);
struct cmsghdr *CMSG_NXTHDR(struct msghdr **msgh*,
struct cmsghdr ***cmsg);**
size_t CMSG_ALIGN(size_t *length***);**
size_t CMSG_SPACE(size_t *length***);**
size_t CMSG_LEN(size_t *length***);**
unsigned char *CMSG_DATA(struct cmsghdr **cmsg*);

DESCRIPTION

These macros are used to create and access control messages (also called ancillary data) that are not a part of the socket payload. This control information may include the interface the packet was received on, various rarely used header fields, an extended error description, a set of file descriptors, or UNIX credentials. For instance, control messages can be used to send additional header fields such as IP options. Ancillary data is sent by calling *sendmsg*(2) and received by calling *recvmsg*(2). See their manual pages for more information.

Ancillary data is a sequence of *cmsghdr* structures with appended data. See the specific protocol man pages for the available control message types. The maximum ancillary buffer size allowed per socket can be set using */proc/sys/net/core/optmem_max*; see *socket*(7).

The *cmsghdr* structure is defined as follows:

```
struct cmsghdr {
    size_t cmsg_len;    /* Data byte count, including header
                           (type is socklen_t in POSIX) */
    int    cmsg_level;  /* Originating protocol */
    int    cmsg_type;   /* Protocol-specific type */
/* followed by
   unsigned char cmsg_data[]; */
};
```

The sequence of *cmsghdr* structures should never be accessed directly. Instead, use only the following macros:

CMSG_FIRSTHDR()
: returns a pointer to the first *cmsghdr* in the ancillary data buffer associated with the passed *msghdr*. It returns NULL if there isn't enough space for a *cmsghdr* in the buffer.

CMSG_NXTHDR()
: returns the next valid *cmsghdr* after the passed *cmsghdr*. It returns NULL when there isn't enough space left in the buffer.

  When initializing a buffer that will contain a series of *cmsghdr* structures (e.g., to be sent with *sendmsg*(2)), that buffer should first be zero-initialized to ensure the correct operation of **CMSG_NXTHDR**().

CMSG_ALIGN(),
: given a length, returns it including the required alignment. This is a constant expression.

CMSG_SPACE()
: returns the number of bytes an ancillary element with payload of the passed data length occupies. This is a constant expression.

CMSG_DATA()
: returns a pointer to the data portion of a *cmsghdr*. The pointer returned cannot be assumed to be suitably aligned for accessing arbitrary payload data types. Applications should not cast it to a pointer type matching the payload, but should instead use *memcpy*(3) to copy data to or from a suitably declared object.

CMSG_LEN()
: returns the value to store in the *cmsg_len* member of the *cmsghdr* structure, taking into account any necessary alignment. It takes the data length as an argument. This is a constant expression.

To create ancillary data, first initialize the *msg_controllen* member of the *msghdr* with the length of the control message buffer. Use **CMSG_FIRSTHDR**() on the *msghdr* to get the first control message and **CMSG_NXTHDR**() to get all subsequent ones. In each control message, initialize *cmsg_len* (with *CMSG_LEN*()), the other *cmsghdr* header fields, and the data portion using **CMSG_DATA**(). Finally, the *msg_controllen* field of the *msghdr* should be set to the sum of the **CMSG_SPACE**() of the length of all control messages in the buffer. For more information on the *msghdr*, see *recvmsg*(2).

VERSIONS

For portability, ancillary data should be accessed using only the macros described here.

In Linux, **CMSG_LEN**(), **CMSG_DATA**(), and **CMSG_ALIGN**() are constant expressions (assuming their argument is constant), meaning that these values can be used to declare the size of global variables. This may not be portable, however.

STANDARDS

CMSG_FIRSTHDR()
CMSG_NXTHDR()
CMSG_DATA()
: POSIX.1-2008.

CMSG_SPACE()
CMSG_LEN()
CMSG_ALIGN()
: Linux.

HISTORY

This ancillary data model conforms to the POSIX.1g draft, 4.4BSD-Lite, the IPv6 advanced API described in RFC 2292 and SUSv2.

CMSG_SPACE() and **CMSG_LEN**() will be included in the next POSIX release (Issue 8).

EXAMPLES

This code looks for the **IP_TTL** option in a received ancillary buffer:

```
struct msghdr msgh;
struct cmsghdr *cmsg;
int received_ttl;

/* Receive auxiliary data in msgh */

for (cmsg = CMSG_FIRSTHDR(&msgh); cmsg != NULL;
        cmsg = CMSG_NXTHDR(&msgh, cmsg)) {
    if (cmsg->cmsg_level == IPPROTO_IP
            && cmsg->cmsg_type == IP_TTL) {
        memcpy(&receive_ttl, CMSG_DATA(cmsg), sizeof(received_ttl));
        break;
    }
}

if (cmsg == NULL) {
    /* Error: IP_TTL not enabled or small buffer or I/O error */
}
```

The code below passes an array of file descriptors over a UNIX domain socket using **SCM_RIGHTS**:

```
struct msghdr msg = { 0 };
struct cmsghdr *cmsg;
int myfds[NUM_FD];  /* Contains the file descriptors to pass */
char iobuf[1];
struct iovec io = {
```

```
        .iov_base = iobuf,
        .iov_len = sizeof(iobuf)
    };
    union {         /* Ancillary data buffer, wrapped in a union
                       in order to ensure it is suitably aligned */
        char buf[CMSG_SPACE(sizeof(myfds))];
        struct cmsghdr align;
    } u;

    msg.msg_iov = &io;
    msg.msg_iovlen = 1;
    msg.msg_control = u.buf;
    msg.msg_controllen = sizeof(u.buf);
    cmsg = CMSG_FIRSTHDR(&msg);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;
    cmsg->cmsg_len = CMSG_LEN(sizeof(myfds));
    memcpy(CMSG_DATA(cmsg), myfds, sizeof(myfds));
```

For a complete code example that shows passing of file descriptors over a UNIX domain socket, see *seccomp_unotify*(2).

SEE ALSO

recvmsg(2), *sendmsg*(2)

RFC 2292

NAME

confstr − get configuration dependent string variables

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

size_t confstr(int *name***, char** *buf***[.**size**], size_t** *size***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

confstr():
_POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE

DESCRIPTION

confstr() gets the value of configuration-dependent string variables.

The *name* argument is the system variable to be queried. The following variables are supported:

_CS_GNU_LIBC_VERSION (GNU C library only; since glibc 2.3.2)
A string which identifies the GNU C library version on this system (e.g., "glibc 2.3.4").

_CS_GNU_LIBPTHREAD_VERSION (GNU C library only; since glibc 2.3.2)
A string which identifies the POSIX implementation supplied by this C library (e.g., "NPTL 2.3.4" or "linuxthreads−0.10").

_CS_PATH
A value for the **PATH** variable which indicates where all the POSIX.2 standard utilities can be found.

If *buf* is not NULL and *size* is not zero, **confstr**() copies the value of the string to *buf* truncated to *size − 1* bytes if necessary, with a null byte ('\0') as terminator. This can be detected by comparing the return value of **confstr**() against *size*.

If *size* is zero and *buf* is NULL, **confstr**() just returns the value as defined below.

RETURN VALUE

If *name* is a valid configuration variable, **confstr**() returns the number of bytes (including the terminating null byte) that would be required to hold the entire value of that variable. This value may be greater than *size*, which means that the value in *buf* is truncated.

If *name* is a valid configuration variable, but that variable does not have a value, then **confstr**() returns 0. If *name* does not correspond to a valid configuration variable, **confstr**() returns 0, and *errno* is set to **EINVAL**.

ERRORS

EINVAL
The value of *name* is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
confstr()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The following code fragment determines the path where to find the POSIX.2 system utilities:

```
char *pathbuf;
size_t n;

n = confstr(_CS_PATH, NULL, (size_t) 0);
pathbuf = malloc(n);
```

```
        if (pathbuf == NULL)
            abort();
        confstr(_CS_PATH, pathbuf, n);
```

SEE ALSO

getconf(1), *sh*(1), *exec*(3), *fpathconf*(3), *pathconf*(3), *sysconf*(3), *system*(3)

NAME

conj, conjf, conjl − calculate the complex conjugate

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex conj(double complex *z***);**
float complex conjf(float complex *z***);**
long double complex conjl(long double complex *z***);**

DESCRIPTION

These functions return the complex conjugate value of *z*. That is the value obtained by changing the sign of the imaginary part.

One has:

```
cabs(z) = csqrt(z * conj(z))
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
conj(), **conjf**(), **conjl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *csqrt*(3), *complex*(7)

NAME

copysign, copysignf, copysignl − copy sign of a number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

copysign(), **copysignf**(), **copysignl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return a value whose absolute value matches that of *x*, but whose sign bit matches that of *y*.

For example, *copysign(42.0, −1.0)* and *copysign(−42.0, −1.0)* both return −42.0.

RETURN VALUE

On success, these functions return a value whose magnitude is taken from *x* and whose sign is taken from *y*.

If *x* is a NaN, a NaN with the sign bit of *y* is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
copysign(), **copysignf**(), **copysignl**()	Thread safety	MT-Safe

VERSIONS

On architectures where the floating-point formats are not IEEE 754 compliant, these functions may treat a negative zero as positive.

STANDARDS

C11, POSIX.1-2008.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

HISTORY

C99, POSIX.1-2001, 4.3BSD.

SEE ALSO

signbit(3)

NAME

cos, cosf, cosl − cosine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

cosf(), **cosl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the cosine of *x*, where *x* is given in radians.

RETURN VALUE

On success, these functions return the cosine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity or negative infinity, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cos(), **cosf**(), **cosl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

BUGS

Before glibc 2.10, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred.

SEE ALSO

acos(3), *asin*(3), *atan*(3), *atan2*(3), *ccos*(3), *sin*(3), *sincos*(3), *tan*(3)

NAME

cosh, coshf, coshl − hyperbolic cosine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double cosh(double *x***);**
float coshf(float *x***);**
long double coshl(long double *x***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

coshf(), **coshl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the hyperbolic cosine of *x*, which is defined mathematically as:

```
cosh(x) = (exp(x) + exp(−x)) / 2
```

RETURN VALUE

On success, these functions return the hyperbolic cosine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 or −0, 1 is returned.

If *x* is positive infinity or negative infinity, positive infinity is returned.

If the result overflows, a range error occurs, and the functions return +**HUGE_VAL**, +**HUGE_VALF**, or +**HUGE_VALL**, respectively.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cosh(), **coshf**(), **coshl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

BUGS

In glibc 2.3.4 and earlier, an overflow floating-point (**FE_OVERFLOW**) exception is not raised when an overflow occurs.

SEE ALSO

acosh(3), *asinh*(3), *atanh*(3), *ccos*(3), *sinh*(3), *tanh*(3)

NAME

cpow, cpowf, cpowl − complex power function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <complex.h>

double complex cpow(double complex x, double complex z);
float complex cpowf(float complex x, float complex z);
long double complex cpowl(long double complex x,
                          long double complex z);
```

DESCRIPTION

These functions calculate *x* raised to the power *z* (with a branch cut for *x* along the negative real axis).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cpow(), **cpowf**(), **cpowl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *pow*(3), *complex*(7)

NAME

cproj, cprojf, cprojl − project into Riemann Sphere

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex cproj(double complex *z***);**
float complex cprojf(float complex *z***);**
long double complex cprojl(long double complex *z***);**

DESCRIPTION

These functions project a point in the plane onto the surface of a Riemann Sphere, the one-point compactification of the complex plane. Each finite point *z* projects to *z* itself. Every complex infinite value is projected to a single infinite value, namely to positive infinity on the real axis.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
cproj(), **cprojf**(), **cprojl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

In glibc 2.11 and earlier, the implementation does something different (a *stereographic* projection onto a Riemann Sphere).

SEE ALSO

cabs(3), *complex*(7)

NAME

CPU_SET, CPU_CLR, CPU_ISSET, CPU_ZERO, CPU_COUNT, CPU_AND, CPU_OR, CPU_XOR, CPU_EQUAL, CPU_ALLOC, CPU_ALLOC_SIZE, CPU_FREE, CPU_SET_S, CPU_CLR_S, CPU_ISSET_S, CPU_ZERO_S, CPU_COUNT_S, CPU_AND_S, CPU_OR_S, CPU_XOR_S, CPU_EQUAL_S − macros for manipulating CPU sets

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int  CPU_ISSET(int cpu, cpu_set_t *set);

int  CPU_COUNT(cpu_set_t *set);

void CPU_AND(cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_OR(cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_XOR(cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);

int  CPU_EQUAL(cpu_set_t *set1, cpu_set_t *set2);

cpu_set_t *CPU_ALLOC(int num_cpus);
void CPU_FREE(cpu_set_t *set);
size_t CPU_ALLOC_SIZE(int num_cpus);

void CPU_ZERO_S(size_t setsize, cpu_set_t *set);

void CPU_SET_S(int cpu, size_t setsize, cpu_set_t *set);
void CPU_CLR_S(int cpu, size_t setsize, cpu_set_t *set);
int  CPU_ISSET_S(int cpu, size_t setsize, cpu_set_t *set);

int  CPU_COUNT_S(size_t setsize, cpu_set_t *set);

void CPU_AND_S(size_t setsize, cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_OR_S(size_t setsize, cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);
void CPU_XOR_S(size_t setsize, cpu_set_t *destset,
        cpu_set_t *srcset1, cpu_set_t *srcset2);

int  CPU_EQUAL_S(size_t setsize, cpu_set_t *set1, cpu_set_t *set2);
```

DESCRIPTION

The *cpu_set_t* data structure represents a set of CPUs. CPU sets are used by *sched_setaffinity*(2) and similar interfaces.

The *cpu_set_t* data type is implemented as a bit mask. However, the data structure should be treated as opaque: all manipulation of CPU sets should be done via the macros described in this page.

The following macros are provided to operate on the CPU set *set*:

CPU_ZERO()
: Clears *set*, so that it contains no CPUs.

CPU_SET()
: Add CPU *cpu* to *set*.

CPU_CLR()
: Remove CPU *cpu* from *set*.

CPU_ISSET()
Test to see if CPU *cpu* is a member of *set*.

CPU_COUNT()
Return the number of CPUs in *set*.

Where a *cpu* argument is specified, it should not produce side effects, since the above macros may evaluate the argument more than once.

The first CPU on the system corresponds to a *cpu* value of 0, the next CPU corresponds to a *cpu* value of 1, and so on. No assumptions should be made about particular CPUs being available, or the set of CPUs being contiguous, since CPUs can be taken offline dynamically or be otherwise absent. The constant **CPU_SETSIZE** (currently 1024) specifies a value one greater than the maximum CPU number that can be stored in *cpu_set_t*.

The following macros perform logical operations on CPU sets:

CPU_AND()
Store the intersection of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets).

CPU_OR()
Store the union of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets).

CPU_XOR()
Store the XOR of the sets *srcset1* and *srcset2* in *destset* (which may be one of the source sets). The XOR means the set of CPUs that are in either *srcset1* or *srcset2*, but not both.

CPU_EQUAL()
Test whether two CPU set contain exactly the same CPUs.

Dynamically sized CPU sets

Because some applications may require the ability to dynamically size CPU sets (e.g., to allocate sets larger than that defined by the standard *cpu_set_t* data type), glibc nowadays provides a set of macros to support this.

The following macros are used to allocate and deallocate CPU sets:

CPU_ALLOC()
Allocate a CPU set large enough to hold CPUs in the range 0 to *num_cpus−1*.

CPU_ALLOC_SIZE()
Return the size in bytes of the CPU set that would be needed to hold CPUs in the range 0 to *num_cpus−1*. This macro provides the value that can be used for the *setsize* argument in the **CPU_*_S**() macros described below.

CPU_FREE()
Free a CPU set previously allocated by **CPU_ALLOC**().

The macros whose names end with "_S" are the analogs of the similarly named macros without the suffix. These macros perform the same tasks as their analogs, but operate on the dynamically allocated CPU set(s) whose size is *setsize* bytes.

RETURN VALUE

CPU_ISSET() and **CPU_ISSET_S**() return nonzero if *cpu* is in *set*; otherwise, it returns 0.

CPU_COUNT() and **CPU_COUNT_S**() return the number of CPUs in *set*.

CPU_EQUAL() and **CPU_EQUAL_S**() return nonzero if the two CPU sets are equal; otherwise they return 0.

CPU_ALLOC() returns a pointer on success, or NULL on failure. (Errors are as for *malloc*(3).)

CPU_ALLOC_SIZE() returns the number of bytes required to store a CPU set of the specified cardinality.

The other functions do not return a value.

STANDARDS

Linux.

HISTORY

The **CPU_ZERO**(), **CPU_SET**(), **CPU_CLR**(), and **CPU_ISSET**() macros were added in glibc 2.3.3.

CPU_COUNT() first appeared in glibc 2.6.

CPU_AND(), **CPU_OR**(), **CPU_XOR**(), **CPU_EQUAL**(), **CPU_ALLOC**(), **CPU_ALLOC_SIZE**(), **CPU_FREE**(), **CPU_ZERO_S**(), **CPU_SET_S**(), **CPU_CLR_S**(), **CPU_ISSET_S**(), **CPU_AND_S**(), **CPU_OR_S**(), **CPU_XOR_S**(), and **CPU_EQUAL_S**() first appeared in glibc 2.7.

NOTES

To duplicate a CPU set, use *memcpy*(3).

Since CPU sets are bit masks allocated in units of long words, the actual number of CPUs in a dynamically allocated CPU set will be rounded up to the next multiple of *sizeof(unsigned long)*. An application should consider the contents of these extra bits to be undefined.

Notwithstanding the similarity in the names, note that the constant **CPU_SETSIZE** indicates the number of CPUs in the *cpu_set_t* data type (thus, it is effectively a count of the bits in the bit mask), while the *setsize* argument of the **CPU_*_S**() macros is a size in bytes.

The data types for arguments and return values shown in the SYNOPSIS are hints what about is expected in each case. However, since these interfaces are implemented as macros, the compiler won't necessarily catch all type errors if you violate the suggestions.

BUGS

On 32-bit platforms with glibc 2.8 and earlier, **CPU_ALLOC**() allocates twice as much space as is required, and **CPU_ALLOC_SIZE**() returns a value twice as large as it should. This bug should not affect the semantics of a program, but does result in wasted memory and less efficient operation of the macros that operate on dynamically allocated CPU sets. These bugs are fixed in glibc 2.9.

EXAMPLES

The following program demonstrates the use of some of the macros used for dynamically allocated CPU sets.

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <assert.h>

int
main(int argc, char *argv[])
{
    cpu_set_t *cpusetp;
    size_t size, num_cpus;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <num-cpus>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    num_cpus = atoi(argv[1]);

    cpusetp = CPU_ALLOC(num_cpus);
    if (cpusetp == NULL) {
        perror("CPU_ALLOC");
        exit(EXIT_FAILURE);
    }

    size = CPU_ALLOC_SIZE(num_cpus);
```

```
        CPU_ZERO_S(size, cpusetp);
        for (size_t cpu = 0; cpu < num_cpus; cpu += 2)
            CPU_SET_S(cpu, size, cpusetp);

        printf("CPU_COUNT() of set:    %d\n", CPU_COUNT_S(size, cpusetp));

        CPU_FREE(cpusetp);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

sched_setaffinity(2), *pthread_attr_setaffinity_np*(3), *pthread_setaffinity_np*(3), *cpuset*(7)

NAME

creal, crealf, creall − get real part of a complex number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double creal(double complex *z***);**
float crealf(float complex *z***);**
long double creall(long double complex *z***);**

DESCRIPTION

These functions return the real part of the complex number *z*.

One has:

z = creal(z) + I * cimag(z)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
creal(), **crealf**(), **creall**()	Thread safety	MT-Safe

VERSIONS

GCC supports also __real__. That is a GNU extension.

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cimag*(3), *complex*(7)

NAME

crypt, crypt_r − password hashing

LIBRARY

Password hashing library (*libcrypt*, *−lcrypt*)

SYNOPSIS

#include <unistd.h>

char *crypt(const char *_key_, const char *_salt_);

#include <crypt.h>

char *crypt_r(const char *_key_, const char *_salt_,
struct crypt_data *restrict *data*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

crypt():
Since glibc 2.28:
_DEFAULT_SOURCE
glibc 2.27 and earlier:
_XOPEN_SOURCE

crypt_r():
_GNU_SOURCE

DESCRIPTION

crypt() is the password hashing function. It is based on the Data Encryption Standard algorithm with variations intended (among other things) to discourage use of hardware implementations of a key search.

key is a user's typed password.

salt is a two-character string chosen from the set [**a−zA−Z0−9./**]. This string is used to perturb the algorithm in one of 4096 different ways.

By taking the lowest 7 bits of each of the first eight characters of the *key*, a 56-bit key is obtained. This 56-bit key is used to encrypt repeatedly a constant string (usually a string consisting of all zeros). The returned value points to the hashed password, a series of 13 printable ASCII characters (the first two characters represent the salt itself). The return value points to static data whose content is overwritten by each call.

Warning: the key space consists of 2^{56} equal 7.2e16 possible values. Exhaustive searches of this key space are possible using massively parallel computers. Software, such as *crack*(1), is available which will search the portion of this key space that is generally used by humans for passwords. Hence, password selection should, at minimum, avoid common words and names. The use of a *passwd*(1) program that checks for crackable passwords during the selection process is recommended.

The DES algorithm itself has a few quirks which make the use of the **crypt**() interface a very poor choice for anything other than password authentication. If you are planning on using the **crypt**() interface for a cryptography project, don't do it: get a good book on encryption and one of the widely available DES libraries.

crypt_r() is a reentrant version of **crypt**(). The structure pointed to by *data* is used to store result data and bookkeeping information. Other than allocating it, the only thing that the caller should do with this structure is to set *data−>initialized* to zero before the first call to **crypt_r**().

RETURN VALUE

On success, a pointer to the hashed password is returned. On error, NULL is returned.

ERRORS

EINVAL
salt has the wrong format.

ENOSYS
The **crypt**() function was not implemented, probably because of U.S.A. export restrictions.

EPERM
/proc/sys/crypto/fips_enabled has a nonzero value, and an attempt was made to use a weak hashing type, such as DES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
crypt()	Thread safety	MT-Unsafe race:crypt
crypt_r()	Thread safety	MT-Safe

STANDARDS

crypt() POSIX.1-2008.

crypt_r()
GNU.

HISTORY

crypt() POSIX.1-2001, SVr4, 4.3BSD.

Availability in glibc

The **crypt**(), *encrypt*(3), and *setkey*(3) functions are part of the POSIX.1-2008 XSI Options Group for Encryption and are optional. If the interfaces are not available, then the symbolic constant **_XOPEN_CRYPT** is either not defined, or it is defined to −1 and availability can be checked at run time with *sysconf*(3). This may be the case if the downstream distribution has switched from glibc crypt to *libxcrypt*. When recompiling applications in such distributions, the programmer must detect if **_XOPEN_CRYPT** is not available and include <*crypt.h*> for the function prototypes; otherwise *libxcrypt* is an ABI-compatible drop-in replacement.

NOTES

Features in glibc

The glibc version of this function supports additional hashing algorithms.

If *salt* is a character string starting with the characters "$*id*$" followed by a string optionally terminated by "$", then the result has the form:

$*id*$*salt*$*hashed*

id identifies the hashing method used instead of DES and this then determines how the rest of the password string is interpreted. The following values of *id* are supported:

ID	Method
1	MD5
2a	Blowfish (not in mainline glibc; added in some Linux distributions)
5	SHA-256 (since glibc 2.7)
6	SHA-512 (since glibc 2.7)

Thus, 5*salt*$*hashed* and 6*salt*$*hashed* contain the password hashed with, respectively, functions based on SHA-256 and SHA-512.

"*salt*" stands for the up to 16 characters following "$*id*$" in the salt. The "*hashed*" part of the password string is the actual computed password. The size of this string is fixed:

MD5 22 characters
SHA-256 43 characters
SHA-512 86 characters

The characters in "*salt*" and "*hashed*" are drawn from the set [**a−zA−Z0−9./**]. In the MD5 and SHA implementations the entire *key* is significant (instead of only the first 8 bytes in DES).

Since glibc 2.7, the SHA-256 and SHA-512 implementations support a user-supplied number of hashing rounds, defaulting to 5000. If the "$*id*$" characters in the salt are followed by "rounds=*xxx*$", where *xxx* is an integer, then the result has the form

$*id*$*rounds=yyy*$*salt*$*hashed*

where *yyy* is the number of hashing rounds actually used. The number of rounds actually used is 1000 if *xxx* is less than 1000, 999999999 if *xxx* is greater than 999999999, and is equal to *xxx* otherwise.

SEE ALSO

login(1), *passwd*(1), *encrypt*(3), *getpass*(3), *passwd*(5)

NAME

csin, csinf, csinl − complex sine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex csin(double complex *z***);**
float complex csinf(float complex *z***);**
long double complex csinl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex sine of *z*.

The complex sine function is defined as:

```
csin(z) = (exp(i * z) − exp(−i * z)) / (2 * i)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
csin(), **csinf**(), **csinl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *casin*(3), *ccos*(3), *ctan*(3), *complex*(7)

NAME

csinh, csinhf, csinhl − complex hyperbolic sine

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex csinh(double complex *z***);**
float complex csinhf(float complex *z***);**
long double complex csinhl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex hyperbolic sine of *z*.

The complex hyperbolic sine function is defined as:

```
csinh(z) = (exp(z)−exp(−z))/2
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
csinh(), **csinhf**(), **csinhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *casinh*(3), *ccosh*(3), *ctanh*(3), *complex*(7)

NAME

csqrt, csqrtf, csqrtl − complex square root

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex csqrt(double complex *z***);**
float complex csqrtf(float complex *z***);**
long double complex csqrtl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex square root of *z*, with a branch cut along the negative real axis. (That means that *csqrt(−1+eps*I)* will be close to I while *csqrt(−1−eps*I)* will be close to −I, *if eps* is a small positive real number.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
csqrt(), **csqrtf**(), **csqrtl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *cexp*(3), *complex*(7)

NAME

ctan, ctanf, ctanl − complex tangent function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex ctan(double complex *z***);**
float complex ctanf(float complex *z***);**
long double complex ctanl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex tangent of *z*.

The complex tangent function is defined as:

```
ctan(z) = csin(z) / ccos(z)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ctan(), **ctanf**(), **ctanl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *catan*(3), *ccos*(3), *csin*(3), *complex*(7)

NAME

ctanh, ctanhf, ctanhl − complex hyperbolic tangent

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

double complex ctanh(double complex *z***);**
float complex ctanhf(float complex *z***);**
long double complex ctanhl(long double complex *z***);**

DESCRIPTION

These functions calculate the complex hyperbolic tangent of *z*.

The complex hyperbolic tangent function is defined mathematically as:

```
ctanh(z) = csinh(z) / ccosh(z)
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ctanh(), **ctanhf**(), **ctanhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

cabs(3), *catanh*(3), *ccosh*(3), *csinh*(3), *complex*(7)

NAME

ctermid − get controlling terminal name

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

char *ctermid(char ****s*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ctermid():
_POSIX_C_SOURCE

DESCRIPTION

ctermid() returns a string which is the pathname for the current controlling terminal for this process. If *s* is NULL, a static buffer is used, otherwise *s* points to a buffer used to hold the terminal pathname. The symbolic constant **L_ctermid** is the maximum number of characters in the returned pathname.

RETURN VALUE

The pointer to the pathname.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ctermid()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, Svr4.

BUGS

The returned pathname may not uniquely identify the controlling terminal; it may, for example, be */dev/tty*.

It is not assured that the program can open the terminal.

SEE ALSO

ttyname(3)

NAME

asctime, ctime, gmtime, localtime, mktime, asctime_r, ctime_r, gmtime_r, localtime_r − transform date and time to broken-down time or ASCII

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *restrict tm,
                char buf[restrict 26]);

char *ctime(const time_t *timep);
char *ctime_r(const time_t *restrict timep,
                char buf[restrict 26]);

struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *restrict timep,
                struct tm *restrict result);

struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *restrict timep,
                struct tm *restrict result);

time_t mktime(struct tm *tm);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

asctime_r(), **ctime_r**(), **gmtime_r**(), **localtime_r**():
_POSIX_C_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **ctime**(), **gmtime**(), and **localtime**() functions all take an argument of data type *time_t*, which represents calendar time. When interpreted as an absolute time value, it represents the number of seconds elapsed since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

The **asctime**() and **mktime**() functions both take an argument representing broken-down time, which is a representation separated into year, month, day, and so on.

Broken-down time is stored in the structure *tm*, described in *tm*(3type).

The call **ctime(*t*)** is equivalent to **asctime(localtime(*t*))**. It converts the calendar time *t* into a null-terminated string of the form

```
"Wed Jun 30 21:49:08 1993\n"
```

The abbreviations for the days of the week are "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", and "Sat". The abbreviations for the months are "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", and "Dec". The return value points to a statically allocated string which might be overwritten by subsequent calls to any of the date and time functions. The function also sets the external variables *tzname*, *timezone*, and *daylight* (see *tzset*(3)) with information about the current timezone. The reentrant version **ctime_r**() does the same, but stores the string in a user-supplied buffer which should have room for at least 26 bytes. It need not set *tzname*, *timezone*, and *daylight*.

The **gmtime**() function converts the calendar time *timep* to broken-down time representation, expressed in Coordinated Universal Time (UTC). It may return NULL when the year does not fit into an integer. The return value points to a statically allocated struct which might be overwritten by subsequent calls to any of the date and time functions. The **gmtime_r**() function does the same, but stores the data in a user-supplied struct.

The **localtime**() function converts the calendar time *timep* to broken-down time representation, expressed relative to the user's specified timezone. The function acts as if it called *tzset*(3) and sets the external variables *tzname* with information about the current timezone, *timezone* with the difference between Coordinated Universal Time (UTC) and local standard time in seconds, and *daylight* to a nonzero value if daylight savings time rules apply during some part of the year. The return value points to a statically allocated struct which might be overwritten by subsequent calls to any of the date and

time functions. The **localtime_r**() function does the same, but stores the data in a user-supplied struct. It need not set *tzname*, *timezone*, and *daylight*.

The **asctime**() function converts the broken-down time value *tm* into a null-terminated string with the same format as **ctime**(). The return value points to a statically allocated string which might be overwritten by subsequent calls to any of the date and time functions. The **asctime_r**() function does the same, but stores the string in a user-supplied buffer which should have room for at least 26 bytes.

The **mktime**() function converts a broken-down time structure, expressed as local time, to calendar time representation. The function ignores the values supplied by the caller in the *tm_wday* and *tm_yday* fields. The value specified in the *tm_isdst* field informs **mktime**() whether or not daylight saving time (DST) is in effect for the time supplied in the *tm* structure: a positive value means DST is in effect; zero means that DST is not in effect; and a negative value means that **mktime**() should (use timezone information and system databases to) attempt to determine whether DST is in effect at the specified time.

The **mktime**() function modifies the fields of the *tm* structure as follows: *tm_wday* and *tm_yday* are set to values determined from the contents of the other fields; if structure members are outside their valid interval, they will be normalized (so that, for example, 40 October is changed into 9 November); *tm_isdst* is set (regardless of its initial value) to a positive value or to 0, respectively, to indicate whether DST is or is not in effect at the specified time. Calling **mktime**() also sets the external variable *tzname* with information about the current timezone.

If the specified broken-down time cannot be represented as calendar time (seconds since the Epoch), **mktime**() returns *(time_t) −1* and does not alter the members of the broken-down time structure.

RETURN VALUE

On success, **gmtime**() and **localtime**() return a pointer to a *struct tm*.

On success, **gmtime_r**() and **localtime_r**() return the address of the structure pointed to by *result*.

On success, **asctime**() and **ctime**() return a pointer to a string.

On success, **asctime_r**() and **ctime_r**() return a pointer to the string pointed to by *buf*.

On success, **mktime**() returns the calendar time (seconds since the Epoch), expressed as a value of type *time_t*.

On error, **mktime**() returns the value *(time_t) −1*. The remaining functions return NULL on error. On error, *errno* is set to indicate the error.

ERRORS

EOVERFLOW
The result cannot be represented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
asctime()	Thread safety	MT-Unsafe race:asctime locale
asctime_r()	Thread safety	MT-Safe locale
ctime()	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
ctime_r(), **gmtime_r**(), **localtime_r**(), **mktime**()	Thread safety	MT-Safe env locale
gmtime(), **localtime**()	Thread safety	MT-Unsafe race:tmbuf env locale

VERSIONS

POSIX doesn't specify the parameters of **ctime_r**() to be *restrict*; that is specific to glibc.

In many implementations, including glibc, a 0 in *tm_mday* is interpreted as meaning the last day of the preceding month.

According to POSIX.1-2001, **localtime**() is required to behave as though *tzset*(3) was called, while **localtime_r**() does not have this requirement. For portable code, *tzset*(3) should be called before **localtime_r**().

STANDARDS

asctime()
ctime()
gmtime()
localtime()
mktime()
C11, POSIX.1-2008.

asctime_r()
ctime_r()
gmtime_r()
localtime_r()
POSIX.1-2008.

HISTORY

gmtime()
localtime()
mktime()
C89, POSIX.1-2001.

asctime()
ctime() C89, POSIX.1-2001. Marked obsolete in POSIX.1-2008 (recommending *strftime*(3)).

gmtime_r()
localtime_r()
POSIX.1-2001.

asctime_r()
ctime_r()
POSIX.1-2001. Marked obsolete in POSIX.1-2008 (recommending *strftime*(3)).

NOTES

The four functions **asctime**(), **ctime**(), **gmtime**(), and **localtime**() return a pointer to static data and hence are not thread-safe. The thread-safe versions, **asctime_r**(), **ctime_r**(), **gmtime_r**(), and **localtime_r**(), are specified by SUSv2.

POSIX.1-2001 says: "The **asctime**(), **ctime**(), **gmtime**(), and **localtime**() functions shall return values in one of two static objects: a broken-down time structure and an array of type *char*. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions." This can occur in the glibc implementation.

SEE ALSO

date(1), *gettimeofday*(2), *time*(2), *utime*(2), *clock*(3), *difftime*(3), *strftime*(3), *strptime*(3), *timegm*(3), *tzset*(3), *time*(7)

NAME

daemon − run in the background

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int daemon(int *nochdir***, int** *noclose***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

daemon():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

The **daemon**() function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

If *nochdir* is zero, **daemon**() changes the process's current working directory to the root directory ("/"); otherwise, the current working directory is left unchanged.

If *noclose* is zero, **daemon**() redirects standard input, standard output, and standard error to */dev/null*; otherwise, no changes are made to these file descriptors.

RETURN VALUE

(This function forks, and if the *fork*(2) succeeds, the parent calls *_exit*(2), so that further errors are seen by the child only.) On success **daemon**() returns zero. If an error occurs, **daemon**() returns −1 and sets *errno* to any of the errors specified for the *fork*(2) and *setsid*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
daemon()	Thread safety	MT-Safe

VERSIONS

A similar function appears on the BSDs.

The glibc implementation can also return −1 when */dev/null* exists but is not a character device with the expected major and minor numbers. In this case, *errno* need not be set.

STANDARDS

None.

HISTORY

4.4BSD.

BUGS

The GNU C library implementation of this function was taken from BSD, and does not employ the double-fork technique (i.e., *fork*(2), *setsid*(2), *fork*(2)) that is necessary to ensure that the resulting daemon process is not a session leader. Instead, the resulting daemon *is* a session leader. On systems that follow System V semantics (e.g., Linux), this means that if the daemon opens a terminal that is not already a controlling terminal for another session, then that terminal will inadvertently become the controlling terminal for the daemon.

SEE ALSO

fork(2), *setsid*(2), *daemon*(7), *logrotate*(8)

NAME

dbopen − database access methods

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <limits.h>
#include <db.h>
#include <fcntl.h>

DB *dbopen(const char * *file***, int** *flags***, int** *mode***, DBTYPE** *type***,**
const void ***openinfo*);**

DESCRIPTION

Note well: This page documents interfaces provided up until glibc 2.1. Since glibc 2.2, glibc no longer provides these interfaces. Probably, you are looking for the APIs provided by the *libdb* library instead.

dbopen() is the library interface to database files. The supported file formats are btree, hashed, and UNIX file oriented. The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The flat-file format is a byte stream file with fixed or variable length records. The formats and file-format-specific information are described in detail in their respective manual pages *btree*(3), *hash*(3), and *recno*(3).

dbopen() opens *file* for reading and/or writing. Files never intended to be preserved on disk may be created by setting the *file* argument to NULL.

The *flags* and *mode* arguments are as specified to the *open*(2) routine, however, only the **O_CREAT**, **O_EXCL**, **O_EXLOCK**, **O_NONBLOCK**, **O_RDONLY**, **O_RDWR**, **O_SHLOCK**, and **O_TRUNC** flags are meaningful. (Note, opening a database file **O_WRONLY** is not possible.)

The *type* argument is of type *DBTYPE* (as defined in the <*db.h*> include file) and may be set to **DB_BTREE**, **DB_HASH**, or **DB_RECNO**.

The *openinfo* argument is a pointer to an access-method-specific structure described in the access method's manual page. If *openinfo* is NULL, each access method will use defaults appropriate for the system and the access method.

dbopen() returns a pointer to a *DB* structure on success and NULL on error. The *DB* structure is defined in the <*db.h*> include file, and contains at least the following fields:

```
typedef struct {
    DBTYPE type;
    int (*close)(const DB *db);
    int (*del)(const DB *db, const DBT *key, unsigned int flags);
    int (*fd)(const DB *db);
    int (*get)(const DB *db, DBT *key, DBT *data,
               unsigned int flags);
    int (*put)(const DB *db, DBT *key, const DBT *data,
               unsigned int flags);
    int (*sync)(const DB *db, unsigned int flags);
    int (*seq)(const DB *db, DBT *key, DBT *data,
               unsigned int flags);
} DB;
```

These elements describe a database type and a set of functions performing various actions. These functions take a pointer to a structure as returned by **dbopen**(), and sometimes one or more pointers to key/data structures and a flag value.

type The type of the underlying access method (and file format).

close A pointer to a routine to flush any cached information to disk, free any allocated resources, and close the underlying file(s). Since key/data pairs may be cached in memory, failing to sync the file with a *close* or *sync* function may result in inconsistent or lost information. *close* routines return −1 on error (setting *errno*) and 0 on success.

del A pointer to a routine to remove key/data pairs from the database.

The argument *flag* may be set to the following value:

R_CURSOR
Delete the record referenced by the cursor. The cursor must have previously been initialized.

delete routines return −1 on error (setting *errno*), 0 on success, and 1 if the specified *key* was not in the file.

fd A pointer to a routine which returns a file descriptor representative of the underlying database. A file descriptor referencing the same file will be returned to all processes which call **dbopen**() with the same *file* name. This file descriptor may be safely used as an argument to the *fcntl*(2) and *flock*(2) locking functions. The file descriptor is not necessarily associated with any of the underlying files used by the access method. No file descriptor is available for in memory databases. *fd* routines return −1 on error (setting *errno*), and the file descriptor on success.

get A pointer to a routine which is the interface for keyed retrieval from the database. The address and length of the data associated with the specified *key* are returned in the structure referenced by *data*. *get* routines return −1 on error (setting *errno*), 0 on success, and 1 if the *key* was not in the file.

put A pointer to a routine to store key/data pairs in the database.

The argument *flag* may be set to one of the following values:

R_CURSOR
Replace the key/data pair referenced by the cursor. The cursor must have previously been initialized.

R_IAFTER
Append the data immediately after the data referenced by *key*, creating a new key/data pair. The record number of the appended key/data pair is returned in the *key* structure. (Applicable only to the **DB_RECNO** access method.)

R_IBEFORE
Insert the data immediately before the data referenced by *key*, creating a new key/data pair. The record number of the inserted key/data pair is returned in the *key* structure. (Applicable only to the **DB_RECNO** access method.)

R_NOOVERWRITE
Enter the new key/data pair only if the key does not previously exist.

R_SETCURSOR
Store the key/data pair, setting or initializing the position of the cursor to reference it. (Applicable only to the **DB_BTREE** and **DB_RECNO** access methods.)

R_SETCURSOR is available only for the **DB_BTREE** and **DB_RECNO** access methods because it implies that the keys have an inherent order which does not change.

R_IAFTER and **R_IBEFORE** are available only for the **DB_RECNO** access method because they each imply that the access method is able to create new keys. This is true only if the keys are ordered and independent, record numbers for example.

The default behavior of the *put* routines is to enter the new key/data pair, replacing any previously existing key.

put routines return −1 on error (setting *errno*), 0 on success, and 1 if the **R_NOOVERWRITE** *flag* was set and the key already exists in the file.

seq A pointer to a routine which is the interface for sequential retrieval from the database. The address and length of the key are returned in the structure referenced by *key*, and the address and length of the data are returned in the structure referenced by *data*.

Sequential key/data pair retrieval may begin at any time, and the position of the "cursor" is not affected by calls to the *del*, *get*, *put*, or *sync* routines. Modifications to the database during a sequential scan will be reflected in the scan, that is, records inserted behind the cursor will not

be returned while records inserted in front of the cursor will be returned.

The flag value **must** be set to one of the following values:

R_CURSOR
The data associated with the specified key is returned. This differs from the *get* routines in that it sets or initializes the cursor to the location of the key as well. (Note, for the **DB_BTREE** access method, the returned key is not necessarily an exact match for the specified key. The returned key is the smallest key greater than or equal to the specified key, permitting partial key matches and range searches.)

R_FIRST
The first key/data pair of the database is returned, and the cursor is set or initialized to reference it.

R_LAST
The last key/data pair of the database is returned, and the cursor is set or initialized to reference it. (Applicable only to the **DB_BTREE** and **DB_RECNO** access methods.)

R_NEXT
Retrieve the key/data pair immediately after the cursor. If the cursor is not yet set, this is the same as the **R_FIRST** flag.

R_PREV
Retrieve the key/data pair immediately before the cursor. If the cursor is not yet set, this is the same as the **R_LAST** flag. (Applicable only to the **DB_BTREE** and **DB_RECNO** access methods.)

R_LAST and **R_PREV** are available only for the **DB_BTREE** and **DB_RECNO** access methods because they each imply that the keys have an inherent order which does not change.

seq routines return −1 on error (setting *errno*), 0 on success and 1 if there are no key/data pairs less than or greater than the specified or current key. If the **DB_RECNO** access method is being used, and if the database file is a character special file and no complete key/data pairs are currently available, the *seq* routines return 2.

sync A pointer to a routine to flush any cached information to disk. If the database is in memory only, the *sync* routine has no effect and will always succeed.

The flag value may be set to the following value:

R_RECNOSYNC
If the **DB_RECNO** access method is being used, this flag causes the sync routine to apply to the btree file which underlies the recno file, not the recno file itself. (See the *bfname* field of the *recno*(3) manual page for more information.)

sync routines return −1 on error (setting *errno*) and 0 on success.

Key/data pairs

Access to all file types is based on key/data pairs. Both keys and data are represented by the following data structure:

```
typedef struct {
    void  *data;
    size_t size;
} DBT;
```

The elements of the *DBT* structure are defined as follows:

data A pointer to a byte string.

size The length of the byte string.

Key and data byte strings may reference strings of essentially unlimited length although any two of them must fit into available memory at the same time. It should be noted that the access methods provide no guarantees about byte string alignment.

ERRORS

The **dbopen**() routine may fail and set *errno* for any of the errors specified for the library routines *open*(2) and *malloc*(3) or the following:

EFTYPE
A file is incorrectly formatted.

EINVAL
A parameter has been specified (hash function, pad byte, etc.) that is incompatible with the current file specification or which is not meaningful for the function (for example, use of the cursor without prior initialization) or there is a mismatch between the version number of file and the software.

The *close* routines may fail and set *errno* for any of the errors specified for the library routines *close*(2), *read*(2), *write*(2), *free*(3), or *fsync*(2).

The *del*, *get*, *put*, and *seq* routines may fail and set *errno* for any of the errors specified for the library routines *read*(2), *write*(2), *free*(3), or *malloc*(3).

The *fd* routines will fail and set *errno* to **ENOENT** for in memory databases.

The *sync* routines may fail and set *errno* for any of the errors specified for the library routine *fsync*(2).

BUGS

The typedef *DBT* is a mnemonic for "data base thang", and was used because no one could think of a reasonable name that wasn't already used.

The file descriptor interface is a kludge and will be deleted in a future version of the interface.

None of the access methods provide any form of concurrent access, locking, or transactions.

SEE ALSO

btree(3), *hash*(3), *mpool*(3), *recno*(3)

LIBTP: Portable, Modular Transactions for UNIX, Margo Seltzer, Michael Olson, USENIX proceedings, Winter 1992.

NAME

des_crypt, ecb_crypt, cbc_crypt, des_setparity, DES_FAILED − fast DES encryption

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <rpc/des_crypt.h>

[[deprecated]] int ecb_crypt(char ****key*, char** *data***[.***datalen***],**
unsigned int *datalen***, unsigned int** *mode***);**
[[deprecated]] int cbc_crypt(char ****key*, char** *data***[.***datalen***],**
unsigned int *datalen***, unsigned int** *mode***,**
char ****ivec*);**

[[deprecated]] void des_setparity(char ****key*);**

[[deprecated]] int DES_FAILED(int *status***);**

DESCRIPTION

ecb_crypt() and **cbc_crypt**() implement the NBS DES (Data Encryption Standard). These routines are faster and more general purpose than *crypt*(3). They also are able to utilize DES hardware if it is available. **ecb_crypt**() encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently. **cbc_crypt**() encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions, and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

Here is how to use these routines. The first argument, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use **des_setparity**(). The second argument, *data*, contains the data to be encrypted or decrypted. The third argument, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth argument, *mode*, is formed by ORing together some things. For the encryption direction OR in either **DES_ENCRYPT** or **DES_DECRYPT**. For software versus hardware encryption, OR in either **DES_HW** or **DES_SW**. If **DES_HW** is specified, and there is no hardware, then the encryption is performed in software and the routine returns **DESERR_NOHWDEVICE**. For **cbc_crypt**(), the argument *ivec* is the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon return.

RETURN VALUE

DESERR_NONE
No error.

DESERR_NOHWDEVICE
Encryption succeeded, but done in software instead of the requested hardware.

DESERR_HWERROR
An error occurred in the hardware or driver.

DESERR_BADPARAM
Bad argument to routine.

Given a result status *stat*, the macro **DES_FAILED(***stat***)** is false only for the first two statuses.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ecb_crypt(), **cbc_crypt**(), **des_setparity**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD. glibc 2.1. Removed in glibc 2.28.

Because they employ the DES block cipher, which is no longer considered secure, these functions were removed. Applications should switch to a modern cryptography library, such as **libgcrypt**.

SEE ALSO

des(1), *crypt*(3), *xcrypt*(3)

NAME

difftime − calculate time difference

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <time.h>

double difftime(time_t *time1***, time_t** *time0***);**

DESCRIPTION

The **difftime**() function returns the number of seconds elapsed between time *time1* and time *time0*, represented as a *double*. Each time is a count of seconds.

difftime(b, a) acts like *(b−a)* except that the result does not overflow and is rounded to *double*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
difftime()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

date(1), *gettimeofday*(2), *time*(2), *ctime*(3), *gmtime*(3), *localtime*(3)

NAME

dirfd − get directory stream file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

int dirfd(DIR *dirp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

dirfd():

```
/* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200809L
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

The function **dirfd**() returns the file descriptor associated with the directory stream *dirp*.

This file descriptor is the one used internally by the directory stream. As a result, it is useful only for functions which do not depend on or alter the file position, such as *fstat*(2) and *fchdir*(2). It will be automatically closed when *closedir*(3) is called.

RETURN VALUE

On success, **dirfd**() returns a file descriptor (a nonnegative integer). On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

POSIX.1-2008 specifies two errors, neither of which is returned by the current implementation.

EINVAL
: *dirp* does not refer to a valid directory stream.

ENOTSUP
: The implementation does not support the association of a file descriptor with a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dirfd()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

4.3BSD-Reno (not in 4.2BSD).

SEE ALSO

open(2), *openat*(2), *closedir*(3), *opendir*(3), *readdir*(3), *rewinddir*(3), *scandir*(3), *seekdir*(3), *telldir*(3)

NAME

div, ldiv, lldiv, imaxdiv − compute quotient and remainder of an integer division

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

div_t div(int numerator, int denominator);
ldiv_t ldiv(long numerator, long denominator);
lldiv_t lldiv(long long numerator, long long denominator);

#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numerator, intmax_t denominator);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

lldiv():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **div**() function computes the value *numerator*/*denominator* and returns the quotient and remainder in a structure named *div_t* that contains two integer members (in unspecified order) named *quot* and *rem*. The quotient is rounded toward zero. The result satisfies *quot***denominator*+*rem* = *numerator*.

The **ldiv**(), **lldiv**(), and **imaxdiv**() functions do the same, dividing numbers of the indicated type and returning the result in a structure of the indicated name, in all cases with fields *quot* and *rem* of the same type as the function arguments.

RETURN VALUE

The *div_t* (etc.) structure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
div(), **ldiv**(), **lldiv**(), **imaxdiv**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

lldiv() and **imaxdiv**() were added in C99.

EXAMPLES

After

```
div_t q = div(−5, 3);
```

the values *q.quot* and *q.rem* are −1 and −2, respectively.

SEE ALSO

abs(3), *remainder*(3)

NAME

dl_iterate_phdr − walk through list of shared objects

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <link.h>

int dl_iterate_phdr(
      int (*callback)(struct dl_phdr_info *info,
                      size_t size, void *data),
      void *data);
```

DESCRIPTION

The **dl_iterate_phdr**() function allows an application to inquire at run time to find out which shared objects it has loaded, and the order in which they were loaded.

The **dl_iterate_phdr**() function walks through the list of an application's shared objects and calls the function *callback* once for each object, until either all shared objects have been processed or *callback* returns a nonzero value.

Each call to *callback* receives three arguments: *info*, which is a pointer to a structure containing information about the shared object; *size*, which is the size of the structure pointed to by *info*; and *data*, which is a copy of whatever value was passed by the calling program as the second argument (also named *data*) in the call to **dl_iterate_phdr**().

The *info* argument is a structure of the following type:

```
struct dl_phdr_info {
    ElfW(Addr)        dlpi_addr;  /* Base address of object */
    const char       *dlpi_name;  /* (Null-terminated) name of
                                     object */
    const ElfW(Phdr) *dlpi_phdr;  /* Pointer to array of
                                     ELF program headers
                                     for this object */
    ElfW(Half)        dlpi_phnum; /* # of items in dlpi_phdr */

    /* The following fields were added in glibc 2.4, after the first
       version of this structure was available.  Check the size
       argument passed to the dl_iterate_phdr callback to determine
       whether or not each later member is available.  */

    unsigned long long dlpi_adds;
                    /* Incremented when a new object may
                       have been added */
    unsigned long long dlpi_subs;
                    /* Incremented when an object may
                       have been removed */
    size_t dlpi_tls_modid;
                    /* If there is a PT_TLS segment, its module
                       ID as used in TLS relocations, else zero */
    void  *dlpi_tls_data;
                    /* The address of the calling thread's instance
                       of this module's PT_TLS segment, if it has
                       one and it has been allocated in the calling
                       thread, otherwise a null pointer */
};
```

(The *ElfW*() macro definition turns its argument into the name of an ELF data type suitable for the hardware architecture. For example, on a 32-bit platform, *ElfW(Addr)* yields the data type name *Elf32_Addr*. Further information on these types can be found in the <*elf.h*> and <*link.h*> header files.)

The *dlpi_addr* field indicates the base address of the shared object (i.e., the difference between the

virtual memory address of the shared object and the offset of that object in the file from which it was loaded). The *dlpi_name* field is a null-terminated string giving the pathname from which the shared object was loaded.

To understand the meaning of the *dlpi_phdr* and *dlpi_phnum* fields, we need to be aware that an ELF shared object consists of a number of segments, each of which has a corresponding program header describing the segment. The *dlpi_phdr* field is a pointer to an array of the program headers for this shared object. The *dlpi_phnum* field indicates the size of this array.

These program headers are structures of the following form:

```
typedef struct {
    Elf32_Word  p_type;    /* Segment type */
    Elf32_Off   p_offset;  /* Segment file offset */
    Elf32_Addr  p_vaddr;   /* Segment virtual address */
    Elf32_Addr  p_paddr;   /* Segment physical address */
    Elf32_Word  p_filesz;  /* Segment size in file */
    Elf32_Word  p_memsz;   /* Segment size in memory */
    Elf32_Word  p_flags;   /* Segment flags */
    Elf32_Word  p_align;   /* Segment alignment */
} Elf32_Phdr;
```

Note that we can calculate the location of a particular program header, *x*, in virtual memory using the formula:

```
addr == info->dlpi_addr + info->dlpi_phdr[x].p_vaddr;
```

Possible values for *p_type* include the following (see <*elf.h*> for further details):

```
#define PT_LOAD         1    /* Loadable program segment */
#define PT_DYNAMIC      2    /* Dynamic linking information */
#define PT_INTERP       3    /* Program interpreter */
#define PT_NOTE         4    /* Auxiliary information */
#define PT_SHLIB        5    /* Reserved */
#define PT_PHDR         6    /* Entry for header table itself */
#define PT_TLS          7    /* Thread-local storage segment */
#define PT_GNU_EH_FRAME 0x6474e550 /* GCC .eh_frame_hdr segment */
#define PT_GNU_STACK  0x6474e551 /* Indicates stack executability */
#define PT_GNU_RELRO  0x6474e552 /* Read-only after relocation */
```

RETURN VALUE

The **dl_iterate_phdr**() function returns whatever value was returned by the last call to *callback*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dl_iterate_phdr()	Thread safety	MT-Safe

VERSIONS

Various other systems provide a version of this function, although details of the returned *dl_phdr_info* structure differ. On the BSDs and Solaris, the structure includes the fields *dlpi_addr*, *dlpi_name*, *dlpi_phdr*, and *dlpi_phnum* in addition to other implementation-specific fields.

Future versions of the C library may add further fields to the *dl_phdr_info* structure; in that event, the *size* argument provides a mechanism for the callback function to discover whether it is running on a system with added fields.

STANDARDS

None.

HISTORY

glibc 2.2.4.

NOTES

The first object visited by *callback* is the main program. For the main program, the *dlpi_name* field will be an empty string.

EXAMPLES

The following program displays a list of pathnames of the shared objects it has loaded. For each shared object, the program lists some information (virtual address, size, flags, and type) for each of the objects ELF segments.

The following shell session demonstrates the output produced by the program on an x86-64 system. The first shared object for which output is displayed (where the name is an empty string) is the main program.

```
$ ./a.out
Name: "" (9 segments)
     0: [      0x400040; memsz:    1f8] flags: 0x5; PT_PHDR
     1: [      0x400238; memsz:     1c] flags: 0x4; PT_INTERP
     2: [      0x400000; memsz:    ac4] flags: 0x5; PT_LOAD
     3: [      0x600e10; memsz:    240] flags: 0x6; PT_LOAD
     4: [      0x600e28; memsz:    1d0] flags: 0x6; PT_DYNAMIC
     5: [      0x400254; memsz:     44] flags: 0x4; PT_NOTE
     6: [      0x400970; memsz:     3c] flags: 0x4; PT_GNU_EH_FRAME
     7: [         (nil); memsz:      0] flags: 0x6; PT_GNU_STACK
     8: [      0x600e10; memsz:    1f0] flags: 0x4; PT_GNU_RELRO
Name: "linux-vdso.so.1" (4 segments)
     0: [0x7ffc6edd1000; memsz:    e89] flags: 0x5; PT_LOAD
     1: [0x7ffc6edd1360; memsz:    110] flags: 0x4; PT_DYNAMIC
     2: [0x7ffc6edd17b0; memsz:     3c] flags: 0x4; PT_NOTE
     3: [0x7ffc6edd17ec; memsz:     3c] flags: 0x4; PT_GNU_EH_FRAME
Name: "/lib64/libc.so.6" (10 segments)
     0: [0x7f55712ce040; memsz:    230] flags: 0x5; PT_PHDR
     1: [0x7f557145b980; memsz:     1c] flags: 0x4; PT_INTERP
     2: [0x7f55712ce000; memsz: 1b6a5c] flags: 0x5; PT_LOAD
     3: [0x7f55716857a0; memsz:   9240] flags: 0x6; PT_LOAD
     4: [0x7f5571688b80; memsz:    1f0] flags: 0x6; PT_DYNAMIC
     5: [0x7f55712ce270; memsz:     44] flags: 0x4; PT_NOTE
     6: [0x7f55716857a0; memsz:     78] flags: 0x4; PT_TLS
     7: [0x7f557145b99c; memsz:   544c] flags: 0x4; PT_GNU_EH_FRAME
     8: [0x7f55712ce000; memsz:      0] flags: 0x6; PT_GNU_STACK
     9: [0x7f55716857a0; memsz:   3860] flags: 0x4; PT_GNU_RELRO
Name: "/lib64/ld-linux-x86-64.so.2" (7 segments)
     0: [0x7f557168f000; memsz:  20828] flags: 0x5; PT_LOAD
     1: [0x7f55718afba0; memsz:   15a8] flags: 0x6; PT_LOAD
     2: [0x7f55718afe10; memsz:    190] flags: 0x6; PT_DYNAMIC
     3: [0x7f557168f1c8; memsz:     24] flags: 0x4; PT_NOTE
     4: [0x7f55716acec4; memsz:    604] flags: 0x4; PT_GNU_EH_FRAME
     5: [0x7f557168f000; memsz:      0] flags: 0x6; PT_GNU_STACK
     6: [0x7f55718afba0; memsz:    460] flags: 0x4; PT_GNU_RELRO
```

Program source

```
#define _GNU_SOURCE
#include <link.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

static int
callback(struct dl_phdr_info *info, size_t size, void *data)
{
    char *type;
    int p_type;

    printf("Name: \"%s\" (%d segments)\n", info->dlpi_name,
```

```
                info->dlpi_phnum);

            for (size_t j = 0; j < info->dlpi_phnum; j++) {
                p_type = info->dlpi_phdr[j].p_type;
                type = (p_type == PT_LOAD) ? "PT_LOAD" :
                       (p_type == PT_DYNAMIC) ? "PT_DYNAMIC" :
                       (p_type == PT_INTERP) ? "PT_INTERP" :
                       (p_type == PT_NOTE) ? "PT_NOTE" :
                       (p_type == PT_INTERP) ? "PT_INTERP" :
                       (p_type == PT_PHDR) ? "PT_PHDR" :
                       (p_type == PT_TLS) ? "PT_TLS" :
                       (p_type == PT_GNU_EH_FRAME) ? "PT_GNU_EH_FRAME" :
                       (p_type == PT_GNU_STACK) ? "PT_GNU_STACK" :
                       (p_type == PT_GNU_RELRO) ? "PT_GNU_RELRO" : NULL;

                printf("    %2zu: [%14p; memsz:%7jx] flags: %#jx; ", j,
                       (void *) (info->dlpi_addr + info->dlpi_phdr[j].p_vaddr),
                       (uintmax_t) info->dlpi_phdr[j].p_memsz,
                       (uintmax_t) info->dlpi_phdr[j].p_flags);
                if (type != NULL)
                    printf("%s\n", type);
                else
                    printf("[other (%#x)]\n", p_type);
            }

            return 0;
        }

        int
        main(void)
        {
            dl_iterate_phdr(callback, NULL);

            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

ldd(1), *objdump*(1), *readelf* (1), *dladdr*(3), *dlopen*(3), *elf*(5), *ld.so*(8)

Executable and Linking Format Specification, available at various locations online.

NAME

dladdr, dladdr1 − translate address to symbolic information

LIBRARY

Dynamic linking library (*libdl*, *−ldl*)

SYNOPSIS

```
#define _GNU_SOURCE
#include <dlfcn.h>

int dladdr(const void *addr, Dl_info *info);
int dladdr1(const void *addr, Dl_info *info, void **extra_info,
      int flags);
```

DESCRIPTION

The function **dladdr**() determines whether the address specified in *addr* is located in one of the shared objects loaded by the calling application. If it is, then **dladdr**() returns information about the shared object and symbol that overlaps *addr*. This information is returned in a *Dl_info* structure:

```
typedef struct {
    const char *dli_fname;  /* Pathname of shared object that
                               contains address */
    void       *dli_fbase;  /* Base address at which shared
                               object is loaded */
    const char *dli_sname;  /* Name of symbol whose definition
                               overlaps addr */
    void       *dli_saddr;  /* Exact address of symbol named
                               in dli_sname */
} Dl_info;
```

If no symbol matching *addr* could be found, then *dli_sname* and *dli_saddr* are set to NULL.

The function **dladdr1**() is like **dladdr**(), but returns additional information via the argument *extra_info*. The information returned depends on the value specified in *flags*, which can have one of the following values:

RTLD_DL_LINKMAP

Obtain a pointer to the link map for the matched file. The *extra_info* argument points to a pointer to a *link_map* structure (i.e., *struct link_map* **), defined in <*link.h*> as:

```
struct link_map {
    ElfW(Addr) l_addr;  /* Difference between the
                           address in the ELF file and
                           the address in memory */
    char      *l_name;  /* Absolute pathname where
                           object was found */
    ElfW(Dyn) *l_ld;    /* Dynamic section of the
                           shared object */
    struct link_map *l_next, *l_prev;
                        /* Chain of loaded objects */

    /* Plus additional fields private to the
       implementation */
};
```

RTLD_DL_SYMENT

Obtain a pointer to the ELF symbol table entry of the matching symbol. The *extra_info* argument is a pointer to a symbol pointer: *const ElfW(Sym)* **. The *ElfW*() macro definition turns its argument into the name of an ELF data type suitable for the hardware architecture. For example, on a 64-bit platform, *ElfW(Sym)* yields the data type name *Elf64_Sym*, which is defined in <*elf.h*> as:

```
typedef struct  {
    Elf64_Word    st_name;     /* Symbol name */
    unsigned char st_info;     /* Symbol type and binding */
```

```
    unsigned char st_other;    /* Symbol visibility */
    Elf64_Section st_shndx;    /* Section index */
    Elf64_Addr    st_value;    /* Symbol value */
    Elf64_Xword   st_size;     /* Symbol size */
} Elf64_Sym;
```

The *st_name* field is an index into the string table.

The *st_info* field encodes the symbol's type and binding. The type can be extracted using the macro **ELF64_ST_TYPE(st_info)** (or **ELF32_ST_TYPE()** on 32-bit platforms), which yields one of the following values:

Value	Description
STT_NOTYPE	Symbol type is unspecified
STT_OBJECT	Symbol is a data object
STT_FUNC	Symbol is a code object
STT_SECTION	Symbol associated with a section
STT_FILE	Symbol's name is filename
STT_COMMON	Symbol is a common data object
STT_TLS	Symbol is thread-local data object
STT_GNU_IFUNC	Symbol is indirect code object

The symbol binding can be extracted from the *st_info* field using the macro **ELF64_ST_BIND(st_info)** (or **ELF32_ST_BIND()** on 32-bit platforms), which yields one of the following values:

Value	Description
STB_LOCAL	Local symbol
STB_GLOBAL	Global symbol
STB_WEAK	Weak symbol
STB_GNU_UNIQUE	Unique symbol

The *st_other* field contains the symbol's visibility, which can be extracted using the macro **ELF64_ST_VISIBILITY(st_info)** (or **ELF32_ST_VISIBILITY()** on 32-bit platforms), which yields one of the following values:

Value	Description
STV_DEFAULT	Default symbol visibility rules
STV_INTERNAL	Processor-specific hidden class
STV_HIDDEN	Symbol unavailable in other modules
STV_PROTECTED	Not preemptible, not exported

RETURN VALUE

On success, these functions return a nonzero value. If the address specified in *addr* could be matched to a shared object, but not to a symbol in the shared object, then the *info->dli_sname* and *info->dli_saddr* fields are set to NULL.

If the address specified in *addr* could not be matched to a shared object, then these functions return 0. In this case, an error message is *not* available via *dlerror*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dladdr(), **dladdr1**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

dladdr()
: glibc 2.0.

dladdr1()
: glibc 2.3.3.

Solaris.

BUGS

Sometimes, the function pointers you pass to **dladdr**() may surprise you. On some architectures (notably i386 and x86-64), *dli_fname* and *dli_fbase* may end up pointing back at the object from which you called **dladdr**(), even if the function used as an argument should come from a dynamically linked library.

The problem is that the function pointer will still be resolved at compile time, but merely point to the *plt* (Procedure Linkage Table) section of the original object (which dispatches the call after asking the dynamic linker to resolve the symbol). To work around this, you can try to compile the code to be position-independent: then, the compiler cannot prepare the pointer at compile time any more and *gcc*(1) will generate code that just loads the final symbol address from the *got* (Global Offset Table) at run time before passing it to **dladdr**().

SEE ALSO

dl_iterate_phdr(3), *dlinfo*(3), *dlopen*(3), *dlsym*(3), *ld.so*(8)

NAME

dlerror − obtain error diagnostic for functions in the dlopen API

LIBRARY

Dynamic linking library (*libdl*, *−ldl*)

SYNOPSIS

#include <dlfcn.h>

char *dlerror(void);

DESCRIPTION

The **dlerror**() function returns a human-readable, null-terminated string describing the most recent error that occurred from a call to one of the functions in the dlopen API since the last call to **dlerror**(). The returned string does *not* include a trailing newline.

dlerror() returns NULL if no errors have occurred since initialization or since it was last called.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dlerror()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0. POSIX.1-2001.

SunOS.

NOTES

The message returned by **dlerror**() may reside in a statically allocated buffer that is overwritten by subsequent **dlerror**() calls.

EXAMPLES

See *dlopen*(3).

SEE ALSO

dladdr(3), *dlinfo*(3), *dlopen*(3), *dlsym*(3)

NAME

dlinfo − obtain information about a dynamically loaded object

LIBRARY

Dynamic linking library (*libdl*, *−ldl*)

SYNOPSIS

#define _GNU_SOURCE
#include <link.h>
#include <dlfcn.h>

int dlinfo(void *restrict handle**, int** request**, void *restrict** info**);**

DESCRIPTION

The **dlinfo**() function obtains information about the dynamically loaded object referred to by *handle* (typically obtained by an earlier call to *dlopen*(3) or *dlmopen*(3)). The *request* argument specifies which information is to be returned. The *info* argument is a pointer to a buffer used to store information returned by the call; the type of this argument depends on *request*.

The following values are supported for *request* (with the corresponding type for *info* shown in parentheses):

RTLD_DI_LMID (*Lmid_t* *)
Obtain the ID of the link-map list (namespace) in which *handle* is loaded.

RTLD_DI_LINKMAP (*struct link_map* **)
Obtain a pointer to the *link_map* structure corresponding to *handle*. The *info* argument points to a pointer to a *link_map* structure, defined in <*link.h*> as:

```
struct link_map {
    ElfW(Addr) l_addr;  /* Difference between the
                           address in the ELF file and
                           the address in memory */
    char      *l_name;  /* Absolute pathname where
                           object was found */
    ElfW(Dyn) *l_ld;    /* Dynamic section of the
                           shared object */
    struct link_map *l_next, *l_prev;
                        /* Chain of loaded objects */

    /* Plus additional fields private to the
       implementation */
};
```

RTLD_DI_ORIGIN (*char* *)
Copy the pathname of the origin of the shared object corresponding to *handle* to the location pointed to by *info*.

RTLD_DI_SERINFO (*Dl_serinfo* *)
Obtain the library search paths for the shared object referred to by *handle*. The *info* argument is a pointer to a *Dl_serinfo* that contains the search paths. Because the number of search paths may vary, the size of the structure pointed to by *info* can vary. The **RTLD_DI_SERINFOSIZE** request described below allows applications to size the buffer suitably. The caller must perform the following steps:

(1) Use a **RTLD_DI_SERINFOSIZE** request to populate a *Dl_serinfo* structure with the size (*dls_size*) of the structure needed for the subsequent **RTLD_DI_SERINFO** request.

(2) Allocate a *Dl_serinfo* buffer of the correct size (*dls_size*).

(3) Use a further **RTLD_DI_SERINFOSIZE** request to populate the *dls_size* and *dls_cnt* fields of the buffer allocated in the previous step.

(4) Use a **RTLD_DI_SERINFO** to obtain the library search paths.

The *Dl_serinfo* structure is defined as follows:

```
typedef struct {
    size_t dls_size;           /* Size in bytes of
                                  the whole buffer */
    unsigned int dls_cnt;      /* Number of elements
                                  in 'dls_serpath' */
    Dl_serpath dls_serpath[1]; /* Actually longer,
                                  'dls_cnt' elements */
} Dl_serinfo;
```

Each of the *dls_serpath* elements in the above structure is a structure of the following form:

```
typedef struct {
    char *dls_name;            /* Name of library search
                                  path directory */
    unsigned int dls_flags;    /* Indicates where this
                                  directory came from */
} Dl_serpath;
```

The *dls_flags* field is currently unused, and always contains zero.

RTLD_DI_SERINFOSIZE (*Dl_serinfo **)
Populate the *dls_size* and *dls_cnt* fields of the *Dl_serinfo* structure pointed to by *info* with values suitable for allocating a buffer for use in a subsequent **RTLD_DI_SERINFO** request.

RTLD_DI_TLS_MODID (*size_t **, since glibc 2.4)
Obtain the module ID of this shared object's TLS (thread-local storage) segment, as used in TLS relocations. If this object does not define a TLS segment, zero is placed in **info*.

RTLD_DI_TLS_DATA (*void ***, since glibc 2.4)
Obtain a pointer to the calling thread's TLS block corresponding to this shared object's TLS segment. If this object does not define a PT_TLS segment, or if the calling thread has not allocated a block for it, NULL is placed in **info*.

RETURN VALUE

On success, **dlinfo**() returns 0. On failure, it returns −1; the cause of the error can be diagnosed using *dlerror*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dlinfo()	Thread safety	MT-Safe

VERSIONS

The sets of requests supported by the various implementations overlaps only partially.

STANDARDS

GNU.

HISTORY

glibc 2.3.3. Solaris.

EXAMPLES

The program below opens a shared objects using *dlopen*(3) and then uses the **RTLD_DI_SERINFOSIZE** and **RTLD_DI_SERINFO** requests to obtain the library search path list for the library. Here is an example of what we might see when running the program:

```
$ ./a.out /lib64/libm.so.6
dls_serpath[0].dls_name = /lib64
dls_serpath[1].dls_name = /usr/lib64
```

Program source

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <link.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int
main(int argc, char *argv[])
{
    void *handle;
    Dl_serinfo serinfo;
    Dl_serinfo *sip;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <libpath>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Obtain a handle for shared object specified on command line. */

    handle = dlopen(argv[1], RTLD_NOW);
    if (handle == NULL) {
        fprintf(stderr, "dlopen() failed: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    /* Discover the size of the buffer that we must pass to
       RTLD_DI_SERINFO. */

    if (dlinfo(handle, RTLD_DI_SERINFOSIZE, &serinfo) == -1) {
        fprintf(stderr, "RTLD_DI_SERINFOSIZE failed: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    /* Allocate the buffer for use with RTLD_DI_SERINFO. */

    sip = malloc(serinfo.dls_size);
    if (sip == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    /* Initialize the 'dls_size' and 'dls_cnt' fields in the newly
       allocated buffer. */

    if (dlinfo(handle, RTLD_DI_SERINFOSIZE, sip) == -1) {
        fprintf(stderr, "RTLD_DI_SERINFOSIZE failed: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    /* Fetch and print library search list. */

    if (dlinfo(handle, RTLD_DI_SERINFO, sip) == -1) {
        fprintf(stderr, "RTLD_DI_SERINFO failed: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    for (size_t j = 0; j < serinfo.dls_cnt; j++)
        printf("dls_serpath[%zu].dls_name = %s\n",
               j, sip->dls_serpath[j].dls_name);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

dl_iterate_phdr(3), *dladdr*(3), *dlerror*(3), *dlopen*(3), *dlsym*(3), *ld.so*(8)

NAME

dlclose, dlopen, dlmopen − open and close a shared object

LIBRARY

Dynamic linking library (*libdl*, *−ldl*)

SYNOPSIS

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flags);
int dlclose(void *handle);

#define _GNU_SOURCE
#include <dlfcn.h>

void *dlmopen(Lmid_t lmid, const char *filename, int flags);
```

DESCRIPTION

dlopen()

The function **dlopen**() loads the dynamic shared object (shared library) file named by the null-terminated string *filename* and returns an opaque "handle" for the loaded object. This handle is employed with other functions in the dlopen API, such as *dlsym*(3), *dladdr*(3), *dlinfo*(3), and **dlclose**().

If *filename* is NULL, then the returned handle is for the main program. If *filename* contains a slash ("/"), then it is interpreted as a (relative or absolute) pathname. Otherwise, the dynamic linker searches for the object as follows (see *ld.so*(8) for further details):

- (ELF only) If the calling object (i.e., the shared library or executable from which **dlopen**() is called) contains a DT_RPATH tag, and does not contain a DT_RUNPATH tag, then the directories listed in the DT_RPATH tag are searched.
- If, at the time that the program was started, the environment variable **LD_LIBRARY_PATH** was defined to contain a colon-separated list of directories, then these are searched. (As a security measure, this variable is ignored for set-user-ID and set-group-ID programs.)
- (ELF only) If the calling object contains a DT_RUNPATH tag, then the directories listed in that tag are searched.
- The cache file */etc/ld.so.cache* (maintained by *ldconfig*(8)) is checked to see whether it contains an entry for *filename*.
- The directories */lib* and */usr/lib* are searched (in that order).

If the object specified by *filename* has dependencies on other shared objects, then these are also automatically loaded by the dynamic linker using the same rules. (This process may occur recursively, if those objects in turn have dependencies, and so on.)

One of the following two values must be included in *flags*:

RTLD_LAZY
: Perform lazy binding. Resolve symbols only as the code that references them is executed. If the symbol is never referenced, then it is never resolved. (Lazy binding is performed only for function references; references to variables are always immediately bound when the shared object is loaded.) Since glibc 2.1.1, this flag is overridden by the effect of the **LD_BIND_NOW** environment variable.

RTLD_NOW
: If this value is specified, or the environment variable **LD_BIND_NOW** is set to a nonempty string, all undefined symbols in the shared object are resolved before **dlopen**() returns. If this cannot be done, an error is returned.

Zero or more of the following values may also be ORed in *flags*:

RTLD_GLOBAL
: The symbols defined by this shared object will be made available for symbol resolution of subsequently loaded shared objects.

RTLD_LOCAL
: This is the converse of **RTLD_GLOBAL**, and the default if neither flag is specified. Symbols defined in this shared object are not made available to resolve references in subsequently

loaded shared objects.

RTLD_NODELETE (since glibc 2.2)
Do not unload the shared object during **dlclose**(). Consequently, the object's static and global variables are not reinitialized if the object is reloaded with **dlopen**() at a later time.

RTLD_NOLOAD (since glibc 2.2)
Don't load the shared object. This can be used to test if the object is already resident (**dlopen**() returns NULL if it is not, or the object's handle if it is resident). This flag can also be used to promote the flags on a shared object that is already loaded. For example, a shared object that was previously loaded with **RTLD_LOCAL** can be reopened with **RTLD_NOLOAD | RTLD_GLOBAL**.

RTLD_DEEPBIND (since glibc 2.3.4)
Place the lookup scope of the symbols in this shared object ahead of the global scope. This means that a self-contained object will use its own symbols in preference to global symbols with the same name contained in objects that have already been loaded.

If *filename* is NULL, then the returned handle is for the main program. When given to *dlsym*(3), this handle causes a search for a symbol in the main program, followed by all shared objects loaded at program startup, and then all shared objects loaded by **dlopen**() with the flag **RTLD_GLOBAL**.

Symbol references in the shared object are resolved using (in order): symbols in the link map of objects loaded for the main program and its dependencies; symbols in shared objects (and their dependencies) that were previously opened with **dlopen**() using the **RTLD_GLOBAL** flag; and definitions in the shared object itself (and any dependencies that were loaded for that object).

Any global symbols in the executable that were placed into its dynamic symbol table by *ld*(1) can also be used to resolve references in a dynamically loaded shared object. Symbols may be placed in the dynamic symbol table either because the executable was linked with the flag "−rdynamic" (or, synonymously, "−−export−dynamic"), which causes all of the executable's global symbols to be placed in the dynamic symbol table, or because *ld*(1) noted a dependency on a symbol in another object during static linking.

If the same shared object is opened again with **dlopen**(), the same object handle is returned. The dynamic linker maintains reference counts for object handles, so a dynamically loaded shared object is not deallocated until **dlclose**() has been called on it as many times as **dlopen**() has succeeded on it. Constructors (see below) are called only when the object is actually loaded into memory (i.e., when the reference count increases to 1).

A subsequent **dlopen**() call that loads the same shared object with **RTLD_NOW** may force symbol resolution for a shared object earlier loaded with **RTLD_LAZY**. Similarly, an object that was previously opened with **RTLD_LOCAL** can be promoted to **RTLD_GLOBAL** in a subsequent **dlopen**().

If **dlopen**() fails for any reason, it returns NULL.

dlmopen()
This function performs the same task as **dlopen**()—the *filename* and *flags* arguments, as well as the return value, are the same, except for the differences noted below.

The **dlmopen**() function differs from **dlopen**() primarily in that it accepts an additional argument, *lmid*, that specifies the link-map list (also referred to as a *namespace*) in which the shared object should be loaded. (By comparison, **dlopen**() adds the dynamically loaded shared object to the same namespace as the shared object from which the **dlopen**() call is made.) The *Lmid_t* type is an opaque handle that refers to a namespace.

The *lmid* argument is either the ID of an existing namespace (which can be obtained using the *dlinfo*(3) **RTLD_DI_LMID** request) or one of the following special values:

LM_ID_BASE
Load the shared object in the initial namespace (i.e., the application's namespace).

LM_ID_NEWLM
Create a new namespace and load the shared object in that namespace. The object must have been correctly linked to reference all of the other shared objects that it requires, since the new namespace is initially empty.

If *filename* is NULL, then the only permitted value for *lmid* is **LM_ID_BASE**.

dlclose()

The function **dlclose**() decrements the reference count on the dynamically loaded shared object referred to by *handle*.

If the object's reference count drops to zero and no symbols in this object are required by other objects, then the object is unloaded after first calling any destructors defined for the object. (Symbols in this object might be required in another object because this object was opened with the **RTLD_GLOBAL** flag and one of its symbols satisfied a relocation in another object.)

All shared objects that were automatically loaded when **dlopen**() was invoked on the object referred to by *handle* are recursively closed in the same manner.

A successful return from **dlclose**() does not guarantee that the symbols associated with *handle* are removed from the caller's address space. In addition to references resulting from explicit **dlopen**() calls, a shared object may have been implicitly loaded (and reference counted) because of dependencies in other shared objects. Only when all references have been released can the shared object be removed from the address space.

RETURN VALUE

On success, **dlopen**() and **dlmopen**() return a non-NULL handle for the loaded object. On error (file could not be found, was not readable, had the wrong format, or caused errors during loading), these functions return NULL.

On success, **dlclose**() returns 0; on error, it returns a nonzero value.

Errors from these functions can be diagnosed using *dlerror*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dlopen(), **dlmopen**(), **dlclose**()	Thread safety	MT-Safe

STANDARDS

dlopen()
dlclose()
POSIX.1-2008.

dlmopen()
RTLD_NOLOAD
RTLD_NODELETE
GNU.

RTLD_DEEPBIND
Solaris.

HISTORY

dlopen()
dlclose()
glibc 2.0. POSIX.1-2001.

dlmopen()
glibc 2.3.4.

NOTES

dlmopen() and namespaces

A link-map list defines an isolated namespace for the resolution of symbols by the dynamic linker. Within a namespace, dependent shared objects are implicitly loaded according to the usual rules, and symbol references are likewise resolved according to the usual rules, but such resolution is confined to the definitions provided by the objects that have been (explicitly and implicitly) loaded into the namespace.

The **dlmopen**() function permits object-load isolation—the ability to load a shared object in a new namespace without exposing the rest of the application to the symbols made available by the new object. Note that the use of the **RTLD_LOCAL** flag is not sufficient for this purpose, since it prevents a shared object's symbols from being available to *any* other shared object. In some cases, we may want to make the symbols provided by a dynamically loaded shared object available to (a subset of) other shared objects without exposing those symbols to the entire application. This can be achieved by using

a separate namespace and the **RTLD_GLOBAL** flag.

The **dlmopen**() function also can be used to provide better isolation than the **RTLD_LOCAL** flag. In particular, shared objects loaded with **RTLD_LOCAL** may be promoted to **RTLD_GLOBAL** if they are dependencies of another shared object loaded with **RTLD_GLOBAL**. Thus, **RTLD_LOCAL** is insufficient to isolate a loaded shared object except in the (uncommon) case where one has explicit control over all shared object dependencies.

Possible uses of **dlmopen**() are plugins where the author of the plugin-loading framework can't trust the plugin authors and does not wish any undefined symbols from the plugin framework to be resolved to plugin symbols. Another use is to load the same object more than once. Without the use of **dlmopen**(), this would require the creation of distinct copies of the shared object file. Using **dlmopen**(), this can be achieved by loading the same shared object file into different namespaces.

The glibc implementation supports a maximum of 16 namespaces.

Initialization and finalization functions

Shared objects may export functions using the **__attribute__((constructor))** and **__attribute__((destructor))** function attributes. Constructor functions are executed before **dlopen**() returns, and destructor functions are executed before **dlclose**() returns. A shared object may export multiple constructors and destructors, and priorities can be associated with each function to determine the order in which they are executed. See the **gcc** info pages (under "Function attributes") for further information.

An older method of (partially) achieving the same result is via the use of two special symbols recognized by the linker: **_init** and **_fini**. If a dynamically loaded shared object exports a routine named **_init**(), then that code is executed after loading a shared object, before **dlopen**() returns. If the shared object exports a routine named **_fini**(), then that routine is called just before the object is unloaded. In this case, one must avoid linking against the system startup files, which contain default versions of these files; this can be done by using the *gcc*(1) *−nostartfiles* command-line option.

Use of **_init** and **_fini** is now deprecated in favor of the aforementioned constructors and destructors, which among other advantages, permit multiple initialization and finalization functions to be defined.

Since glibc 2.2.3, *atexit*(3) can be used to register an exit handler that is automatically called when a shared object is unloaded.

History

These functions are part of the dlopen API, derived from SunOS.

BUGS

As at glibc 2.24, specifying the **RTLD_GLOBAL** flag when calling **dlmopen**() generates an error. Furthermore, specifying **RTLD_GLOBAL** when calling **dlopen**() results in a program crash (**SIGSEGV**) if the call is made from any object loaded in a namespace other than the initial namespace.

EXAMPLES

The program below loads the (glibc) math library, looks up the address of the *cos*(3) function, and prints the cosine of 2.0. The following is an example of building and running the program:

```
$ cc dlopen_demo.c -ldl
$ ./a.out
-0.416147
```

Program source

```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

#include <gnu/lib-names.h>  /* Defines LIBM_SO (which will be a
                               string such as "libm.so.6") */
int
main(void)
{
    void *handle;
    double (*cosine)(double);
```

```
        char *error;

        handle = dlopen(LIBM_SO, RTLD_LAZY);
        if (!handle) {
            fprintf(stderr, "%s\n", dlerror());
            exit(EXIT_FAILURE);
        }

        dlerror();    /* Clear any existing error */

        cosine = (double (*)(double)) dlsym(handle, "cos");

        /* According to the ISO C standard, casting between function
           pointers and 'void *', as done above, produces undefined results.
           POSIX.1-2001 and POSIX.1-2008 accepted this state of affairs and
           proposed the following workaround:

               *(void **) (&cosine) = dlsym(handle, "cos");

           This (clumsy) cast conforms with the ISO C standard and will
           avoid any compiler warnings.

           The 2013 Technical Corrigendum 1 to POSIX.1-2008 improved matters
           by requiring that conforming implementations support casting
           'void *' to a function pointer.  Nevertheless, some compilers
           (e.g., gcc with the '-pedantic' option) may complain about the
           cast used in this program. */

        error = dlerror();
        if (error != NULL) {
            fprintf(stderr, "%s\n", error);
            exit(EXIT_FAILURE);
        }

        printf("%f\n", (*cosine)(2.0));
        dlclose(handle);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ld(1), *ldd*(1), *pldd*(1), *dl_iterate_phdr*(3), *dladdr*(3), *dlerror*(3), *dlinfo*(3), *dlsym*(3), *rtld-audit*(7), *ld.so*(8), *ldconfig*(8)

gcc info pages, ld info pages

NAME

dlsym, dlvsym − obtain address of a symbol in a shared object or executable

LIBRARY

Dynamic linking library (*libdl*, *−ldl*)

SYNOPSIS

```
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict symbol);

#define _GNU_SOURCE
#include <dlfcn.h>

void *dlvsym(void *restrict handle, const char *restrict symbol,
        const char *restrict version);
```

DESCRIPTION

The function **dlsym**() takes a "handle" of a dynamic loaded shared object returned by *dlopen*(3) along with a null-terminated symbol name, and returns the address where that symbol is loaded into memory. If the symbol is not found, in the specified object or any of the shared objects that were automatically loaded by *dlopen*(3) when that object was loaded, **dlsym**() returns NULL. (The search performed by **dlsym**() is breadth first through the dependency tree of these shared objects.)

In unusual cases (see NOTES) the value of the symbol could actually be NULL. Therefore, a NULL return from **dlsym**() need not indicate an error. The correct way to distinguish an error from a symbol whose value is NULL is to call *dlerror*(3) to clear any old error conditions, then call **dlsym**(), and then call *dlerror*(3) again, saving its return value into a variable, and check whether this saved value is not NULL.

There are two special pseudo-handles that may be specified in *handle*:

RTLD_DEFAULT
: Find the first occurrence of the desired symbol using the default shared object search order. The search will include global symbols in the executable and its dependencies, as well as symbols in shared objects that were dynamically loaded with the **RTLD_GLOBAL** flag.

RTLD_NEXT
: Find the next occurrence of the desired symbol in the search order after the current object. This allows one to provide a wrapper around a function in another shared object, so that, for example, the definition of a function in a preloaded shared object (see **LD_PRELOAD** in *ld.so*(8)) can find and invoke the "real" function provided in another shared object (or for that matter, the "next" definition of the function in cases where there are multiple layers of preloading).

The **_GNU_SOURCE** feature test macro must be defined in order to obtain the definitions of **RTLD_DEFAULT** and **RTLD_NEXT** from <*dlfcn.h*>.

The function **dlvsym**() does the same as **dlsym**() but takes a version string as an additional argument.

RETURN VALUE

On success, these functions return the address associated with *symbol*. On failure, they return NULL; the cause of the error can be diagnosed using *dlerror*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dlsym(), **dlvsym**()	Thread safety	MT-Safe

STANDARDS

dlsym()
: POSIX.1-2008.

dlvsym()
: GNU.

HISTORY

dlsym()
glibc 2.0. POSIX.1-2001.

dlvsym()
glibc 2.1.

NOTES

There are several scenarios when the address of a global symbol is NULL. For example, a symbol can be placed at zero address by the linker, via a linker script or with −−*defsym* command-line option. Undefined weak symbols also have NULL value. Finally, the symbol value may be the result of a GNU indirect function (IFUNC) resolver function that returns NULL as the resolved value. In the latter case, **dlsym**() also returns NULL without error. However, in the former two cases, the behavior of GNU dynamic linker is inconsistent: relocation processing succeeds and the symbol can be observed to have NULL value, but **dlsym**() fails and **dlerror**() indicates a lookup error.

History

The **dlsym**() function is part of the dlopen API, derived from SunOS. That system does not have **dlvsym**().

EXAMPLES

See *dlopen*(3).

SEE ALSO

dl_iterate_phdr(3), *dladdr*(3), *dlerror*(3), *dlinfo*(3), *dlopen*(3), *ld.so*(8)

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 − generate uniformly distributed pseudo-random numbers

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short xsubi[3]);

long lrand48(void);
long nrand48(unsigned short xsubi[3]);

long mrand48(void);
long jrand48(unsigned short xsubi[3]);

void srand48(long seedval);
unsigned short *seed48(unsigned short seed16v[3]);
void lcong48(unsigned short param[7]);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
_XOPEN_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

DESCRIPTION

These functions generate pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48**() and **erand48**() functions return nonnegative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

The **lrand48**() and **nrand48**() functions return nonnegative long integers uniformly distributed over the interval [0, 2^31).

The **mrand48**() and **jrand48**() functions return signed long integers uniformly distributed over the interval [−2^31, 2^31).

The **srand48**(), **seed48**(), and **lcong48**() functions are initialization functions, one of which should be called before using **drand48**(), **lrand48**(), or **mrand48**(). The functions **erand48**(), **nrand48**(), and **jrand48**() do not require an initialization function to be called first.

All the functions work by generating a sequence of 48-bit integers, *Xi*, according to the linear congruential formula:

```
Xn+1 = (aXn + c) mod m, where n >= 0
```

The parameter *m* = 2^48, hence 48-bit integer arithmetic is performed. Unless **lcong48**() is called, *a* and *c* are given by:

```
a = 0x5DEECE66D
c = 0xB
```

The value returned by any of the functions **drand48**(), **erand48**(), **lrand48**(), **nrand48**(), **mrand48**(), or **jrand48**() is computed by first generating the next 48-bit *Xi* in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, is copied from the high-order bits of *Xi* and transformed into the returned value.

The functions **drand48**(), **lrand48**(), and **mrand48**() store the last 48-bit *Xi* generated in an internal buffer. The functions **erand48**(), **nrand48**(), and **jrand48**() require the calling program to provide storage for the successive *Xi* values in the array argument *xsubi*. The functions are initialized by placing the initial value of *Xi* into the array before calling the function for the first time.

The initializer function **srand48**() sets the high order 32-bits of *Xi* to the argument *seedval*. The low order 16-bits are set to the arbitrary value 0x330E.

The initializer function **seed48**() sets the value of *Xi* to the 48-bit value specified in the array argument *seed16v*. The previous value of *Xi* is copied into an internal buffer and a pointer to this buffer is returned by **seed48**().

The initialization function **lcong48**() allows the user to specify initial values for *Xi*, *a*, and *c*. Array argument elements *param[0−2]* specify *Xi*, *param[3−5]* specify *a*, and *param[6]* specifies *c*. After **lcong48**() has been called, a subsequent call to either **srand48**() or **seed48**() will restore the standard values of *a* and *c*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
drand48(), **erand48**(), **lrand48**(), **nrand48**(), **mrand48**(), **jrand48**(), **srand48**(), **seed48**(), **lcong48**()	Thread safety	MT-Unsafe race:drand48

The above functions record global state information for the random number generator, so they are not thread-safe.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

SEE ALSO

rand(3), *random*(3)

NAME

drand48_r, erand48_r, lrand48_r, nrand48_r, mrand48_r, jrand48_r, srand48_r, seed48_r, lcong48_r − generate uniformly distributed pseudo-random numbers reentrantly

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int drand48_r(struct drand48_data *restrict buffer,
              double *restrict result);
int erand48_r(unsigned short xsubi[3],
              struct drand48_data *restrict buffer,
              double *restrict result);

int lrand48_r(struct drand48_data *restrict buffer,
              long *restrict result);
int nrand48_r(unsigned short xsubi[3],
              struct drand48_data *restrict buffer,
              long *restrict result);

int mrand48_r(struct drand48_data *restrict buffer,
              long *restrict result);
int jrand48_r(unsigned short xsubi[3],
              struct drand48_data *restrict buffer,
              long *restrict result);

int srand48_r(long int seedval, struct drand48_data *buffer);
int seed48_r(unsigned short seed16v[3], struct drand48_data *buffer);
int lcong48_r(unsigned short param[7], struct drand48_data *buffer);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:

```
/* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

DESCRIPTION

These functions are the reentrant analogs of the functions described in *drand48*(3). Instead of modifying the global random generator state, they use the supplied data *buffer*.

Before the first use, this struct must be initialized, for example, by filling it with zeros, or by calling one of the functions **srand48_r**(), **seed48_r**(), or **lcong48_r**().

RETURN VALUE

The return value is 0.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
drand48_r(), **erand48_r**(), **lrand48_r**(), **nrand48_r**(), **mrand48_r**(), **jrand48_r**(), **srand48_r**(), **seed48_r**(), **lcong48_r**()	Thread safety	MT-Safe race:buffer

STANDARDS

GNU.

SEE ALSO

drand48(3), *rand*(3), *random*(3)

NAME

duplocale − duplicate a locale object

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <locale.h>

locale_t duplocale(locale_t *locobj***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

duplocale():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **duplocale**() function creates a duplicate of the locale object referred to by *locobj*.

If *locobj* is **LC_GLOBAL_LOCALE**, **duplocale**() creates a locale object containing a copy of the global locale determined by *setlocale*(3).

RETURN VALUE

On success, **duplocale**() returns a handle for the new locale object. On error, it returns *(locale_t) 0*, and sets *errno* to indicate the error.

ERRORS

ENOMEM
Insufficient memory to create the duplicate locale object.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.3.

NOTES

Duplicating a locale can serve the following purposes:

- To create a copy of a locale object in which one of more categories are to be modified (using *newlocale*(3)).
- To obtain a handle for the current locale which can used in other functions that employ a locale handle, such as *toupper_l*(3). This is done by applying **duplocale**() to the value returned by the following call:

  ```
  loc = uselocale((locale_t) 0);
  ```

  This technique is necessary, because the above *uselocale*(3) call may return the value **LC_GLOBAL_LOCALE**, which results in undefined behavior if passed to functions such as *toupper_l*(3). Calling **duplocale**() can be used to ensure that the **LC_GLOBAL_LOCALE** value is converted into a usable locale object. See EXAMPLES, below.

Each locale object created by **duplocale**() should be deallocated using *freelocale*(3).

EXAMPLES

The program below uses *uselocale*(3) and **duplocale**() to obtain a handle for the current locale which is then passed to *toupper_l*(3). The program takes one command-line argument, a string of characters that is converted to uppercase and displayed on standard output. An example of its use is the following:

```
$ ./a.out abc
ABC
```

Program source

```
#define _XOPEN_SOURCE 700
#include <ctype.h>
```

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    locale_t loc, nloc;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s string\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* This sequence is necessary, because uselocale() might return
       the value LC_GLOBAL_LOCALE, which can't be passed as an
       argument to toupper_l(). */

    loc = uselocale((locale_t) 0);
    if (loc == (locale_t) 0)
        errExit("uselocale");

    nloc = duplocale(loc);
    if (nloc == (locale_t) 0)
        errExit("duplocale");

    for (char *p = argv[1]; *p; p++)
        putchar(toupper_l(*p, nloc));

    printf("\n");

    freelocale(nloc);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

freelocale(3), *newlocale*(3), *setlocale*(3), *uselocale*(3), *locale*(5), *locale*(7)

NAME

dysize − get number of days for a given year

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <time.h>

int dysize(int *year***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

dysize():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The function returns 365 for a normal year and 366 for a leap year. The calculation for leap year is based on:

```
(year) %4 == 0 && ((year) %100 != 0 || (year) %400 == 0)
```

The formula is defined in the macro *__isleap(year)* also found in <*time.h*>.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
dysize()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

SunOS 4.x.

This is a compatibility function only. Don’t use it in new programs.

SEE ALSO

strftime(3)

NAME

ecvt, fcvt − convert a floating-point number to a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

[[deprecated]] char *ecvt(double number, int ndigits,
                          int *restrict decpt, int *restrict sign);
[[deprecated]] char *fcvt(double number, int ndigits,
                          int *restrict decpt, int *restrict sign);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ecvt(), **fcvt**():
- Since glibc 2.17
  - (_XOPEN_SOURCE >= 500 && ! (_POSIX_C_SOURCE >= 200809L))
    - || /* glibc >= 2.20 */ _DEFAULT_SOURCE
    - || /* glibc <= 2.19 */ _SVID_SOURCE
- glibc 2.12 to glibc 2.16:
  - (_XOPEN_SOURCE >= 500 && ! (_POSIX_C_SOURCE >= 200112L))
    - || _SVID_SOURCE
- Before glibc 2.12:
  - _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

The **ecvt**() function converts *number* to a null-terminated string of *ndigits* digits (where *ndigits* is reduced to a system-specific limit determined by the precision of a *double*), and returns a pointer to the string. The high-order digit is nonzero, unless *number* is zero. The low order digit is rounded. The string itself does not contain a decimal point; however, the position of the decimal point relative to the start of the string is stored in **decpt*. A negative value for **decpt* means that the decimal point is to the left of the start of the string. If the sign of *number* is negative, **sign* is set to a nonzero value, otherwise it is set to 0. If *number* is zero, it is unspecified whether **decpt* is 0 or 1.

The **fcvt**() function is identical to **ecvt**(), except that *ndigits* specifies the number of digits after the decimal point.

RETURN VALUE

Both the **ecvt**() and **fcvt**() functions return a pointer to a static string containing the ASCII representation of *number*. The static string is overwritten by each call to **ecvt**() or **fcvt**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ecvt()	Thread safety	MT-Unsafe race:ecvt
fcvt()	Thread safety	MT-Unsafe race:fcvt

STANDARDS

None.

HISTORY

SVr2; marked as LEGACY in POSIX.1-2001. POSIX.1-2008 removes the specifications of **ecvt**() and **fcvt**(), recommending the use of *sprintf*(3) instead (though *snprintf*(3) may be preferable).

NOTES

Not all locales use a point as the radix character ("decimal point").

SEE ALSO

ecvt_r(3), *gcvt*(3), *qecvt*(3), *setlocale*(3), *sprintf*(3)

NAME

ecvt_r, fcvt_r, qecvt_r, qfcvt_r − convert a floating-point number to a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

[[deprecated]] int ecvt_r(double number, int ndigits,
                          int *restrict decpt, int *restrict sign,
                          char *restrict buf, size_t len);
[[deprecated]] int fcvt_r(double number, int ndigits,
                          int *restrict decpt, int *restrict sign,
                          char *restrict buf, size_t len);

[[deprecated]] int qecvt_r(long double number, int ndigits,
                          int *restrict decpt, int *restrict sign,
                          char *restrict buf, size_t len);
[[deprecated]] int qfcvt_r(long double number, int ndigits,
                          int *restrict decpt, int *restrict sign,
                          char *restrict buf, size_t len);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ecvt_r(), **fcvt_r**(), **qecvt_r**(), **qfcvt_r**():

```
/* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

DESCRIPTION

The functions **ecvt_r**(), **fcvt_r**(), **qecvt_r**(), and **qfcvt_r**() are identical to *ecvt*(3), *fcvt*(3), *qecvt*(3), and *qfcvt*(3), respectively, except that they do not return their result in a static buffer, but instead use the supplied *buf* of size *len*. See *ecvt*(3) and *qecvt*(3).

RETURN VALUE

These functions return 0 on success, and −1 otherwise.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ecvt_r(), **fcvt_r**(), **qecvt_r**(), **qfcvt_r**()	Thread safety	MT-Safe

STANDARDS

GNU.

NOTES

These functions are obsolete. Instead, *sprintf*(3) is recommended.

SEE ALSO

ecvt(3), *qecvt*(3), *sprintf*(3)

NAME

encrypt, setkey, encrypt_r, setkey_r − encrypt 64-bit messages

LIBRARY

Password hashing library (*libcrypt*, *−lcrypt*)

SYNOPSIS

```
#define _XOPEN_SOURCE       /* See feature_test_macros(7) */
#include <unistd.h>

[[deprecated]] void encrypt(char block[64], int edflag);

#define _XOPEN_SOURCE       /* See feature_test_macros(7) */
#include <stdlib.h>

[[deprecated]] void setkey(const char *key);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <crypt.h>

[[deprecated]] void setkey_r(const char *key, struct crypt_data *data);
[[deprecated]] void encrypt_r(char *block, int edflag,
                              struct crypt_data *data);
```

DESCRIPTION

These functions encrypt and decrypt 64-bit messages. The **setkey**() function sets the key used by **encrypt**(). The *key* argument used here is an array of 64 bytes, each of which has numerical value 1 or 0. The bytes key[n] where n=8*i-1 are ignored, so that the effective key length is 56 bits.

The **encrypt**() function modifies the passed buffer, encoding if *edflag* is 0, and decoding if 1 is being passed. Like the *key* argument, also *block* is a bit vector representation of the actual value that is encoded. The result is returned in that same vector.

These two functions are not reentrant, that is, the key data is kept in static storage. The functions **setkey_r**() and **encrypt_r**() are the reentrant versions. They use the following structure to hold the key data:

```
struct crypt_data {
    char keysched[16 * 8];
    char sb0[32768];
    char sb1[32768];
    char sb2[32768];
    char sb3[32768];
    char crypt_3_buf[14];
    char current_salt[2];
    long current_saltbits;
    int  direction;
    int  initialized;
};
```

Before calling **setkey_r**() set *data−>initialized* to zero.

RETURN VALUE

These functions do not return any value.

ERRORS

Set *errno* to zero before calling the above functions. On success, *errno* is unchanged.

ENOSYS
The function is not provided. (For example because of former USA export restrictions.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
encrypt(), **setkey**()	Thread safety	MT-Unsafe race:crypt
encrypt_r(), **setkey_r**()	Thread safety	MT-Safe

STANDARDS

encrypt()
setkey()
POSIX.1-2008.

encrypt_r()
setkey_r()
None.

HISTORY

Removed in glibc 2.28.

Because they employ the DES block cipher, which is no longer considered secure, these functions were removed from glibc. Applications should switch to a modern cryptography library, such as **libgcrypt**.

encrypt()
setkey()
POSIX.1-2001, SUS, SVr4.

Availability in glibc

See *crypt*(3).

Features in glibc

In glibc 2.2, these functions use the DES algorithm.

EXAMPLES

```
#define _XOPEN_SOURCE
#include <crypt.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
    char key[64];
    char orig[9] = "eggplant";
    char buf[64];
    char txt[9];

    for (size_t i = 0; i < 64; i++) {
        key[i] = rand() & 1;
    }

    for (size_t i = 0; i < 8; i++) {
        for (size_t j = 0; j < 8; j++) {
            buf[i * 8 + j] = orig[i] >> j & 1;
        }
        setkey(key);
    }
    printf("Before encrypting: %s\n", orig);

    encrypt(buf, 0);
    for (size_t i = 0; i < 8; i++) {
        for (size_t j = 0, txt[i] = '\0'; j < 8; j++) {
            txt[i] |= buf[i * 8 + j] << j;
        }
        txt[8] = '\0';
    }
    printf("After encrypting:  %s\n", txt);

    encrypt(buf, 1);
    for (size_t i = 0; i < 8; i++) {
```

```
            for (size_t j = 0, txt[i] = '\0'; j < 8; j++) {
                txt[i] |= buf[i * 8 + j] << j;
            }
            txt[8] = '\0';
        }
        printf("After decrypting:  %s\n", txt);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

cbc_crypt(3), *crypt*(3), *ecb_crypt*(3)

NAME

etext, edata, end − end of program segments

SYNOPSIS

extern *etext***;**
extern *edata***;**
extern *end***;**

DESCRIPTION

The addresses of these symbols indicate the end of various program segments:

etext This is the first address past the end of the text segment (the program code).

edata This is the first address past the end of the initialized data segment.

end This is the first address past the end of the uninitialized data segment (also known as the BSS segment).

STANDARDS

None.

HISTORY

Although these symbols have long been provided on most UNIX systems, they are not standardized; use with caution.

NOTES

The program must explicitly declare these symbols; they are not defined in any header file.

On some systems the names of these symbols are preceded by underscores, thus: *_etext*, *_edata*, and *_end*. These symbols are also defined for programs compiled on Linux.

At the start of program execution, the program break will be somewhere near *&end* (perhaps at the start of the following page). However, the break will change as memory is allocated via *brk*(2) or *malloc*(3). Use *sbrk*(2) with an argument of zero to find the current value of the program break.

EXAMPLES

When run, the program below produces output such as the following:

```
$ ./a.out
First address past:
    program text (etext)       0x8048568
    initialized data (edata)   0x804a01c
    uninitialized data (end)   0x804a024
```

Program source

```
#include <stdio.h>
#include <stdlib.h>

extern char etext, edata, end; /* The symbols must have some type,
                                  or "gcc -Wall" complains */

int
main(void)
{
    printf("First address past:\n");
    printf("    program text (etext)      %10p\n", &etext);
    printf("    initialized data (edata)  %10p\n", &edata);
    printf("    uninitialized data (end)  %10p\n", &end);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

objdump(1), *readelf*(1), *sbrk*(2), *elf*(5)

NAME

htobe16, htole16, be16toh, le16toh, htobe32, htole32, be32toh, le32toh, htobe64, htole64, be64toh, le64toh − convert values between host and big-/little-endian byte order

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <endian.h>

uint16_t htobe16(uint16_t host_16bits);
uint16_t htole16(uint16_t host_16bits);
uint16_t be16toh(uint16_t big_endian_16bits);
uint16_t le16toh(uint16_t little_endian_16bits);

uint32_t htobe32(uint32_t host_32bits);
uint32_t htole32(uint32_t host_32bits);
uint32_t be32toh(uint32_t big_endian_32bits);
uint32_t le32toh(uint32_t little_endian_32bits);

uint64_t htobe64(uint64_t host_64bits);
uint64_t htole64(uint64_t host_64bits);
uint64_t be64toh(uint64_t big_endian_64bits);
uint64_t le64toh(uint64_t little_endian_64bits);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

htobe16(), **htole16**(), **be16toh**(), **le16toh**(), **htobe32**(), **htole32**(), **be32toh**(), **le32toh**(), **htobe64**(), **htole64**(), **be64toh**(), **le64toh**():

Since glibc 2.19:
_DEFAULT_SOURCE
In glibc up to and including 2.19:
_BSD_SOURCE

DESCRIPTION

These functions convert the byte encoding of integer values from the byte order that the current CPU (the "host") uses, to and from little-endian and big-endian byte order.

The number, *nn*, in the name of each function indicates the size of integer handled by the function, either 16, 32, or 64 bits.

The functions with names of the form "htobe*nn*" convert from host byte order to big-endian order.

The functions with names of the form "htole*nn*" convert from host byte order to little-endian order.

The functions with names of the form "be*nn*toh" convert from big-endian order to host byte order.

The functions with names of the form "le*nn*toh" convert from little-endian order to host byte order.

VERSIONS

Similar functions are present on the BSDs, where the required header file is <*sys/endian.h*> instead of <*endian.h*>. Unfortunately, NetBSD, FreeBSD, and glibc haven't followed the original OpenBSD naming convention for these functions, whereby the *nn* component always appears at the end of the function name (thus, for example, in NetBSD, FreeBSD, and glibc, the equivalent of OpenBSDs "betoh32" is "be32toh").

STANDARDS

None.

HISTORY

glibc 2.9.

These functions are similar to the older *byteorder*(3) family of functions. For example, **be32toh**() is identical to **ntohl**().

The advantage of the *byteorder*(3) functions is that they are standard functions available on all UNIX systems. On the other hand, the fact that they were designed for use in the context of TCP/IP means that they lack the 64-bit and little-endian variants described in this page.

EXAMPLES

The program below display the results of converting an integer from host byte order to both little-endian and big-endian byte order. Since host byte order is either little-endian or big-endian, only one of these conversions will have an effect. When we run this program on a little-endian system such as x86-32, we see the following:

```
$ ./a.out
x.u32 = 0x44332211
htole32(x.u32) = 0x44332211
htobe32(x.u32) = 0x11223344
```

Program source

```
#include <endian.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    union {
        uint32_t u32;
        uint8_t arr[4];
    } x;

    x.arr[0] = 0x11;  /* Lowest-address byte */
    x.arr[1] = 0x22;
    x.arr[2] = 0x33;
    x.arr[3] = 0x44;  /* Highest-address byte */

    printf("x.u32 = %#x\n", x.u32);
    printf("htole32(x.u32) = %#x\n", htole32(x.u32));
    printf("htobe32(x.u32) = %#x\n", htobe32(x.u32));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

bswap(3), *byteorder*(3)

NAME

envz_add, envz_entry, envz_get, envz_merge, envz_remove, envz_strip − environment string support

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <envz.h>

error_t envz_add(char **restrict envz, size_t *restrict envz_len,
        const char *restrict name, const char *restrict value);

char *envz_entry(const char *restrict envz, size_t envz_len,
        const char *restrict name);

char *envz_get(const char *restrict envz, size_t envz_len,
        const char *restrict name);

error_t envz_merge(char **restrict envz, size_t *restrict envz_len,
        const char *restrict envz2, size_t envz2_len,
        int override);

void envz_remove(char **restrict envz, size_t *restrict envz_len,
        const char *restrict name);

void envz_strip(char **restrict envz, size_t *restrict envz_len);
```

DESCRIPTION

These functions are glibc-specific.

An argz vector is a pointer to a character buffer together with a length, see *argz_add*(3). An envz vector is a special argz vector, namely one where the strings have the form "name=value". Everything after the first '=' is considered to be the value. If there is no '=', the value is taken to be NULL. (While the value in case of a trailing '=' is the empty string "".)

These functions are for handling envz vectors.

envz_add() adds the string "*name=value*" (in case *value* is non-NULL) or "*name*" (in case *value* is NULL) to the envz vector (**envz*, **envz_len*) and updates **envz* and **envz_len*. If an entry with the same *name* existed, it is removed.

envz_entry() looks for *name* in the envz vector (*envz*, *envz_len*) and returns the entry if found, or NULL if not.

envz_get() looks for *name* in the envz vector (*envz*, *envz_len*) and returns the value if found, or NULL if not. (Note that the value can also be NULL, namely when there is an entry for *name* without '=' sign.)

envz_merge() adds each entry in *envz2* to **envz*, as if with **envz_add**(). If *override* is true, then values in *envz2* will supersede those with the same name in **envz*, otherwise not.

envz_remove() removes the entry for *name* from (**envz*, **envz_len*) if there was one.

envz_strip() removes all entries with value NULL.

RETURN VALUE

All envz functions that do memory allocation have a return type of *error_t* (an integer type), and return 0 for success, and **ENOMEM** if an allocation error occurs.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
envz_add(), **envz_entry**(), **envz_get**(), **envz_merge**(), **envz_remove**(), **envz_strip**()	Thread safety	MT-Safe

STANDARDS

GNU.

EXAMPLES

```
#include <envz.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int
main(int argc, char *argv[], char *envp[])
{
    char    *str;
    size_t  e_len = 0;

    for (size_t i = 0; envp[i] != NULL; i++)
        e_len += strlen(envp[i]) + 1;

    str = envz_entry(*envp, e_len, "HOME");
    printf("%s\n", str);
    str = envz_get(*envp, e_len, "HOME");
    printf("%s\n", str);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

argz_add(3)

NAME

erf, erff, erfl − error function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double erf(double x);
float erff(float x);
long double erfl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

erf():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

erff(), **erfl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the error function of *x*, defined as

```
erf(x) = 2/sqrt(pi) * integral from 0 to x of exp(-t*t) dt
```

RETURN VALUE

On success, these functions return the value of the error function of *x*, a value in the range [−1, 1].

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity (negative infinity), +1 (−1) is returned.

If *x* is subnormal, a range error occurs, and the return value is 2*x/sqrt(pi).

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result underflow (*x* is subnormal)
An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
erf(), **erff**(), **erfl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

cerf (3), *erfc*(3), *exp*(3)

NAME

erfc, erfcf, erfcl − complementary error function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

erfc():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

erfcf(), **erfcl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the complementary error function of *x*, that is, 1.0 − erf(x).

RETURN VALUE

On success, these functions return the complementary error function of *x*, a value in the range [0,2].

If *x* is a NaN, a NaN is returned.

If *x* is +0 or −0, 1 is returned.

If *x* is positive infinity, +0 is returned.

If *x* is negative infinity, +2 is returned.

If the function result underflows and produces an unrepresentable value, the return value is 0.0.

If the function result underflows but produces a representable (i.e., subnormal) value, that value is returned, and a range error occurs.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result underflow (result is subnormal)
An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
erfc(), **erfcf**(), **erfcl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

NOTES

The **erfc**(), **erfcf**(), and **erfcl**() functions are provided to avoid the loss accuracy that would occur for the calculation 1-erf(x) for large values of *x* (for which the value of erf(x) approaches 1).

SEE ALSO

cerf (3), *erf*(3), *exp*(3)

NAME

err, verr, errx, verrx, warn, vwarn, warnx, vwarnx − formatted error messages

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <err.h>

[[noreturn]] void err(int eval, const char * fmt, ...);
[[noreturn]] void errx(int eval, const char * fmt, ...);

void warn(const char * fmt, ...);
void warnx(const char * fmt, ...);

#include <stdarg.h>

[[noreturn]] void verr(int eval, const char * fmt, va_list args);
[[noreturn]] void verrx(int eval, const char * fmt, va_list args);

void vwarn(const char * fmt, va_list args);
void vwarnx(const char * fmt, va_list args);
```

DESCRIPTION

The **err**() and **warn**() family of functions display a formatted error message on the standard error output. In all cases, the last component of the program name, a colon character, and a space are output. If the *fmt* argument is not NULL, the *printf*(3)-like formatted error message is output. The output is terminated by a newline character.

The **err**(), **verr**(), **warn**(), and **vwarn**() functions append an error message obtained from *strerror*(3) based on the global variable *errno*, preceded by another colon and space unless the *fmt* argument is NULL.

The **errx**() and **warnx**() functions do not append an error message.

The **err**(), **verr**(), **errx**(), and **verrx**() functions do not return, but exit with the value of the argument *eval*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
err(), **errx**(), **warn**(), **warnx**(), **verr**(), **verrx**(), **vwarn**(), **vwarnx**()	Thread safety	MT-Safe locale

STANDARDS

BSD.

HISTORY

err()
warn() 4.4BSD.

EXAMPLES

Display the current *errno* information string and exit:

```
p = malloc(size);
if (p == NULL)
    err(EXIT_FAILURE, NULL);
fd = open(file_name, O_RDONLY, 0);
if (fd == -1)
    err(EXIT_FAILURE, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
    errx(EXIT_FAILURE, "too early, wait until %s",
            start_time_string);
```

Warn of an error:

```
fd = open(raw_device, O_RDONLY, 0);
```

```
        if (fd == -1)
            warnx("%s: %s: trying the block device",
                    raw_device, strerror(errno));
        fd = open(block_device, O_RDONLY, 0);
        if (fd == -1)
            err(EXIT_FAILURE, "%s", block_device);
```

SEE ALSO

error(3), *exit*(3), *perror*(3), *printf*(3), *strerror*(3)

NAME

errno − number of last error

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <errno.h>

DESCRIPTION

The <*errno.h*> header file defines the integer variable *errno*, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

errno

The value in *errno* is significant only when the return value of the call indicated an error (i.e., −1 from most system calls; −1 or NULL from most library functions); a function that succeeds *is* allowed to change *errno*. The value of *errno* is never set to zero by any system call or library function.

For some system calls and library functions (e.g., *getpriority*(2)), −1 is a valid return on success. In such cases, a successful return can be distinguished from an error return by setting *errno* to zero before the call, and then, if the call returns a status that indicates that an error may have occurred, checking to see if *errno* has a nonzero value.

errno is defined by the ISO C standard to be a modifiable lvalue of type *int*, and must not be explicitly declared; *errno* may be a macro. *errno* is thread-local; setting it in one thread does not affect its value in any other thread.

Error numbers and names

Valid error numbers are all positive numbers. The <*errno.h*> header file defines symbolic names for each of the possible error numbers that may appear in *errno*.

All the error names specified by POSIX.1 must have distinct values, with the exception of **EAGAIN** and **EWOULDBLOCK**, which may be the same. On Linux, these two have the same value on all architectures.

The error numbers that correspond to each symbolic name vary across UNIX systems, and even across different architectures on Linux. Therefore, numeric values are not included as part of the list of error names below. The *perror*(3) and *strerror*(3) functions can be used to convert these names to corresponding textual error messages.

On any particular Linux system, one can obtain a list of all symbolic error names and the corresponding error numbers using the *errno*(1) command (part of the *moreutils* package):

```
$ errno -l
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
ESRCH 3 No such process
EINTR 4 Interrupted system call
EIO 5 Input/output error
...
```

The *errno*(1) command can also be used to look up individual error numbers and names, and to search for errors using strings from the error description, as in the following examples:

```
$ errno 2
ENOENT 2 No such file or directory
$ errno ESRCH
ESRCH 3 No such process
$ errno -s permission
EACCES 13 Permission denied
```

List of error names

In the list of the symbolic error names below, various names are marked as follows:

POSIX.1-2001

The name is defined by POSIX.1-2001, and is defined in later POSIX.1 versions, unless otherwise indicated.

POSIX.1-2008
The name is defined in POSIX.1-2008, but was not present in earlier POSIX.1 standards.

C99 The name is defined by C99.

Below is a list of the symbolic error names that are defined on Linux:

E2BIG Argument list too long (POSIX.1-2001).

EACCES Permission denied (POSIX.1-2001).

EADDRINUSE Address already in use (POSIX.1-2001).

EADDRNOTAVAIL
Address not available (POSIX.1-2001).

EAFNOSUPPORT
Address family not supported (POSIX.1-2001).

EAGAIN Resource temporarily unavailable (may be the same value as **EWOULDBLOCK**) (POSIX.1-2001).

EALREADY Connection already in progress (POSIX.1-2001).

EBADE Invalid exchange.

EBADF Bad file descriptor (POSIX.1-2001).

EBADFD File descriptor in bad state.

EBADMSG Bad message (POSIX.1-2001).

EBADR Invalid request descriptor.

EBADRQC Invalid request code.

EBADSLT Invalid slot.

EBUSY Device or resource busy (POSIX.1-2001).

ECANCELED Operation canceled (POSIX.1-2001).

ECHILD No child processes (POSIX.1-2001).

ECHRNG Channel number out of range.

ECOMM Communication error on send.

ECONNABORTED
Connection aborted (POSIX.1-2001).

ECONNREFUSED
Connection refused (POSIX.1-2001).

ECONNRESET Connection reset (POSIX.1-2001).

EDEADLK Resource deadlock avoided (POSIX.1-2001).

EDEADLOCK On most architectures, a synonym for **EDEADLK**. On some architectures (e.g., Linux MIPS, PowerPC, SPARC), it is a separate error code "File locking deadlock error".

EDESTADDRREQ
Destination address required (POSIX.1-2001).

EDOM Mathematics argument out of domain of function (POSIX.1, C99).

EDQUOT Disk quota exceeded (POSIX.1-2001).

EEXIST File exists (POSIX.1-2001).

EFAULT Bad address (POSIX.1-2001).

EFBIG File too large (POSIX.1-2001).

EHOSTDOWN Host is down.

EHOSTUNREACH
Host is unreachable (POSIX.1-2001).

EHWPOISON Memory page has hardware error.

EIDRM Identifier removed (POSIX.1-2001).

EILSEQ Invalid or incomplete multibyte or wide character (POSIX.1, C99).

The text shown here is the glibc error description; in POSIX.1, this error is described as "Illegal byte sequence".

EINPROGRESS Operation in progress (POSIX.1-2001).

EINTR Interrupted function call (POSIX.1-2001); see *signal*(7).

EINVAL Invalid argument (POSIX.1-2001).

EIO Input/output error (POSIX.1-2001).

EISCONN Socket is connected (POSIX.1-2001).

EISDIR Is a directory (POSIX.1-2001).

EISNAM Is a named type file.

EKEYEXPIRED Key has expired.

EKEYREJECTED
Key was rejected by service.

EKEYREVOKED
Key has been revoked.

EL2HLT Level 2 halted.

EL2NSYNC Level 2 not synchronized.

EL3HLT Level 3 halted.

EL3RST Level 3 reset.

ELIBACC Cannot access a needed shared library.

ELIBBAD Accessing a corrupted shared library.

ELIBMAX Attempting to link in too many shared libraries.

ELIBSCN .lib section in a.out corrupted

ELIBEXEC Cannot exec a shared library directly.

ELNRNG Link number out of range.

ELOOP Too many levels of symbolic links (POSIX.1-2001).

EMEDIUMTYPE
Wrong medium type.

EMFILE Too many open files (POSIX.1-2001). Commonly caused by exceeding the **RLIMIT_NOFILE** resource limit described in *getrlimit*(2). Can also be caused by exceeding the limit specified in */proc/sys/fs/nr_open*.

EMLINK Too many links (POSIX.1-2001).

EMSGSIZE Message too long (POSIX.1-2001).

EMULTIHOP Multihop attempted (POSIX.1-2001).

ENAMETOOLONG
Filename too long (POSIX.1-2001).

ENETDOWN Network is down (POSIX.1-2001).

ENETRESET Connection aborted by network (POSIX.1-2001).

ENETUNREACH
Network unreachable (POSIX.1-2001).

ENFILE Too many open files in system (POSIX.1-2001). On Linux, this is probably a result of encountering the */proc/sys/fs/file−max* limit (see *proc*(5)).

ENOANO No anode.

ENOBUFS No buffer space available (POSIX.1 (XSI STREAMS option)).

ENODATA The named attribute does not exist, or the process has no access to this attribute; see *xattr*(7).

In POSIX.1-2001 (XSI STREAMS option), this error was described as "No message is available on the STREAM head read queue".

ENODEV No such device (POSIX.1-2001).

ENOENT No such file or directory (POSIX.1-2001).

Typically, this error results when a specified pathname does not exist, or one of the components in the directory prefix of a pathname does not exist, or the specified pathname is a dangling symbolic link.

ENOEXEC Exec format error (POSIX.1-2001).

ENOKEY Required key not available.

ENOLCK No locks available (POSIX.1-2001).

ENOLINK Link has been severed (POSIX.1-2001).

ENOMEDIUM No medium found.

ENOMEM Not enough space/cannot allocate memory (POSIX.1-2001).

ENOMSG No message of the desired type (POSIX.1-2001).

ENONET Machine is not on the network.

ENOPKG Package not installed.

ENOPROTOOPT
Protocol not available (POSIX.1-2001).

ENOSPC No space left on device (POSIX.1-2001).

ENOSR No STREAM resources (POSIX.1 (XSI STREAMS option)).

ENOSTR Not a STREAM (POSIX.1 (XSI STREAMS option)).

ENOSYS Function not implemented (POSIX.1-2001).

ENOTBLK Block device required.

ENOTCONN The socket is not connected (POSIX.1-2001).

ENOTDIR Not a directory (POSIX.1-2001).

ENOTEMPTY Directory not empty (POSIX.1-2001).

ENOTRECOVERABLE
State not recoverable (POSIX.1-2008).

ENOTSOCK Not a socket (POSIX.1-2001).

ENOTSUP Operation not supported (POSIX.1-2001).

ENOTTY Inappropriate I/O control operation (POSIX.1-2001).

ENOTUNIQ Name not unique on network.

ENXIO No such device or address (POSIX.1-2001).

EOPNOTSUPP Operation not supported on socket (POSIX.1-2001).

(**ENOTSUP** and **EOPNOTSUPP** have the same value on Linux, but according to POSIX.1 these error values should be distinct.)

EOVERFLOW Value too large to be stored in data type (POSIX.1-2001).

EOWNERDEAD Owner died (POSIX.1-2008).

EPERM Operation not permitted (POSIX.1-2001).

EPFNOSUPPORT
Protocol family not supported.

EPIPE Broken pipe (POSIX.1-2001).

EPROTO Protocol error (POSIX.1-2001).

EPROTONOSUPPORT
Protocol not supported (POSIX.1-2001).

EPROTOTYPE Protocol wrong type for socket (POSIX.1-2001).

ERANGE Result too large (POSIX.1, C99).

EREMCHG Remote address changed.

EREMOTE Object is remote.

EREMOTEIO Remote I/O error.

ERESTART Interrupted system call should be restarted.

ERFKILL Operation not possible due to RF-kill.

EROFS Read-only filesystem (POSIX.1-2001).

ESHUTDOWN Cannot send after transport endpoint shutdown.

ESPIPE Invalid seek (POSIX.1-2001).

ESOCKTNOSUPPORT
Socket type not supported.

ESRCH No such process (POSIX.1-2001).

ESTALE Stale file handle (POSIX.1-2001).

This error can occur for NFS and for other filesystems.

ESTRPIPE Streams pipe error.

ETIME Timer expired (POSIX.1 (XSI STREAMS option)).

(POSIX.1 says "STREAM *ioctl*(2) timeout".)

ETIMEDOUT Connection timed out (POSIX.1-2001).

ETOOMANYREFS
Too many references: cannot splice.

ETXTBSY Text file busy (POSIX.1-2001).

EUCLEAN Structure needs cleaning.

EUNATCH Protocol driver not attached.

EUSERS Too many users.

EWOULDBLOCK
Operation would block (may be same value as **EAGAIN**) (POSIX.1-2001).

EXDEV Invalid cross-device link (POSIX.1-2001).

EXFULL Exchange full.

NOTES

A common mistake is to do

```
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

where *errno* no longer needs to have the value it had upon return from *somecall*() (i.e., it may have been changed by the *printf*(3)). If the value of *errno* should be preserved across a library call, it must

be saved:

```
if (somecall() == -1) {
    int errsv = errno;
    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}
```

Note that the POSIX threads APIs do *not* set *errno* on error. Instead, on failure they return an error number as the function result. These error numbers have the same meanings as the error numbers returned in *errno* by other APIs.

On some ancient systems, <*errno.h*> was not present or did not declare *errno*, so that it was necessary to declare *errno* manually (i.e., *extern int errno*). **Do not do this**. It long ago ceased to be necessary, and it will cause problems with modern versions of the C library.

SEE ALSO

errno(1), *err*(3), *error*(3), *perror*(3), *strerror*(3)

NAME

error, error_at_line, error_message_count, error_one_per_line, error_print_progname − glibc error reporting functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <error.h>

void error(int status, int errnum, const char * format, ...);
void error_at_line(int status, int errnum, const char * filename,
                   unsigned int linenum, const char * format, ...);

extern unsigned int error_message_count;
extern int error_one_per_line;

extern void (*error_print_progname)(void);
```

DESCRIPTION

error() is a general error-reporting function. It flushes *stdout*, and then outputs to *stderr* the program name, a colon and a space, the message specified by the *printf*(3)-style format string *format*, and, if *errnum* is nonzero, a second colon and a space followed by the string given by *strerror(errnum)*. Any arguments required for *format* should follow *format* in the argument list. The output is terminated by a newline character.

The program name printed by **error**() is the value of the global variable *program_invocation_name*(3). *program_invocation_name* initially has the same value as *main*()'s *argv[0]*. The value of this variable can be modified to change the output of **error**().

If *status* has a nonzero value, then **error**() calls *exit*(3) to terminate the program using the given value as the exit status; otherwise it returns after printing the error message.

The **error_at_line**() function is exactly the same as **error**(), except for the addition of the arguments *filename* and *linenum*. The output produced is as for **error**(), except that after the program name are written: a colon, the value of *filename*, a colon, and the value of *linenum*. The preprocessor values **__LINE__** and **__FILE__** may be useful when calling **error_at_line**(), but other values can also be used. For example, these arguments could refer to a location in an input file.

If the global variable *error_one_per_line* is set nonzero, a sequence of **error_at_line**() calls with the same value of *filename* and *linenum* will result in only one message (the first) being output.

The global variable *error_message_count* counts the number of messages that have been output by **error**() and **error_at_line**().

If the global variable *error_print_progname* is assigned the address of a function (i.e., is not NULL), then that function is called instead of prefixing the message with the program name and colon. The function should print a suitable string to *stderr*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
error()	Thread safety	MT-Safe locale
error_at_line()	Thread safety	MT-Unsafe race: error_at_line/error_one_per_line locale

The internal *error_one_per_line* variable is accessed (without any form of synchronization, but since it's an *int* used once, it should be safe enough) and, if *error_one_per_line* is set nonzero, the internal static variables (not exposed to users) used to hold the last printed filename and line number are accessed and modified without synchronization; the update is not atomic and it occurs before disabling cancelation, so it can be interrupted only after one of the two variables is modified. After that, **error_at_line**() is very much like **error**().

STANDARDS

GNU.

SEE ALSO

err(3), *errno*(3), *exit*(3), *perror*(3), *program_invocation_name*(3), *strerror*(3)

NAME

ether_aton, ether_ntoa, ether_ntohost, ether_hostton, ether_line, ether_ntoa_r, ether_aton_r − Ethernet address manipulation routines

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netinet/ether.h>

char *ether_ntoa(const struct ether_addr *addr);
struct ether_addr *ether_aton(const char *asc);

int ether_ntohost(char *hostname, const struct ether_addr *addr);
int ether_hostton(const char *hostname, struct ether_addr *addr);

int ether_line(const char *line, struct ether_addr *addr,
        char *hostname);

/* GNU extensions */
char *ether_ntoa_r(const struct ether_addr *addr, char *buf);

struct ether_addr *ether_aton_r(const char *asc,
                    struct ether_addr *addr);
```

DESCRIPTION

ether_aton() converts the 48-bit Ethernet host address *asc* from the standard hex-digits-and-colons notation into binary data in network byte order and returns a pointer to it in a statically allocated buffer, which subsequent calls will overwrite. **ether_aton**() returns NULL if the address is invalid.

The **ether_ntoa**() function converts the Ethernet host address *addr* given in network byte order to a string in standard hex-digits-and-colons notation, omitting leading zeros. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

The **ether_ntohost**() function maps an Ethernet address to the corresponding hostname in */etc/ethers* and returns nonzero if it cannot be found.

The **ether_hostton**() function maps a hostname to the corresponding Ethernet address in */etc/ethers* and returns nonzero if it cannot be found.

The **ether_line**() function parses a line in */etc/ethers* format (ethernet address followed by whitespace followed by hostname; '#' introduces a comment) and returns an address and hostname pair, or nonzero if it cannot be parsed. The buffer pointed to by *hostname* must be sufficiently long, for example, have the same length as *line*.

The functions **ether_ntoa_r**() and **ether_aton_r**() are reentrant thread-safe versions of **ether_ntoa**() and **ether_aton**() respectively, and do not use static buffers.

The structure *ether_addr* is defined in <*net/ethernet.h*> as:

```
struct ether_addr {
    uint8_t ether_addr_octet[6];
}
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ether_aton(), **ether_ntoa**()	Thread safety	MT-Unsafe
ether_ntohost(), **ether_hostton**(), **ether_line**(), **ether_ntoa_r**(), **ether_aton_r**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD, SunOS.

BUGS

In glibc 2.2.5 and earlier, the implementation of **ether_line**() is broken.

SEE ALSO

ethers(5)

NAME

euidaccess, eaccess − check effective user's permissions for a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>

int euidaccess(const char *pathname, int mode);
int eaccess(const char *pathname, int mode);
```

DESCRIPTION

Like *access*(2), **euidaccess**() checks permissions and existence of the file identified by its argument *pathname*. However, whereas *access*(2) performs checks using the real user and group identifiers of the process, **euidaccess**() uses the effective identifiers.

mode is a mask consisting of one or more of **R_OK**, **W_OK**, **X_OK**, and **F_OK**, with the same meanings as for *access*(2).

eaccess() is a synonym for **euidaccess**(), provided for compatibility with some other systems.

RETURN VALUE

On success (all requested permissions granted), zero is returned. On error (at least one bit in *mode* asked for a permission that is denied, or some other error occurred), −1 is returned, and *errno* is set to indicate the error.

ERRORS

As for *access*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
euidaccess(), **eaccess**()	Thread safety	MT-Safe

VERSIONS

Some other systems have an **eaccess**() function.

STANDARDS

None.

HISTORY

eaccess()
: glibc 2.4.

NOTES

Warning: Using this function to check a process's permissions on a file before performing some operation based on that information leads to race conditions: the file permissions may change between the two steps. Generally, it is safer just to attempt the desired operation and handle any permission error that occurs.

This function always dereferences symbolic links. If you need to check the permissions on a symbolic link, use *faccessat*(2) with the flags **AT_EACCESS** and **AT_SYMLINK_NOFOLLOW**.

SEE ALSO

access(2), *chmod*(2), *chown*(2), *faccessat*(2), *open*(2), *setgid*(2), *setuid*(2), *stat*(2), *credentials*(7), *path_resolution*(7)

NAME

execl, execlp, execle, execv, execvp, execvpe − execute a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execl(const char *pathname, const char *arg, ...
        /*, (char *) NULL */);
int execlp(const char *file, const char *arg, ...
        /*, (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
        /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

execvpe():
_GNU_SOURCE

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are layered on top of *execve*(2). (See the manual page for *execve*(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

l - execl(), execlp(), execle()

The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

v - execv(), execvp(), execvpe()

The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

e - execle(), execvpe()

The environment of the new process image is specified via the argument *envp*. The *envp* argument is an array of pointers to null-terminated strings and *must* be terminated by a null pointer.

All other **exec**() functions (which do not include 'e' in the suffix) take the environment for the new process image from the external variable *environ* in the calling process.

p - execlp(), execvp(), execvpe()

These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the **PATH** environment variable. If this variable isn't defined, the path list defaults to a list that includes the directories returned by *confstr(_CS_PATH)* (which typically returns the value "/bin:/usr/bin") and possibly also the current working directory; see NOTES for further details.

execvpe() searches for the program using the value of **PATH** from the caller's environment, not from the *envp* argument.

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified

pathname is executed.

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted *execve*(2) failed with the error **EACCES**), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with *errno* set to **EACCES**.

If the header of a file isn't recognized (the attempted *execve*(2) failed with the error **ENOEXEC**), these functions will execute the shell (*/bin/sh*) with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

All other **exec**() functions (which do not include 'p' in the suffix) take as their first argument a (relative or absolute) pathname that identifies the program to be executed.

RETURN VALUE

The **exec**() functions return only if an error has occurred. The return value is −1, and *errno* is set to indicate the error.

ERRORS

All of these functions may fail and set *errno* for any of the errors specified for *execve*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
execl(), **execle**(), **execv**()	Thread safety	MT-Safe
execlp(), **execvp**(), **execvpe**()	Thread safety	MT-Safe env

VERSIONS

The default search path (used when the environment does not contain the variable **PATH**) shows some variation across systems. It generally includes */bin* and */usr/bin* (in that order) and may also include the current working directory. On some other systems, the current working is included after */bin* and */usr/bin*, as an anti-Trojan-horse measure. The glibc implementation long followed the traditional default where the current working directory is included at the start of the search path. However, some code refactoring during the development of glibc 2.24 caused the current working directory to be dropped altogether from the default search path. This accidental behavior change is considered mildly beneficial, and won't be reverted.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is historic practice, but has not traditionally been documented and is not specified by the POSIX standard. BSD (and possibly other systems) do an automatic sleep and retry if **ETXTBSY** is encountered. Linux treats it as a hard error and returns immediately.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and **ENOMEM** and **E2BIG**, upon which they returned. They now return if any error other than the ones described above occurs.

STANDARDS

environ
execl()
execlp()
execle()
execv()
execvp()
: POSIX.1-2008.

execvpe()
: GNU.

HISTORY

environ
execl()
execlp()
execle()

execv()
execvp()
POSIX.1-2001.

execvpe()
glibc 2.11.

BUGS

Before glibc 2.24, **execl**() and **execle**() employed *realloc*(3) internally and were consequently not async-signal-safe, in violation of the requirements of POSIX.1. This was fixed in glibc 2.24.

Architecture-specific details

On sparc and sparc64, **execv**() is provided as a system call by the kernel (with the prototype shown above) for compatibility with SunOS. This function is *not* employed by the **execv**() wrapper function on those architectures.

SEE ALSO

sh(1), *execve*(2), *execveat*(2), *fork*(2), *ptrace*(2), *fexecve*(3), *system*(3), *environ*(7)

NAME

exit − cause normal process termination

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

[[noreturn]] void exit(int *status***);**

DESCRIPTION

The **exit**() function causes normal process termination and the least significant byte of *status* (i.e., *status & 0xFF*) is returned to the parent (see *wait*(2)).

All functions registered with *atexit*(3) and *on_exit*(3) are called, in the reverse order of their registration. (It is possible for one of these functions to use *atexit*(3) or *on_exit*(3) to register an additional function to be executed during exit processing; the new registration is added to the front of the list of functions that remain to be called.) If one of these functions does not return (e.g., it calls *_exit*(2), or kills itself with a signal), then none of the remaining functions is called, and further exit processing (in particular, flushing of *stdio*(3) streams) is abandoned. If a function has been registered multiple times using *atexit*(3) or *on_exit*(3), then it is called as many times as it was registered.

All open *stdio*(3) streams are flushed and closed. Files created by *tmpfile*(3) are removed.

The C standard specifies two constants, **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to **exit**() to indicate successful or unsuccessful termination, respectively.

RETURN VALUE

The **exit**() function does not return.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
exit()	Thread safety	MT-Unsafe race:exit

The **exit**() function uses a global variable that is not protected, so it is not thread-safe.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001, SVr4, 4.3BSD.

NOTES

The behavior is undefined if one of the functions registered using *atexit*(3) and *on_exit*(3) calls either **exit**() or *longjmp*(3). Note that a call to *execve*(2) removes registrations created using *atexit*(3) and *on_exit*(3).

The use of **EXIT_SUCCESS** and **EXIT_FAILURE** is slightly more portable (to non-UNIX environments) than the use of 0 and some nonzero value like 1 or −1. In particular, VMS uses a different convention.

BSD has attempted to standardize exit codes (which some C libraries such as the GNU C library have also adopted); see the file *<sysexits.h>*.

After **exit**(), the exit status must be transmitted to the parent process. There are three cases:

- If the parent has set **SA_NOCLDWAIT**, or has set the **SIGCHLD** handler to **SIG_IGN**, the status is discarded and the child dies immediately.
- If the parent was waiting on the child, it is notified of the exit status and the child dies immediately.
- Otherwise, the child becomes a "zombie" process: most of the process resources are recycled, but a slot containing minimal information about the child process (termination status, resource usage statistics) is retained in process table. This allows the parent to subsequently use *waitpid*(2) (or similar) to learn the termination status of the child; at that point the zombie process slot is released.

If the implementation supports the **SIGCHLD** signal, this signal is sent to the parent. If the parent has set **SA_NOCLDWAIT**, it is undefined whether a **SIGCHLD** signal is sent.

Signals sent to other processes

If the exiting process is a session leader and its controlling terminal is the controlling terminal of the session, then each process in the foreground process group of this controlling terminal is sent a **SIGHUP** signal, and the terminal is disassociated from this session, allowing it to be acquired by a new controlling process.

If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in this process group. See *setpgid*(2) for an explanation of orphaned process groups.

Except in the above cases, where the signalled processes may be children of the terminating process, termination of a process does *not* in general cause a signal to be sent to children of that process. However, a process can use the *prctl*(2) **PR_SET_PDEATHSIG** operation to arrange that it receives a signal if its parent terminates.

SEE ALSO

_exit(2), *get_robust_list*(2), *setpgid*(2), *wait*(2), *atexit*(3), *on_exit*(3), *tmpfile*(3)

NAME

exp, expf, expl − base-e exponential function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double exp(double x);
float expf(float x);
long double expl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

expf(), **expl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the value of e (the base of natural logarithms) raised to the power of *x*.

RETURN VALUE

On success, these functions return the exponential value of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is negative infinity, +0 is returned.

If the result underflows, a range error occurs, and zero is returned.

If the result overflows, a range error occurs, and the functions return +**HUGE_VAL**, +**HUGE_VALF**, or +**HUGE_VALL**, respectively.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error, overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error, underflow
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
exp(), **expf**(), **expl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

cbrt(3), *cexp*(3), *exp10*(3), *exp2*(3), *expm1*(3), *sqrt*(3)

NAME

exp2, exp2f, exp2l − base-2 exponential function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

exp2(), **exp2f**(), **exp2l**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return the value of 2 raised to the power of *x*.

RETURN VALUE

On success, these functions return the base-2 exponential value of *x*.

For various special cases, including the handling of infinity and NaN, as well as overflows and underflows, see *exp*(3).

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

For a discussion of the errors that can occur for these functions, see *exp*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
exp2(), **exp2f**(), **exp2l**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

cbrt(3), *cexp2*(3), *exp*(3), *exp10*(3), *sqrt*(3)

NAME

exp10, exp10f, exp10l − base-10 exponential function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <math.h>

double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
```

DESCRIPTION

These functions return the value of 10 raised to the power of *x*.

RETURN VALUE

On success, these functions return the base-10 exponential value of *x*.

For various special cases, including the handling of infinity and NaN, as well as overflows and underflows, see *exp*(3).

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

For a discussion of the errors that can occur for these functions, see *exp*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
exp10(), **exp10f**(), **exp10l**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

BUGS

Before glibc 2.19, the glibc implementation of these functions did not set *errno* to **ERANGE** when an underflow error occurred.

SEE ALSO

cbrt(3), *exp*(3), *exp2*(3), *log10*(3), *sqrt*(3)

NAME

expm1, expm1f, expm1l − exponential minus 1

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

expm1():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

expm1f(), **expm1l**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return a value equivalent to

exp(x) − 1

The result is computed in a way that is accurate even if the value of *x* is near zero—a case where *exp(x) − 1* would be inaccurate due to subtraction of two numbers that are nearly equal.

RETURN VALUE

On success, these functions return *exp(x) − 1*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is negative infinity, −1 is returned.

If the result overflows, a range error occurs, and the functions return **−HUGE_VAL**, **−HUGE_VALF**, or **−HUGE_VALL**, respectively.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error, overflow
errno is set to **ERANGE** (but see BUGS). An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
expm1(), **expm1f**(), **expm1l**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001. BSD.

BUGS

Before glibc 2.17, on certain architectures (e.g., x86, but not x86_64) **expm1**() raised a bogus underflow floating-point exception for some large negative *x* values (where the function result approaches

−1).

Before approximately glibc 2.11, **expm1**() raised a bogus invalid floating-point exception in addition to the expected overflow exception, and returned a NaN instead of positive infinity, for some large positive *x* values.

Before glibc 2.11, the glibc implementation did not set *errno* to **ERANGE** when a range error occurred.

SEE ALSO

exp(3), *log*(3), *log1p*(3)

NAME

fabs, fabsf, fabsl − absolute value of floating-point number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fabsf(), **fabsl**():

```
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

These functions return the absolute value of the floating-point number *x*.

RETURN VALUE

These functions return the absolute value of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is −0, +0 is returned.

If *x* is negative infinity or positive infinity, positive infinity is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fabs(), **fabsf**(), **fabsl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

abs(3), *cabs*(3), *ceil*(3), *floor*(3), *labs*(3), *rint*(3)

NAME

fclose − close a stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

int fclose(FILE *****stream*);**

DESCRIPTION

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using *fflush*(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error. In either case, any further access (including another call to *fclose*()) to the stream results in undefined behavior.

ERRORS

EBADF

The file descriptor underlying *stream* is not valid.

The **fclose**() function may also fail and set *errno* for any of the errors specified for the routines *close*(2), *write*(2), or *fflush*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fclose()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

NOTES

Note that **fclose**() flushes only the user-space buffers provided by the C library. To ensure that the data is physically stored on disk the kernel buffers must be flushed too, for example, with *sync*(2) or *fsync*(2).

SEE ALSO

close(2), *fcloseall*(3), *fflush*(3), *fileno*(3), *fopen*(3), *setbuf*(3)

NAME

fcloseall − close all open streams

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <stdio.h>

int fcloseall(void);
```

DESCRIPTION

The **fcloseall**() function closes all of the calling process's open streams. Buffered output for each stream is written before it is closed (as for *fflush*(3)); buffered input is discarded.

The standard streams, *stdin*, *stdout*, and *stderr* are also closed.

RETURN VALUE

This function returns 0 if all files were successfully closed; on error, **EOF** is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fcloseall()	Thread safety	MT-Unsafe race:streams

The **fcloseall**() function does not lock the streams, so it is not thread-safe.

STANDARDS

GNU.

SEE ALSO

close(2), *fclose*(3), *fflush*(3), *fopen*(3), *setbuf*(3)

NAME

fdim, fdimf, fdiml − positive difference

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fdimf(), **fdiml**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return the positive difference, max(*x*-*y*,0), between their arguments.

RETURN VALUE

On success, these functions return the positive difference.

If *x* or *y* is a NaN, a NaN is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fdim(), **fdimf**(), **fdiml**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

BUGS

Before glibc 2.24 on certain architectures (e.g., x86, but not x86_64) these functions did not set *errno*.

SEE ALSO

fmax(3)

NAME

feclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag, fetestexcept, fegetenv, fegetround, feholdexcept, fesetround, fesetenv, feupdateenv, feenableexcept, fedisableexcept, fegetexcept − floating-point rounding and exception handling

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <fenv.h>

int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);

int fegetround(void);
int fesetround(int rounding_mode);

int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

DESCRIPTION

These eleven functions were defined in C99, and describe the handling of floating-point rounding and exceptions (overflow, zero-divide, etc.).

Exceptions

The *divide-by-zero* exception occurs when an operation on finite numbers produces infinity as exact answer.

The *overflow* exception occurs when a result has to be represented as a floating-point number, but has (much) larger absolute value than the largest (finite) floating-point number that is representable.

The *underflow* exception occurs when a result has to be represented as a floating-point number, but has smaller absolute value than the smallest positive normalized floating-point number (and would lose much accuracy when represented as a denormalized number).

The *inexact* exception occurs when the rounded result of an operation is not equal to the infinite precision result. It may occur whenever *overflow* or *underflow* occurs.

The *invalid* exception occurs when there is no well-defined result for an operation, as for 0/0 or infinity − infinity or sqrt(−1).

Exception handling

Exceptions are represented in two ways: as a single bit (exception present/absent), and these bits correspond in some implementation-defined way with bit positions in an integer, and also as an opaque structure that may contain more information about the exception (perhaps the code address where it occurred).

Each of the macros **FE_DIVBYZERO**, **FE_INEXACT**, **FE_INVALID**, **FE_OVERFLOW**, **FE_UNDERFLOW** is defined when the implementation supports handling of the corresponding exception, and if so then defines the corresponding bit(s), so that one can call exception handling functions, for example, using the integer argument **FE_OVERFLOW**|**FE_UNDERFLOW**. Other exceptions may be supported. The macro **FE_ALL_EXCEPT** is the bitwise OR of all bits corresponding to supported exceptions.

The **feclearexcept**() function clears the supported exceptions represented by the bits in its argument.

The **fegetexceptflag**() function stores a representation of the state of the exception flags represented by the argument *excepts* in the opaque object **flagp*.

The **feraiseexcept**() function raises the supported exceptions represented by the bits in *excepts*.

The **fesetexceptflag**() function sets the complete status for the exceptions represented by *excepts* to the value **flagp*. This value must have been obtained by an earlier call of **fegetexceptflag**() with a last argument that contained all bits in *excepts*.

The **fetestexcept**() function returns a word in which the bits are set that were set in the argument *excepts* and for which the corresponding exception is currently set.

Rounding mode

The rounding mode determines how the result of floating-point operations is treated when the result cannot be exactly represented in the significand. Various rounding modes may be provided: round to nearest (the default), round up (toward positive infinity), round down (toward negative infinity), and round toward zero.

Each of the macros **FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, and **FE_TOWARDZERO** is defined when the implementation supports getting and setting the corresponding rounding direction.

The **fegetround**() function returns the macro corresponding to the current rounding mode.

The **fesetround**() function sets the rounding mode as specified by its argument and returns zero when it was successful.

C99 and POSIX.1-2008 specify an identifier, **FLT_ROUNDS**, defined in <*float.h*>, which indicates the implementation-defined rounding behavior for floating-point addition. This identifier has one of the following values:

−1 The rounding mode is not determinable.

0 Rounding is toward 0.

1 Rounding is toward nearest number.

2 Rounding is toward positive infinity.

3 Rounding is toward negative infinity.

Other values represent machine-dependent, nonstandard rounding modes.

The value of **FLT_ROUNDS** should reflect the current rounding mode as set by **fesetround**() (but see BUGS).

Floating-point environment

The entire floating-point environment, including control modes and status flags, can be handled as one opaque object, of type *fenv_t*. The default environment is denoted by **FE_DFL_ENV** (of type *const fenv_t **). This is the environment setup at program start and it is defined by ISO C to have round to nearest, all exceptions cleared and a nonstop (continue on exceptions) mode.

The **fegetenv**() function saves the current floating-point environment in the object **envp*.

The **feholdexcept**() function does the same, then clears all exception flags, and sets a nonstop (continue on exceptions) mode, if available. It returns zero when successful.

The **fesetenv**() function restores the floating-point environment from the object **envp*. This object must be known to be valid, for example, the result of a call to **fegetenv**() or **feholdexcept**() or equal to **FE_DFL_ENV**. This call does not raise exceptions.

The **feupdateenv**() function installs the floating-point environment represented by the object **envp*, except that currently raised exceptions are not cleared. After calling this function, the raised exceptions will be a bitwise OR of those previously set with those in **envp*. As before, the object **envp* must be known to be valid.

RETURN VALUE

These functions return zero on success and nonzero if an error occurred.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
feclearexcept(), **fegetexceptflag**(), **feraiseexcept**(), **fesetexceptflag**(), **fetestexcept**(), **fegetround**(), **fesetround**(), **fegetenv**(), **feholdexcept**(), **fesetenv**(), **feupdateenv**(), **feenableexcept**(), **fedisableexcept**(), **fegetexcept**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008, IEC 60559 (IEC 559:1989), ANSI/IEEE 854.

HISTORY

C99, POSIX.1-2001. glibc 2.1.

NOTES

glibc notes

If possible, the GNU C Library defines a macro **FE_NOMASK_ENV** which represents an environment where every exception raised causes a trap to occur. You can test for this macro using **#ifdef**. It is defined only if **_GNU_SOURCE** is defined. The C99 standard does not define a way to set individual bits in the floating-point mask, for example, to trap on specific flags. Since glibc 2.2, glibc supports the functions **feenableexcept**() and **fedisableexcept**() to set individual floating-point traps, and **fegetexcept**() to query the state.

```
#define _GNU_SOURCE       /* See feature_test_macros(7) */
#include <fenv.h>

int feenableexcept(int excepts);
int fedisableexcept(int excepts);
int fegetexcept(void);
```

The **feenableexcept**() and **fedisableexcept**() functions enable (disable) traps for each of the exceptions represented by *excepts* and return the previous set of enabled exceptions when successful, and −1 otherwise. The **fegetexcept**() function returns the set of all currently enabled exceptions.

BUGS

C99 specifies that the value of **FLT_ROUNDS** should reflect changes to the current rounding mode, as set by **fesetround**(). Currently, this does not occur: **FLT_ROUNDS** always has the value 1.

SEE ALSO

math_error(7)

NAME

clearerr, feof, ferror − check and reset stream status

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

DESCRIPTION

The function **clearerr**() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof**() tests the end-of-file indicator for the stream pointed to by *stream*, returning nonzero if it is set. The end-of-file indicator can be cleared only by the function **clearerr**().

The function **ferror**() tests the error indicator for the stream pointed to by *stream*, returning nonzero if it is set. The error indicator can be reset only by the **clearerr**() function.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

The **feof**() function returns nonzero if the end-of-file indicator is set for *stream*; otherwise, it returns zero.

The **ferror**() function returns nonzero if the error indicator is set for *stream*; otherwise, it returns zero.

ERRORS

These functions should not fail and do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
clearerr(), **feof**(), **ferror**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

NOTES

POSIX.1-2008 specifies that these functions shall not change the value of *errno* if *stream* is valid.

CAVEATS

Normally, programs should read the return value of an input function, such as *fgetc*(3), before using functions of the *feof*(3) family. Only when the function returned the sentinel value **EOF** it makes sense to distinguish between the end of a file or an error with *feof*(3) or *ferror*(3).

SEE ALSO

open(2), *fdopen*(3), *fileno*(3), *stdio*(3), *unlocked_stdio*(3)

NAME

fexecve − execute program specified via file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int fexecve(int *fd***, char *const** *argv***[], char *const** *envp***[]);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fexecve():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

fexecve() performs the same task as *execve*(2), with the difference that the file to be executed is specified via a file descriptor, *fd*, rather than via a pathname. The file descriptor *fd* must be opened read-only (**O_RDONLY**) or with the **O_PATH** flag and the caller must have permission to execute the file that it refers to.

RETURN VALUE

A successful call to **fexecve**() never returns. On error, the function does return, with a result value of −1, and *errno* is set to indicate the error.

ERRORS

Errors are as for *execve*(2), with the following additions:

EINVAL
fd is not a valid file descriptor, or *argv* is NULL, or *envp* is NULL.

ENOENT
The close-on-exec flag is set on *fd*, and *fd* refers to a script. See BUGS.

ENOSYS
The kernel does not provide the *execveat*(2) system call, and the */proc* filesystem could not be accessed.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fexecve()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.3.2.

On Linux with glibc versions 2.26 and earlier, **fexecve**() is implemented using the *proc*(5) filesystem, so */proc* needs to be mounted and available at the time of the call. Since glibc 2.27, if the underlying kernel supports the *execveat*(2) system call, then **fexecve**() is implemented using that system call, with the benefit that */proc* does not need to be mounted.

NOTES

The idea behind **fexecve**() is to allow the caller to verify (checksum) the contents of an executable before executing it. Simply opening the file, checksumming the contents, and then doing an *execve*(2) would not suffice, since, between the two steps, the filename, or a directory prefix of the pathname, could have been exchanged (by, for example, modifying the target of a symbolic link). **fexecve**() does not mitigate the problem that the *contents* of a file could be changed between the checksumming and the call to **fexecve**(); for that, the solution is to ensure that the permissions on the file prevent it from being modified by malicious users.

The natural idiom when using **fexecve**() is to set the close-on-exec flag on *fd*, so that the file descriptor does not leak through to the program that is executed. This approach is natural for two reasons. First,

it prevents file descriptors being consumed unnecessarily. (The executed program normally has no need of a file descriptor that refers to the program itself.) Second, if **fexecve**() is used recursively, employing the close-on-exec flag prevents the file descriptor exhaustion that would result from the fact that each step in the recursion would cause one more file descriptor to be passed to the new program. (But see BUGS.)

BUGS

If *fd* refers to a script (i.e., it is an executable text file that names a script interpreter with a first line that begins with the characters *#!*) and the close-on-exec flag has been set for *fd*, then **fexecve**() fails with the error **ENOENT**. This error occurs because, by the time the script interpreter is executed, *fd* has already been closed because of the close-on-exec flag. Thus, the close-on-exec flag can't be set on *fd* if it refers to a script, leading to the problems described in NOTES.

SEE ALSO

execve(2), *execveat*(2)

NAME

fflush − flush a stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int fflush(FILE *_Nullable stream);
```

DESCRIPTION

For output streams, **fflush**() forces a write of all user-space buffered data for the given output or update *stream* via the stream's underlying write function.

For input streams associated with seekable files (e.g., disk files, but not pipes or terminals), **fflush**() discards any buffered data that has been fetched from the underlying file, but has not been consumed by the application.

The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

For a nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

Upon successful completion 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EBADF
: *stream* is not an open stream, or is not open for writing.

The function **fflush**() may also fail and set *errno* for any of the errors specified for *write*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fflush()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001, POSIX.1-2008.

POSIX.1-2001 did not specify the behavior for flushing of input streams, but the behavior is specified in POSIX.1-2008.

NOTES

Note that **fflush**() flushes only the user-space buffers provided by the C library. To ensure that the data is physically stored on disk the kernel buffers must be flushed too, for example, with *sync*(2) or *fsync*(2).

SEE ALSO

fsync(2), *sync*(2), *write*(2), *fclose*(3), *fileno*(3), *fopen*(3), *fpurge*(3), *setbuf*(3), *unlocked_stdio*(3)

NAME

ffs, ffsl, ffsll − find first bit set in a word

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <strings.h>

int ffs(int i);
int ffsl(long i);
int ffsll(long long i);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ffs():
Since glibc 2.12:
_XOPEN_SOURCE >= 700
|| ! (_POSIX_C_SOURCE >= 200809L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
Before glibc 2.12:
none

ffsl(), **ffsll**():
Since glibc 2.27:
_DEFAULT_SOURCE
Before glibc 2.27:
_GNU_SOURCE

DESCRIPTION

The **ffs**() function returns the position of the first (least significant) bit set in the word *i*. The least significant bit is position 1 and the most significant position is, for example, 32 or 64. The functions **ffsll**() and **ffsl**() do the same but take arguments of possibly different size.

RETURN VALUE

These functions return the position of the first bit set, or 0 if no bits are set in *i*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ffs(), **ffsl**(), **ffsll**()	Thread safety	MT-Safe

STANDARDS

ffs() POSIX.1-2001, POSIX.1-2008, 4.3BSD.

ffsl()
ffsll() GNU.

SEE ALSO

memchr(3)

NAME

fgetc, fgets, getc, getchar, ungetc − input of characters and strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);

char *fgets(char s[restrict .size], int size, FILE *restrict stream);

int ungetc(int c, FILE *stream);
```

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc(***stdin***)**.

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the *stdio* library for the same input stream.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

fgetc(), **getc**(), and **getchar**() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fgetc(), **fgets**(), **getc**(), **getchar**(), **ungetc**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

NOTES

It is not advisable to mix calls to input functions from the *stdio* library with low-level calls to *read*(2) for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), *write*(2), *ferror*(3), *fgetwc*(3), *fgetws*(3), *fopen*(3), *fread*(3), *fseek*(3), *getline*(3), *gets*(3), *getwchar*(3), *puts*(3), *scanf*(3), *ungetwc*(3), *unlocked_stdio*(3), *feature_test_macros*(7)

NAME

fgetgrent − get group file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>
#include <grp.h>

struct group *fgetgrent(FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fgetgrent():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

The **fgetgrent**() function returns a pointer to a structure containing the group information from the file referred to by *stream*. The first time it is called it returns the first entry; thereafter, it returns successive entries. The file referred to by *stream* must have the same format as */etc/group* (see *group*(5)).

The *group* structure is defined in <*grp.h*> as follows:

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* NULL-terminated array of pointers
                               to names of group members */
};
```

RETURN VALUE

The **fgetgrent**() function returns a pointer to a *group* structure, or NULL if there are no more entries or an error occurs. In the event of an error, *errno* is set to indicate the error.

ERRORS

ENOMEM
Insufficient memory to allocate *group* structure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fgetgrent()	Thread safety	MT-Unsafe race:fgetgrent

STANDARDS

None.

HISTORY

SVr4.

SEE ALSO

endgrent(3), *fgetgrent_r*(3), *fopen*(3), *getgrent*(3), *getgrgid*(3), *getgrnam*(3), *putgrent*(3), *setgrent*(3), *group*(5)

NAME

fgetpwent − get password file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>

struct passwd *fgetpwent(FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fgetpwent():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

The **fgetpwent**() function returns a pointer to a structure containing the broken out fields of a line in the file *stream*. The first time it is called it returns the first entry; thereafter, it returns successive entries. The file referred to by *stream* must have the same format as */etc/passwd* (see *passwd*(5)).

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char   *pw_name;       /* username */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* real name */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

RETURN VALUE

The **fgetpwent**() function returns a pointer to a *passwd* structure, or NULL if there are no more entries or an error occurs. In the event of an error, *errno* is set to indicate the error.

ERRORS

ENOMEM
Insufficient memory to allocate *passwd* structure.

FILES

/etc/passwd
password database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fgetpwent()	Thread safety	MT-Unsafe race:fgetpwent

STANDARDS

None.

HISTORY

SVr4.

SEE ALSO

endpwent(3), *fgetpwent_r*(3), *fopen*(3), *getpw*(3), *getpwent*(3), *getpwnam*(3), *getpwuid*(3), *putpwent*(3), *setpwent*(3), *passwd*(5)

NAME

fgetwc, getwc − read a wide character from a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t fgetwc(FILE *stream);
wint_t getwc(FILE *stream);
```

DESCRIPTION

The **fgetwc**() function is the wide-character equivalent of the *fgetc*(3) function. It reads a wide character from *stream* and returns it. If the end of stream is reached, or if *ferror(stream)* becomes true, it returns **WEOF**. If a wide-character conversion error occurs, it sets *errno* to **EILSEQ** and returns **WEOF**.

The **getwc**() function or macro functions identically to **fgetwc**(). It may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use it.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

On success, **fgetwc**() returns the next wide-character from the stream. Otherwise, **WEOF** is returned, and *errno* is set to indicate the error.

ERRORS

Apart from the usual ones, there is

EILSEQ
: The data obtained from the input stream does not form a valid character.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fgetwc(), **getwc**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **fgetwc**() depends on the **LC_CTYPE** category of the current locale.

In the absence of additional information passed to the *fopen*(3) call, it is reasonable to expect that **fgetwc**() will actually read a multibyte sequence from the stream and then convert it to a wide character.

SEE ALSO

fgetws(3), *fputwc*(3), *ungetwc*(3), *unlocked_stdio*(3)

NAME

fgetws − read a wide-character string from a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *fgetws(wchar_t *ws***[restrict .***n***], int** *n***, FILE *restrict** *stream***);**

DESCRIPTION

The **fgetws**() function is the wide-character equivalent of the *fgets*(3) function. It reads a string of at most *n−1* wide characters into the wide-character array pointed to by *ws*, and adds a terminating null wide character (L'\0'). It stops reading wide characters after it has encountered and stored a newline wide character. It also stops when end of stream is reached.

The programmer must ensure that there is room for at least *n* wide characters at *ws*.

For a nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

The **fgetws**() function, if successful, returns *ws*. If end of stream was already reached or if an error occurred, it returns NULL.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fgetws()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **fgetws**() depends on the **LC_CTYPE** category of the current locale.

In the absence of additional information passed to the *fopen*(3) call, it is reasonable to expect that **fgetws**() will actually read a multibyte string from the stream and then convert it to a wide-character string.

This function is unreliable, because it does not permit to deal properly with null wide characters that may be present in the input.

SEE ALSO

fgetwc(3), *unlocked_stdio*(3)

NAME

fileno − obtain file descriptor of a stdio stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

int fileno(FILE *_stream_);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fileno():
_POSIX_C_SOURCE

DESCRIPTION

The function **fileno**() examines the argument *stream* and returns the integer file descriptor used to implement this stream. The file descriptor is still owned by *stream* and will be closed when *fclose*(3) is called. Duplicate the file descriptor with *dup*(2) before passing it to code that might close it.

For the nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

On success, **fileno**() returns the file descriptor associated with *stream*. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
stream is not associated with a file.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fileno()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

open(2), *fdopen*(3), *stdio*(3), *unlocked_stdio*(3)

NAME

finite, finitef, finitel, isinf, isinff, isinfl, isnan, isnanf, isnanl − BSD floating-point classification functions

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

int finite(double x);
int finitef(float x);
int finitel(long double x);

int isinf(double x);
int isinff(float x);
int isinfl(long double x);

int isnan(double x);
int isnanf(float x);
int isnanl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

finite(), **finitef**(), **finitel**():
/* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

isinf():
_XOPEN_SOURCE >= 600 || _ISOC99_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

isinff(), **isinfl**():
/* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

isnan():
_XOPEN_SOURCE || _ISOC99_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

isnanf(), **isnanl**():
_XOPEN_SOURCE >= 600
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **finite**(), **finitef**(), and **finitel**() functions return a nonzero value if *x* is neither infinite nor a "not-a-number" (NaN) value, and 0 otherwise.

The **isnan**(), **isnanf**(), and **isnanl**() functions return a nonzero value if *x* is a NaN value, and 0 otherwise.

The **isinf**(), **isinff**(), and **isinfl**() functions return 1 if *x* is positive infinity, −1 if *x* is negative infinity, and 0 otherwise.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
finite(), **finitef**(), **finitel**(), **isinf**(), **isinff**(), **isinfl**(), **isnan**(), **isnanf**(), **isnanl**()	Thread safety	MT-Safe

NOTES

Note that these functions are obsolete. C99 defines macros **isfinite**(), **isinf**(), and **isnan**() (for all types) replacing them. Further note that the C99 **isinf**() has weaker guarantees on the return value. See *fpclassify*(3).

SEE ALSO

fpclassify(3)

NAME

flockfile, ftrylockfile, funlockfile − lock FILE for stdio

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *filehandle);
int ftrylockfile(FILE *filehandle);
void funlockfile(FILE *filehandle);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:

```
/* Since glibc 2.24: */ _POSIX_C_SOURCE >= 199309L
    || /* glibc <= 2.23: */ _POSIX_C_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

The stdio functions are thread-safe. This is achieved by assigning to each *FILE* object a lockcount and (if the lockcount is nonzero) an owning thread. For each library call, these functions wait until the *FILE* object is no longer locked by a different thread, then lock it, do the requested I/O, and unlock the object again.

(Note: this locking has nothing to do with the file locking done by functions like *flock*(2) and *lockf*(3).)

All this is invisible to the C-programmer, but there may be two reasons to wish for more detailed control. On the one hand, maybe a series of I/O actions by one thread belongs together, and should not be interrupted by the I/O of some other thread. On the other hand, maybe the locking overhead should be avoided for greater efficiency.

To this end, a thread can explicitly lock the *FILE* object, then do its series of I/O actions, then unlock. This prevents other threads from coming in between. If the reason for doing this was to achieve greater efficiency, one does the I/O with the nonlocking versions of the stdio functions: with *getc_unlocked*(3) and *putc_unlocked*(3) instead of *getc*(3) and *putc*(3).

The **flockfile**() function waits for **filehandle* to be no longer locked by a different thread, then makes the current thread owner of **filehandle*, and increments the lockcount.

The **funlockfile**() function decrements the lock count.

The **ftrylockfile**() function is a nonblocking version of **flockfile**(). It does nothing in case some other thread owns **filehandle*, and it obtains ownership and increments the lockcount otherwise.

RETURN VALUE

The **ftrylockfile**() function returns zero for success (the lock was obtained), and nonzero for failure.

ERRORS

None.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
flockfile(), **ftrylockfile**(), **funlockfile**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

These functions are available when **_POSIX_THREAD_SAFE_FUNCTIONS** is defined.

SEE ALSO

unlocked_stdio(3)

NAME

floor, floorf, floorl − largest integral value not greater than argument

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

floorf(), **floorl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the largest integral value that is not greater than *x*.

For example, *floor(0.5)* is 0.0, and *floor(−0.5)* is −1.0.

RETURN VALUE

These functions return the floor of *x*.

If *x* is integral, +0, −0, NaN, or an infinity, *x* itself is returned.

ERRORS

No errors occur. POSIX.1-2001 documents a range error for overflows, but see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
floor(), **floorf**(), **floorl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SUSv2 and POSIX.1-2001 contain text about overflow (which might set *errno* to **ERANGE**, or raise an **FE_OVERFLOW** exception). In practice, the result cannot overflow on any current machine, so this error-handling stuff is just nonsense. (More precisely, overflow can happen only when the maximum value of the exponent is smaller than the number of mantissa bits. For the IEEE-754 standard 32-bit and 64-bit floating-point numbers the maximum value of the exponent is 127 (respectively, 1023), and the number of mantissa bits including the implicit bit is 24 (respectively, 53).)

SEE ALSO

ceil(3), *lrint*(3), *nearbyint*(3), *rint*(3), *round*(3), *trunc*(3)

NAME

fma, fmaf, fmal − floating-point multiply and add

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double fma(double *x***, double** *y***, double** *z***);**
float fmaf(float *x***, float** *y***, float** *z***);**
long double fmal(long double *x***, long double** *y***, long double** *z***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fma(), **fmaf**(), **fmal**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions compute *x* * *y* + *z*. The result is rounded as one ternary operation according to the current rounding mode (see *fenv*(3)).

RETURN VALUE

These functions return the value of *x* * *y* + *z*, rounded as one ternary operation.

If *x* or *y* is a NaN, a NaN is returned.

If *x* times *y* is an exact infinity, and *z* is an infinity with the opposite sign, a domain error occurs, and a NaN is returned.

If one of *x* or *y* is an infinity, the other is 0, and *z* is not a NaN, a domain error occurs, and a NaN is returned.

If one of *x* or *y* is an infinity, and the other is 0, and *z* is a NaN, a domain error occurs, and a NaN is returned.

If *x* times *y* is not an infinity times zero (or vice versa), and *z* is a NaN, a NaN is returned.

If the result overflows, a range error occurs, and an infinity with the correct sign is returned.

If the result underflows, a range error occurs, and a signed 0 is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* * *y* + *z*, or *x* * *y* is invalid and *z* is not a NaN
: An invalid floating-point exception (**FE_INVALID**) is raised.

Range error: result overflow
: An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error: result underflow
: An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fma(), **fmaf**(), **fmal**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

remainder(3), *remquo*(3)

NAME

fmax, fmaxf, fmaxl − determine maximum of two floating-point numbers

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fmax(), **fmaxf**(), **fmaxl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return the larger value of *x* and *y*.

RETURN VALUE

These functions return the maximum of *x* and *y*.

If one argument is a NaN, the other argument is returned.

If both arguments are NaN, a NaN is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fmax(), **fmaxf**(), **fmaxl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

fdim(3), *fmin*(3)

NAME

fmemopen − open memory as stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

FILE *fmemopen(void *buf* **[.***size***], size_t** *size***, const char *** *mode***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fmemopen():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **fmemopen**() function opens a stream that permits the access specified by *mode*. The stream allows I/O to be performed on the string or memory buffer pointed to by *buf*.

The *mode* argument specifies the semantics of I/O on the stream, and is one of the following:

r The stream is opened for reading.

w The stream is opened for writing.

a Append; open the stream for writing, with the initial buffer position set to the first null byte.

r+ Open the stream for reading and writing.

w+ Open the stream for reading and writing. The buffer contents are truncated (i.e., '\0' is placed in the first byte of the buffer).

a+ Append; open the stream for reading and writing, with the initial buffer position set to the first null byte.

The stream maintains the notion of a current position, the location where the next I/O operation will be performed. The current position is implicitly updated by I/O operations. It can be explicitly updated using *fseek*(3), and determined using *ftell*(3). In all modes other than append, the initial position is set to the start of the buffer. In append mode, if no null byte is found within the buffer, then the initial position is *size+1*.

If *buf* is specified as NULL, then **fmemopen**() allocates a buffer of *size* bytes. This is useful for an application that wants to write data to a temporary buffer and then read it back again. The initial position is set to the start of the buffer. The buffer is automatically freed when the stream is closed. Note that the caller has no way to obtain a pointer to the temporary buffer allocated by this call (but see *open_memstream*(3)).

If *buf* is not NULL, then it should point to a buffer of at least *size* bytes allocated by the caller.

When a stream that has been opened for writing is flushed (**fflush**(3)) or closed (**fclose**(3)), a null byte is written at the end of the buffer if there is space. The caller should ensure that an extra byte is available in the buffer (and that *size* counts that byte) to allow for this.

In a stream opened for reading, null bytes ('\0') in the buffer do not cause read operations to return an end-of-file indication. A read from the buffer will indicate end-of-file only when the current buffer position advances *size* bytes past the start of the buffer.

Write operations take place either at the current position (for modes other than append), or at the current size of the stream (for append modes).

Attempts to write more than *size* bytes to the buffer result in an error. By default, such errors will be visible (by the absence of data) only when the *stdio* buffer is flushed. Disabling buffering with the following call may be useful to detect errors at the time of an output operation:

```
setbuf(stream, NULL);
```

RETURN VALUE

Upon successful completion, **fmemopen**() returns a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fmemopen(),	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 1.0.x. POSIX.1-2008.

POSIX.1-2008 specifies that 'b' in *mode* shall be ignored. However, Technical Corrigendum 1 adjusts the standard to allow implementation-specific treatment for this case, thus permitting the glibc treatment of 'b'.

With glibc 2.22, binary mode (see below) was removed, many longstanding bugs in the implementation of **fmemopen**() were fixed, and a new versioned symbol was created for this interface.

Binary mode

From glibc 2.9 to glibc 2.21, the glibc implementation of **fmemopen**() supported a "binary" mode, enabled by specifying the letter 'b' as the second character in *mode*. In this mode, writes don't implicitly add a terminating null byte, and *fseek*(3) **SEEK_END** is relative to the end of the buffer (i.e., the value specified by the *size* argument), rather than the current string length.

An API bug afflicted the implementation of binary mode: to specify binary mode, the 'b' must be the *second* character in *mode*. Thus, for example, "wb+" has the desired effect, but "w+b" does not. This is inconsistent with the treatment of *mode* by *fopen*(3).

Binary mode was removed in glibc 2.22; a 'b' specified in *mode* has no effect.

NOTES

There is no file descriptor associated with the file stream returned by this function (i.e., *fileno*(3) will return an error if called on the returned stream).

BUGS

Before glibc 2.22, if *size* is specified as zero, **fmemopen**() fails with the error **EINVAL**. It would be more consistent if this case successfully created a stream that then returned end-of-file on the first attempt at reading; since glibc 2.22, the glibc implementation provides that behavior.

Before glibc 2.22, specifying append mode ("a" or "a+") for **fmemopen**() sets the initial buffer position to the first null byte, but (if the current position is reset to a location other than the end of the stream) does not force subsequent writes to append at the end of the stream. This bug is fixed in glibc 2.22.

Before glibc 2.22, if the *mode* argument to **fmemopen**() specifies append ("a" or "a+"), and the *size* argument does not cover a null byte in *buf*, then, according to POSIX.1-2008, the initial buffer position should be set to the next byte after the end of the buffer. However, in this case the glibc **fmemopen**() sets the buffer position to −1. This bug is fixed in glibc 2.22.

Before glibc 2.22, when a call to *fseek*(3) with a *whence* value of **SEEK_END** was performed on a stream created by **fmemopen**(), the *offset* was *subtracted* from the end-of-stream position, instead of being added. This bug is fixed in glibc 2.22.

The glibc 2.9 addition of "binary" mode for **fmemopen**() silently changed the ABI: previously, **fmemopen**() ignored 'b' in *mode*.

EXAMPLES

The program below uses **fmemopen**() to open an input buffer, and *open_memstream*(3) to open a dynamically sized output buffer. The program scans its input string (taken from the program's first command-line argument) reading integers, and writes the squares of these integers to the output buffer. An example of the output produced by this program is the following:

```
$ ./a.out '1 23 43'
size=11; ptr=1 529 1849
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    FILE *out, *in;
    int v, s;
    size_t size;
    char *ptr;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s '<num>...'\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    in = fmemopen(argv[1], strlen(argv[1]), "r");
    if (in == NULL)
        err(EXIT_FAILURE, "fmemopen");

    out = open_memstream(&ptr, &size);
    if (out == NULL)
        err(EXIT_FAILURE, "open_memstream");

    for (;;) {
        s = fscanf(in, "%d", &v);
        if (s <= 0)
            break;

        s = fprintf(out, "%d ", v * v);
        if (s == -1)
            err(EXIT_FAILURE, "fprintf");
    }

    fclose(in);
    fclose(out);

    printf("size=%zu; ptr=%s\n", size, ptr);

    free(ptr);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fopen(3), *fopencookie*(3), *open_memstream*(3)

NAME

fmin, fminf, fminl − determine minimum of two floating-point numbers

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fmin(), **fminf**(), **fminl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return the lesser value of *x* and *y*.

RETURN VALUE

These functions return the minimum of *x* and *y*.

If one argument is a NaN, the other argument is returned.

If both arguments are NaN, a NaN is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fmin(), **fminf**(), **fminl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

fdim(3), *fmax*(3)

NAME

fmod, fmodf, fmodl − floating-point remainder function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fmodf(), **fmodl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions compute the floating-point remainder of dividing *x* by *y*. The return value is *x* − *n* * *y*, where *n* is the quotient of *x* / *y*, rounded toward zero to an integer.

To obtain the modulus, more specifically, the Least Positive Residue, you will need to adjust the result from fmod like so:

```
z = fmod(x, y);
if (z < 0)
        z += y;
```

An alternate way to express this is with *fmod(fmod(x, y) + y, y)*, but the second **fmod**() usually costs way more than the one branch.

RETURN VALUE

On success, these functions return the value *x* − *n***y*, for some integer *n*, such that the returned value has the same sign as *x* and a magnitude less than the magnitude of *y*.

If *x* or *y* is a NaN, a NaN is returned.

If *x* is an infinity, a domain error occurs, and a NaN is returned.

If *y* is zero, a domain error occurs, and a NaN is returned.

If *x* is +0 (−0), and *y* is not zero, +0 (−0) is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

Domain error: *y* is zero
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fmod(), **fmodf**(), **fmodl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

BUGS

Before glibc 2.10, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred for an infinite *x*.

EXAMPLES

The call *fmod(372, 360)* returns 348.

The call *fmod(-372, 360)* returns -12.

The call *fmod(-372, -360)* also returns -12.

SEE ALSO

remainder(3)

NAME

fmtmsg − print formatted error messages

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label,
    int severity, const char *text,
    const char *action, const char *tag);
```

DESCRIPTION

This function displays a message described by its arguments on the device(s) specified in the *classification* argument. For messages written to *stderr*, the format depends on the **MSGVERB** environment variable.

The *label* argument identifies the source of the message. The string must consist of two colon separated parts where the first part has not more than 10 and the second part not more than 14 characters.

The *text* argument describes the condition of the error.

The *action* argument describes possible steps to recover from the error. If it is printed, it is prefixed by "TO FIX: ".

The *tag* argument is a reference to the online documentation where more information can be found. It should contain the *label* value and a unique identification number.

Dummy arguments

Each of the arguments can have a dummy value. The dummy classification value **MM_NULLMC** (0L) does not specify any output, so nothing is printed. The dummy severity value **NO_SEV** (0) says that no severity is supplied. The values **MM_NULLLBL**, **MM_NULLTXT**, **MM_NULLACT**, **MM_NULLTAG** are synonyms for *((char *) 0)*, the empty string, and **MM_NULLSEV** is a synonym for **NO_SEV**.

The classification argument

The *classification* argument is the sum of values describing 4 types of information.

The first value defines the output channel.

MM_PRINT Output to *stderr*.

MM_CONSOLE
: Output to the system console.

MM_PRINT | MM_CONSOLE
: Output to both.

The second value is the source of the error:

MM_HARD A hardware error occurred.

MM_FIRM A firmware error occurred.

MM_SOFT A software error occurred.

The third value encodes the detector of the problem:

MM_APPL It is detected by an application.

MM_UTIL It is detected by a utility.

MM_OPSYS
: It is detected by the operating system.

The fourth value shows the severity of the incident:

MM_RECOVER
: It is a recoverable error.

MM_NRECOV
: It is a nonrecoverable error.

The severity argument

The *severity* argument can take one of the following values:

MM_NOSEV
No severity is printed.

MM_HALT This value is printed as HALT.

MM_ERROR
This value is printed as ERROR.

MM_WARNING
This value is printed as WARNING.

MM_INFO This value is printed as INFO.

The numeric values are between 0 and 4. Using *addseverity*(3) or the environment variable **SEV_LEVEL** you can add more levels and strings to print.

RETURN VALUE

The function can return 4 values:

MM_OK Everything went smooth.

MM_NOTOK
Complete failure.

MM_NOMSG
Error writing to *stderr*.

MM_NOCON
Error writing to the console.

ENVIRONMENT

The environment variable **MSGVERB** ("message verbosity") can be used to suppress parts of the output to *stderr*. (It does not influence output to the console.) When this variable is defined, is non-NULL, and is a colon-separated list of valid keywords, then only the parts of the message corresponding to these keywords is printed. Valid keywords are "label", "severity", "text", "action", and "tag".

The environment variable **SEV_LEVEL** can be used to introduce new severity levels. By default, only the five severity levels described above are available. Any other numeric value would make **fmtmsg**() print nothing. If the user puts **SEV_LEVEL** with a format like

SEV_LEVEL=[description[:description[:...]]]

in the environment of the process before the first call to **fmtmsg**(), where each description is of the form

severity-keyword,level,printstring

then **fmtmsg**() will also accept the indicated values for the level (in addition to the standard levels 0–4), and use the indicated printstring when such a level occurs.

The severity-keyword part is not used by **fmtmsg**() but it has to be present. The level part is a string representation of a number. The numeric value must be a number greater than 4. This value must be used in the severity argument of **fmtmsg**() to select this class. It is not possible to overwrite any of the predefined classes. The printstring is the string printed when a message of this class is processed by **fmtmsg**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fmtmsg()	Thread safety	glibc >= 2.16: MT-Safe; glibc < 2.16: MT-Unsafe

Before glibc 2.16, the **fmtmsg**() function uses a static variable that is not protected, so it is not thread-safe.

Since glibc 2.16, the **fmtmsg**() function uses a lock to protect the static variable, so it is thread-safe.

STANDARDS

fmtmsg()
MSGVERB
POSIX.1-2008.

HISTORY

fmtmsg()
System V. POSIX.1-2001 and POSIX.1-2008. glibc 2.1.

MSGVERB
System V. POSIX.1-2001 and POSIX.1-2008.

SEV_LEVEL
System V.

System V and UnixWare man pages tell us that these functions have been replaced by "pfmt() and addsev()" or by "pfmt(), vpfmt(), lfmt(), and vlfmt()", and will be removed later.

EXAMPLES

```
#include <fmtmsg.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    long class = MM_PRINT | MM_SOFT | MM_OPSYS | MM_RECOVER;
    int err;

    err = fmtmsg(class, "util-linux:mount", MM_ERROR,
                 "unknown mount option", "See mount(8).",
                 "util-linux:mount:017");
    switch (err) {
    case MM_OK:
        break;
    case MM_NOTOK:
        printf("Nothing printed\n");
        break;
    case MM_NOMSG:
        printf("Nothing printed to stderr\n");
        break;
    case MM_NOCON:
        printf("No console output\n");
        break;
    default:
        printf("Unknown error from fmtmsg()\n");
    }
    exit(EXIT_SUCCESS);
}
```

The output should be:

```
util-linux:mount: ERROR: unknown mount option
TO FIX: See mount(8).  util-linux:mount:017
```

and after

```
MSGVERB=text:action; export MSGVERB
```

the output becomes:

```
unknown mount option
TO FIX: See mount(8).
```

SEE ALSO

addseverity(3), *perror*(3)

NAME

fnmatch − match filename or pathname

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fnmatch.h>

int fnmatch(const char ****pattern*, const char ****string***, int** *flags***);**

DESCRIPTION

The **fnmatch**() function checks whether the *string* argument matches the *pattern* argument, which is a shell wildcard pattern (see *glob*(7)).

The *flags* argument modifies the behavior; it is the bitwise OR of zero or more of the following flags:

FNM_NOESCAPE
If this flag is set, treat backslash as an ordinary character, instead of an escape character.

FNM_PATHNAME
If this flag is set, match a slash in *string* only with a slash in *pattern* and not by an asterisk (*) or a question mark (?) metacharacter, nor by a bracket expression ([]) containing a slash.

FNM_PERIOD
If this flag is set, a leading period in *string* has to be matched exactly by a period in *pattern*. A period is considered to be leading if it is the first character in *string*, or if both **FNM_PATHNAME** is set and the period immediately follows a slash.

FNM_FILE_NAME
This is a GNU synonym for **FNM_PATHNAME**.

FNM_LEADING_DIR
If this flag (a GNU extension) is set, the pattern is considered to be matched if it matches an initial segment of *string* which is followed by a slash. This flag is mainly for the internal use of glibc and is implemented only in certain cases.

FNM_CASEFOLD
If this flag (a GNU extension) is set, the pattern is matched case-insensitively.

FNM_EXTMATCH
If this flag (a GNU extension) is set, extended patterns are supported, as introduced by 'ksh' and now supported by other shells. The extended format is as follows, with *pattern−list* being a '|' separated list of patterns.

'?(*pattern−list*)'
The pattern matches if zero or one occurrences of any of the patterns in the *pattern−list* match the input *string*.

'*(*pattern−list*)'
The pattern matches if zero or more occurrences of any of the patterns in the *pattern−list* match the input *string*.

'+(*pattern−list*)'
The pattern matches if one or more occurrences of any of the patterns in the *pattern−list* match the input *string*.

'@(*pattern−list*)'
The pattern matches if exactly one occurrence of any of the patterns in the *pattern−list* match the input *string*.

'!(*pattern−list*)'
The pattern matches if the input *string* cannot be matched with any of the patterns in the *pattern−list*.

RETURN VALUE

Zero if *string* matches *pattern*, **FNM_NOMATCH** if there is no match or another nonzero value if there is an error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fnmatch()	Thread safety	MT-Safe env locale

STANDARDS

fnmatch()
: POSIX.1-2008.

FNM_FILE_NAME
FNM_LEADING_DIR
FNM_CASEFOLD
: GNU.

HISTORY

fnmatch()
: POSIX.1-2001, POSIX.2.

SEE ALSO

sh(1), *glob*(3), *scandir*(3), *wordexp*(3), *glob*(7)

NAME

fopen, fdopen, freopen − stream open functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict mode);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *restrict pathname, const char *restrict mode,
              FILE *restrict stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fdopen():
_POSIX_C_SOURCE

DESCRIPTION

The **fopen**() function opens the file whose name is the string pointed to by *pathname* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (possibly followed by additional characters, as described below):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. Output is always appended to the end of the file. POSIX is silent on what the initial read position is when using this mode. For glibc, the initial file position for reading is at the beginning of the file, but for Android/BSD/MacOS, the initial file position for reading is at the end of the file.

The *mode* string can also include the letter 'b' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ISO C and has no effect; the 'b' is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the 'b' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments.)

See NOTES below for details of glibc extensions for *mode*.

Any created file will have the mode **S_IRUSR** | **S_IWUSR** | **S_IRGRP** | **S_IWGRP** | **S_IROTH** | **S_IWOTH** (0666), as modified by the process's umask value (see *umask*(2)).

Reads and writes may be intermixed on read/write streams in any order. Note that ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file. (If this condition is not met, then a read is allowed to return the result of writes other than the most recent.) Therefore it is good practice (and indeed sometimes necessary under Linux) to put an *fseek*(3) or *fsetpos*(3) operation between write and read operations on such a stream. This operation may be an apparent no-op (as in *fseek(..., 0L, SEEK_CUR)* called for its synchronizing side effect).

Opening a file in append mode (**a** as the first character of *mode*) causes all subsequent write operations to this stream to occur at end-of-file, as if preceded by the call:

```
fseek(stream, 0, SEEK_END);
```

The file descriptor associated with the stream is opened as if by a call to *open*(2) with the following flags:

fopen() mode	**open() flags**
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

fdopen()

The **fdopen**() function associates a stream with the existing file descriptor, *fd*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fd*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen**() is closed. The result of applying **fdopen**() to a shared memory object is undefined.

freopen()

The **freopen**() function opens the file whose name is the string pointed to by *pathname* and associates the stream pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is used just as in the **fopen**() function.

If the *pathname* argument is a null pointer, **freopen**() changes the mode of the stream to that specified in *mode*; that is, **freopen**() reopens the pathname that is associated with the stream. The specification for this behavior was added in the C99 standard, which says:

> In this case, the file descriptor associated with the stream need not be closed if the call to **freopen**() succeeds. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

The primary use of the **freopen**() function is to change the file associated with a standard text stream (*stderr*, *stdin*, or *stdout*).

RETURN VALUE

Upon successful completion **fopen**(), **fdopen**(), and **freopen**() return a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**(), **fdopen**(), or **freopen**() was invalid.

The **fopen**(), **fdopen**(), and **freopen**() functions may also fail and set *errno* for any of the errors specified for the routine *malloc*(3).

The **fopen**() function may also fail and set *errno* for any of the errors specified for the routine *open*(2).

The **fdopen**() function may also fail and set *errno* for any of the errors specified for the routine *fcntl*(2).

The **freopen**() function may also fail and set *errno* for any of the errors specified for the routines *open*(2), *fclose*(3), and *fflush*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	**Attribute**	**Value**
fopen(), **fdopen**(), **freopen**()	Thread safety	MT-Safe

STANDARDS

fopen()
freopen()

C11, POSIX.1-2008.

fdopen()

POSIX.1-2008.

HISTORY

fopen()
freopen()
POSIX.1-2001, C89.

fdopen()
POSIX.1-2001.

NOTES

glibc notes

The GNU C library allows the following extensions for the string specified in *mode*:

c (since glibc 2.3.3)
Do not make the open operation, or subsequent read and write operations, thread cancelation points. This flag is ignored for **fdopen**().

e (since glibc 2.7)
Open the file with the **O_CLOEXEC** flag. See *open*(2) for more information. This flag is ignored for **fdopen**().

m (since glibc 2.3)
Attempt to access the file using *mmap*(2), rather than I/O system calls (**read**(2), *write*(2)). Currently, use of *mmap*(2) is attempted only for a file opened for reading.

x Open the file exclusively (like the **O_EXCL** flag of *open*(2)). If the file already exists, **fopen**() fails, and sets *errno* to **EEXIST**. This flag is ignored for **fdopen**().

In addition to the above characters, **fopen**() and **freopen**() support the following syntax in *mode*:

,ccs=*string*

The given *string* is taken as the name of a coded character set and the stream is marked as wide-oriented. Thereafter, internal conversion functions convert I/O to and from the character set *string*. If the **,ccs=***string* syntax is not specified, then the wide-orientation of the stream is determined by the first file operation. If that operation is a wide-character operation, the stream is marked wide-oriented, and functions to convert to the coded character set are loaded.

BUGS

When parsing for individual flag characters in *mode* (i.e., the characters preceding the "ccs" specification), the glibc implementation of **fopen**() and **freopen**() limits the number of characters examined in *mode* to 7 (or, before glibc 2.14, to 6, which was not enough to include possible specifications such as "rb+cmxe"). The current implementation of **fdopen**() parses at most 5 characters in *mode*.

SEE ALSO

open(2), *fclose*(3), *fileno*(3), *fmemopen*(3), *fopencookie*(3), *open_memstream*(3)

NAME

fopencookie − open a custom stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE       /* See feature_test_macros(7) */
#define _FILE_OFFSET_BITS 64
#include <stdio.h>

FILE *fopencookie(void *restrict cookie, const char *restrict mode,
                  cookie_io_functions_t io_funcs);
```

DESCRIPTION

The **fopencookie**() function allows the programmer to create a custom implementation for a standard I/O stream. This implementation can store the stream's data at a location of its own choosing; for example, **fopencookie**() is used to implement *fmemopen*(3), which provides a stream interface to data that is stored in a buffer in memory.

In order to create a custom stream the programmer must:

- Implement four "hook" functions that are used internally by the standard I/O library when performing I/O on the stream.
- Define a "cookie" data type, a structure that provides bookkeeping information (e.g., where to store data) used by the aforementioned hook functions. The standard I/O package knows nothing about the contents of this cookie (thus it is typed as *void* * when passed to *fopencookie*()), but automatically supplies the cookie as the first argument when calling the hook functions.
- Call **fopencookie**() to open a new stream and associate the cookie and hook functions with that stream.

The **fopencookie**() function serves a purpose similar to *fopen*(3): it opens a new stream and returns a pointer to a *FILE* object that is used to operate on that stream.

The *cookie* argument is a pointer to the caller's cookie structure that is to be associated with the new stream. This pointer is supplied as the first argument when the standard I/O library invokes any of the hook functions described below.

The *mode* argument serves the same purpose as for *fopen*(3). The following modes are supported: *r*, *w*, *a*, *r+*, *w+*, and *a+*. See *fopen*(3) for details.

The *io_funcs* argument is a structure that contains four fields pointing to the programmer-defined hook functions that are used to implement this stream. The structure is defined as follows

```
typedef struct {
    cookie_read_function_t  *read;
    cookie_write_function_t *write;
    cookie_seek_function_t  *seek;
    cookie_close_function_t *close;
} cookie_io_functions_t;
```

The four fields are as follows:

*cookie_read_function_t *read*

This function implements read operations for the stream. When called, it receives three arguments:

```
ssize_t read(void *cookie, char *buf, size_t size);
```

The *buf* and *size* arguments are, respectively, a buffer into which input data can be placed and the size of that buffer. As its function result, the *read* function should return the number of bytes copied into *buf*, 0 on end of file, or −1 on error. The *read* function should update the stream offset appropriately.

If **read* is a null pointer, then reads from the custom stream always return end of file.

*cookie_write_function_t *write*
: This function implements write operations for the stream. When called, it receives three arguments:

```
ssize_t write(void *cookie, const char *buf, size_t size);
```

The *buf* and *size* arguments are, respectively, a buffer of data to be output to the stream and the size of that buffer. As its function result, the *write* function should return the number of bytes copied from *buf*, or 0 on error. (The function must not return a negative value.) The *write* function should update the stream offset appropriately.

If **write* is a null pointer, then output to the stream is discarded.

*cookie_seek_function_t *seek*
: This function implements seek operations on the stream. When called, it receives three arguments:

```
int seek(void *cookie, off_t *offset, int whence);
```

The **offset* argument specifies the new file offset depending on which of the following three values is supplied in *whence*:

SEEK_SET
: The stream offset should be set **offset* bytes from the start of the stream.

SEEK_CUR
: **offset* should be added to the current stream offset.

SEEK_END
: The stream offset should be set to the size of the stream plus **offset*.

Before returning, the *seek* function should update **offset* to indicate the new stream offset.

As its function result, the *seek* function should return 0 on success, and −1 on error.

If **seek* is a null pointer, then it is not possible to perform seek operations on the stream.

*cookie_close_function_t *close*
: This function closes the stream. The hook function can do things such as freeing buffers allocated for the stream. When called, it receives one argument:

```
int close(void *cookie);
```

The *cookie* argument is the cookie that the programmer supplied when calling **fopencookie**().

As its function result, the *close* function should return 0 on success, and **EOF** on error.

If **close* is NULL, then no special action is performed when the stream is closed.

RETURN VALUE

On success **fopencookie**() returns a pointer to the new stream. On error, NULL is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fopencookie()	Thread safety	MT-Safe

STANDARDS

GNU.

EXAMPLES

The program below implements a custom stream whose functionality is similar (but not identical) to that available via *fmemopen*(3). It implements a stream whose data is stored in a memory buffer. The program writes its command-line arguments to the stream, and then seeks through the stream reading two out of every five characters and writing them to standard output. The following shell session demonstrates the use of the program:

```
$ ./a.out 'hello world'
/he/
/ w/
/d/
Reached end of file
```

Note that a more general version of the program below could be improved to more robustly handle various error situations (e.g., opening a stream with a cookie that already has an open stream; closing a stream that has already been closed).

Program source

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define INIT_BUF_SIZE 4

struct memfile_cookie {
    char   *buf;        /* Dynamically sized buffer for data */
    size_t  allocated;  /* Size of buf */
    size_t  endpos;     /* Number of characters in buf */
    off_t   offset;     /* Current file offset in buf */
};

ssize_t
memfile_write(void *c, const char *buf, size_t size)
{
    char *new_buff;
    struct memfile_cookie *cookie = c;

    /* Buffer too small? Keep doubling size until big enough. */

    while (size + cookie->offset > cookie->allocated) {
        new_buff = realloc(cookie->buf, cookie->allocated * 2);
        if (new_buff == NULL)
            return -1;
        cookie->allocated *= 2;
        cookie->buf = new_buff;
    }

    memcpy(cookie->buf + cookie->offset, buf, size);

    cookie->offset += size;
    if (cookie->offset > cookie->endpos)
        cookie->endpos = cookie->offset;

    return size;
}

ssize_t
memfile_read(void *c, char *buf, size_t size)
{
    ssize_t xbytes;
    struct memfile_cookie *cookie = c;

    /* Fetch minimum of bytes requested and bytes available. */

    xbytes = size;
    if (cookie->offset + size > cookie->endpos)
        xbytes = cookie->endpos - cookie->offset;
    if (xbytes < 0)     /* offset may be past endpos */
```

```
            xbytes = 0;

        memcpy(buf, cookie->buf + cookie->offset, xbytes);

        cookie->offset += xbytes;
        return xbytes;
    }

    int
    memfile_seek(void *c, off_t *offset, int whence)
    {
        off_t new_offset;
        struct memfile_cookie *cookie = c;

        if (whence == SEEK_SET)
            new_offset = *offset;
        else if (whence == SEEK_END)
            new_offset = cookie->endpos + *offset;
        else if (whence == SEEK_CUR)
            new_offset = cookie->offset + *offset;
        else
            return -1;

        if (new_offset < 0)
            return -1;

        cookie->offset = new_offset;
        *offset = new_offset;
        return 0;
    }

    int
    memfile_close(void *c)
    {
        struct memfile_cookie *cookie = c;

        free(cookie->buf);
        cookie->allocated = 0;
        cookie->buf = NULL;

        return 0;
    }

    int
    main(int argc, char *argv[])
    {
        cookie_io_functions_t  memfile_func = {
            .read  = memfile_read,
            .write = memfile_write,
            .seek  = memfile_seek,
            .close = memfile_close
        };
        FILE *stream;
        struct memfile_cookie mycookie;
        size_t nread;
        char buf[1000];

        /* Set up the cookie before calling fopencookie(). */
```

```
        mycookie.buf = malloc(INIT_BUF_SIZE);
        if (mycookie.buf == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        mycookie.allocated = INIT_BUF_SIZE;
        mycookie.offset = 0;
        mycookie.endpos = 0;

        stream = fopencookie(&mycookie, "w+", memfile_func);
        if (stream == NULL) {
            perror("fopencookie");
            exit(EXIT_FAILURE);
        }

        /* Write command-line arguments to our file. */

        for (size_t j = 1; j < argc; j++)
            if (fputs(argv[j], stream) == EOF) {
                perror("fputs");
                exit(EXIT_FAILURE);
            }

        /* Read two bytes out of every five, until EOF. */

        for (long p = 0; ; p += 5) {
            if (fseek(stream, p, SEEK_SET) == -1) {
                perror("fseek");
                exit(EXIT_FAILURE);
            }
            nread = fread(buf, 1, 2, stream);
            if (nread == 0) {
                if (ferror(stream) != 0) {
                    fprintf(stderr, "fread failed\n");
                    exit(EXIT_FAILURE);
                }
                printf("Reached end of file\n");
                break;
            }

            printf("/%.*s/\n", (int) nread, buf);
        }

        free(mycookie.buf);

        exit(EXIT_SUCCESS);
    }
```

NOTES

_FILE_OFFSET_BITS should be defined to be 64 in code that uses non-null *seek* or that takes the address of **fopencookie**, if the code is intended to be portable to traditional 32-bit x86 and ARM platforms where **off_t**'s width defaults to 32 bits.

SEE ALSO

fclose(3), *fmemopen*(3), *fopen*(3), *fseek*(3)

NAME

fpathconf, pathconf − get configuration values for files

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

long fpathconf(int *fd***, int** *name***);**
long pathconf(const char * *path***, int** *name***);**

DESCRIPTION

fpathconf() gets a value for the configuration option *name* for the open file descriptor *fd*.

pathconf() gets a value for configuration option *name* for the filename *path*.

The corresponding macros defined in *<unistd.h>* are minimum values; if an application wants to take advantage of values which may change, a call to **fpathconf**() or **pathconf**() can be made, which may yield more liberal results.

Setting *name* equal to one of the following constants returns the following configuration options:

_PC_LINK_MAX
The maximum number of links to the file. If *fd* or *path* refer to a directory, then the value applies to the whole directory. The corresponding macro is **_POSIX_LINK_MAX**.

_PC_MAX_CANON
The maximum length of a formatted input line, where *fd* or *path* must refer to a terminal. The corresponding macro is **_POSIX_MAX_CANON**.

_PC_MAX_INPUT
The maximum length of an input line, where *fd* or *path* must refer to a terminal. The corresponding macro is **_POSIX_MAX_INPUT**.

_PC_NAME_MAX
The maximum length of a filename in the directory *path* or *fd* that the process is allowed to create. The corresponding macro is **_POSIX_NAME_MAX**.

_PC_PATH_MAX
The maximum length of a relative pathname when *path* or *fd* is the current working directory. The corresponding macro is **_POSIX_PATH_MAX**.

_PC_PIPE_BUF
The maximum number of bytes that can be written atomically to a pipe of FIFO. For **fpathconf**(), *fd* should refer to a pipe or FIFO. For **fpathconf**(), *path* should refer to a FIFO or a directory; in the latter case, the returned value corresponds to FIFOs created in that directory. The corresponding macro is **_POSIX_PIPE_BUF**.

_PC_CHOWN_RESTRICTED
This returns a positive value if the use of *chown*(2) and *fchown*(2) for changing a file's user ID is restricted to a process with appropriate privileges, and changing a file's group ID to a value other than the process's effective group ID or one of its supplementary group IDs is restricted to a process with appropriate privileges. According to POSIX.1, this variable shall always be defined with a value other than −1. The corresponding macro is **_POSIX_CHOWN_RESTRICTED**.

If *fd* or *path* refers to a directory, then the return value applies to all files in that directory.

_PC_NO_TRUNC
This returns nonzero if accessing filenames longer than **_POSIX_NAME_MAX** generates an error. The corresponding macro is **_POSIX_NO_TRUNC**.

_PC_VDISABLE
This returns nonzero if special character processing can be disabled, where *fd* or *path* must refer to a terminal.

RETURN VALUE

The return value of these functions is one of the following:

- On error, −1 is returned and *errno* is set to indicate the error (for example, **EINVAL**, indicating that *name* is invalid).
- If *name* corresponds to a maximum or minimum limit, and that limit is indeterminate, −1 is returned and *errno* is not changed. (To distinguish an indeterminate limit from an error, set *errno* to zero before the call, and then check whether *errno* is nonzero when −1 is returned.)
- If *name* corresponds to an option, a positive value is returned if the option is supported, and −1 is returned if the option is not supported.
- Otherwise, the current value of the option or limit is returned. This value will not be more restrictive than the corresponding value that was described to the application in <*unistd.h*> or <*limits.h*> when the application was compiled.

ERRORS

EACCES
(**pathconf**()) Search permission is denied for one of the directories in the path prefix of *path*.

EBADF
(**fpathconf**()) *fd* is not a valid file descriptor.

EINVAL
name is invalid.

EINVAL
The implementation does not support an association of *name* with the specified file.

ELOOP
(**pathconf**()) Too many symbolic links were encountered while resolving *path*.

ENAMETOOLONG
(**pathconf**()) *path* is too long.

ENOENT
(**pathconf**()) A component of *path* does not exist, or *path* is an empty string.

ENOTDIR
(**pathconf**()) A component used as a directory in *path* is not in fact a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fpathconf(), **pathconf**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Files with name lengths longer than the value returned for *name* equal to **_PC_NAME_MAX** may exist in the given directory.

Some returned values may be huge; they are not suitable for allocating memory.

SEE ALSO

getconf (1), *open*(2), *statfs*(2), *confstr*(3), *sysconf*(3)

NAME

fpclassify, isfinite, isnormal, isnan, isinf − floating-point classification macros

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

int fpclassify(x);
int isfinite(x);
int isnormal(x);
int isnan(x);
int isinf(x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fpclassify(), **isfinite**(), **isnormal**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

isnan():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

isinf():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Floating point numbers can have special values, such as infinite or NaN. With the macro **fpclassify(***x***)** you can find out what type *x* is. The macro takes any floating-point expression as argument. The result is one of the following values:

FP_NAN *x* is "Not a Number".

FP_INFINITE *x* is either positive infinity or negative infinity.

FP_ZERO *x* is zero.

FP_SUBNORMAL
x is too small to be represented in normalized format.

FP_NORMAL if nothing of the above is correct then it must be a normal floating-point number.

The other macros provide a short answer to some standard questions.

isfinite(*x***)** returns a nonzero value if
(fpclassify(x) != FP_NAN && fpclassify(x) != FP_INFINITE)

isnormal(*x***)** returns a nonzero value if (fpclassify(x) == FP_NORMAL)

isnan(*x***)** returns a nonzero value if (fpclassify(x) == FP_NAN)

isinf(*x***)** returns 1 if *x* is positive infinity, and −1 if *x* is negative infinity.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fpclassify(), **isfinite**(), **isnormal**(), **isnan**(), **isinf**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

In glibc 2.01 and earlier, **isinf**() returns a nonzero value (actually: 1) if *x* is positive infinity or negative infinity. (This is all that C99 requires.)

NOTES

For **isinf**(), the standards merely say that the return value is nonzero if and only if the argument has an infinite value.

SEE ALSO

finite(3), *INFINITY*(3), *isgreater*(3), *signbit*(3)

NAME

fpurge, __fpurge − purge a stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
/* unsupported */
#include <stdio.h>

int fpurge(FILE *stream);

/* supported */
#include <stdio.h>
#include <stdio_ext.h>

void  __fpurge(FILE *stream);
```

DESCRIPTION

The function **fpurge**() clears the buffers of the given stream. For output streams this discards any unwritten output. For input streams this discards any input read from the underlying object but not yet obtained via *getc*(3); this includes any text pushed back via *ungetc*(3). See also *fflush*(3).

The function **__fpurge**() does precisely the same, but without returning a value.

RETURN VALUE

Upon successful completion **fpurge**() returns 0. On error, it returns −1 and sets *errno* to indicate the error.

ERRORS

EBADF
: *stream* is not an open stream.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
__fpurge()	Thread safety	MT-Safe race:stream

STANDARDS

None.

HISTORY

fpurge()
: 4.4BSD. Not available under Linux.

__fpurge()
: Solaris, glibc 2.1.95.

NOTES

Usually it is a mistake to want to discard input buffers.

SEE ALSO

fflush(3), *setbuf*(3), *stdio_ext*(3)

NAME

fputwc, putwc − write a wide character to a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t wc, FILE *stream);
wint_t putwc(wchar_t wc, FILE *stream);
```

DESCRIPTION

The **fputwc**() function is the wide-character equivalent of the *fputc*(3) function. It writes the wide character *wc* to *stream*. If *ferror(stream)* becomes true, it returns **WEOF**. If a wide-character conversion error occurs, it sets *errno* to **EILSEQ** and returns **WEOF**. Otherwise, it returns *wc*.

The **putwc**() function or macro functions identically to **fputwc**(). It may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use it.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

On success, **fputwc**() function returns *wc*. Otherwise, **WEOF** is returned, and *errno* is set to indicate the error.

ERRORS

Apart from the usual ones, there is

EILSEQ
: Conversion of *wc* to the stream's encoding fails.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fputwc(), **putwc**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The behavior of **fputwc**() depends on the **LC_CTYPE** category of the current locale.

In the absence of additional information passed to the *fopen*(3) call, it is reasonable to expect that **fputwc**() will actually write the multibyte sequence corresponding to the wide character *wc*.

SEE ALSO

fgetwc(3), *fputws*(3), *unlocked_stdio*(3)

NAME

fputws − write a wide-character string to a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int fputws(const wchar_t *restrict *ws***, FILE *restrict** *stream***);**

DESCRIPTION

The **fputws**() function is the wide-character equivalent of the *fputs*(3) function. It writes the wide-character string starting at *ws*, up to but not including the terminating null wide character (L'\0'), to *stream*.

For a nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

The **fputws**() function returns a nonnegative integer if the operation was successful, or −1 to indicate an error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fputws()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **fputws**() depends on the **LC_CTYPE** category of the current locale.

In the absence of additional information passed to the *fopen*(3) call, it is reasonable to expect that **fputws**() will actually write the multibyte string corresponding to the wide-character string *ws*.

SEE ALSO

fputwc(3), *unlocked_stdio*(3)

NAME

fread, fwrite − binary stream input/output

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

size_t fread(void ptr[restrict .size * .nmemb],
             size_t size, size_t nmemb,
             FILE *restrict stream);
size_t fwrite(const void ptr[restrict .size * .nmemb],
              size_t size, size_t nmemb,
              FILE *restrict stream);
```

DESCRIPTION

The function **fread**() reads *nmemb* items of data, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* items of data, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

On success, **fread**() and **fwrite**() return the number of items read or written. This number equals the number of bytes transferred only when *size* is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).

The file position indicator for the stream is advanced by the number of bytes successfully read or written.

fread() does not distinguish between end-of-file and error, and callers must use *feof*(3) and *ferror*(3) to determine which occurred.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fread(), **fwrite**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

EXAMPLES

The program below demonstrates the use of **fread**() by parsing /bin/sh ELF executable in binary mode and printing its magic and class:

```
$ ./a.out
ELF magic: 0x7f454c46
Class: 0x02
```

Program source

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

int
main(void)
{
    FILE           *fp;
    size_t         ret;
```

```
        unsigned char  buffer[4];

        fp = fopen("/bin/sh", "rb");
        if (!fp) {
            perror("fopen");
            return EXIT_FAILURE;
        }

        ret = fread(buffer, sizeof(*buffer), ARRAY_SIZE(buffer), fp);
        if (ret != ARRAY_SIZE(buffer)) {
            fprintf(stderr, "fread() failed: %zu\n", ret);
            exit(EXIT_FAILURE);
        }

        printf("ELF magic: %#04x%02x%02x%02x\n", buffer[0], buffer[1],
               buffer[2], buffer[3]);

        ret = fread(buffer, 1, 1, fp);
        if (ret != 1) {
            fprintf(stderr, "fread() failed: %zu\n", ret);
            exit(EXIT_FAILURE);
        }

        printf("Class: %#04x\n", buffer[0]);

        fclose(fp);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

read(2), *write*(2), *feof*(3), *ferror*(3), *unlocked_stdio*(3)

NAME

frexp, frexpf, frexpl − convert floating-point number to fractional and integral components

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double frexp(double x, int *exp);
float frexpf(float x, int *exp);
long double frexpl(long double x, int *exp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

frexpf(), **frexpl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions are used to split the number *x* into a normalized fraction and an exponent which is stored in *exp*.

RETURN VALUE

These functions return the normalized fraction. If the argument *x* is not zero, the normalized fraction is *x* times a power of two, and its absolute value is always in the range 1/2 (inclusive) to 1 (exclusive), that is, [0.5,1).

If *x* is zero, then the normalized fraction is zero and zero is stored in *exp*.

If *x* is a NaN, a NaN is returned, and the value of **exp* is unspecified.

If *x* is positive infinity (negative infinity), positive infinity (negative infinity) is returned, and the value of **exp* is unspecified.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
frexp(), **frexpf**(), **frexpl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

EXAMPLES

The program below produces results such as the following:

```
$ ./a.out 2560
frexp(2560, &e) = 0.625: 0.625 * 2^12 = 2560
$ ./a.out -4
frexp(-4, &e) = -0.5: -0.5 * 2^3 = -4
```

Program source

```
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
```

```
    {
        double x, r;
        int exp;

        x = strtod(argv[1], NULL);
        r = frexp(x, &exp);

        printf("frexp(%g, &e) = %g: %g * %d^%d = %g\n", x, r, r, 2, exp, x);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ldexp(3), *modf*(3)

NAME

fgetpos, fseek, fsetpos, ftell, rewind − reposition a stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);

void rewind(FILE *stream);

int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the *ungetc*(3) function on the same stream.

The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

The **rewind**() function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see *clearerr*(3)).

The **fgetpos**() and **fsetpos**() functions are alternate interfaces equivalent to **ftell**() and **fseek**() (with *whence* set to **SEEK_SET**), setting and storing the current value of the file offset into or from the object referenced by *pos*. On some non-UNIX systems, an *fpos_t* object may be a complex object and these routines may be the only way to portably reposition a text stream.

If the stream refers to a regular file and the resulting stream offset is beyond the size of the file, subsequent writes will extend the file with a hole, up to the offset, before committing any data. See *lseek*(2) for details on file seeking semantics.

RETURN VALUE

The **rewind**() function returns no value. Upon successful completion, **fgetpos**(), **fseek**(), **fsetpos**() return 0, and **ftell**() returns the current offset. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
: The *whence* argument to **fseek**() was not **SEEK_SET**, **SEEK_END**, or **SEEK_CUR**. Or: the resulting file offset would be negative.

ESPIPE
: The file descriptor underlying *stream* is not seekable (e.g., it refers to a pipe, FIFO, or socket).

The functions **fgetpos**(), **fseek**(), **fsetpos**(), and **ftell**() may also fail and set *errno* for any of the errors specified for the routines *fflush*(3), *fstat*(2), *lseek*(2), and *malloc*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fseek(), **ftell**(), **rewind**(), **fgetpos**(), **fsetpos**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

SEE ALSO

lseek(2), *fseeko*(3)

NAME

fseeko, ftello − seek to or report file position

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int fseeko(FILE *stream, off_t offset, int whence);
off_t ftello(FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fseeko(), **ftello**():
_FILE_OFFSET_BITS == 64 || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **fseeko**() and **ftello**() functions are identical to *fseek*(3) and *ftell*(3) (see *fseek*(3)), respectively, except that the *offset* argument of **fseeko**() and the return value of **ftello**() is of type *off_t* instead of *long*.

On some architectures, both *off_t* and *long* are 32-bit types, but defining **_FILE_OFFSET_BITS** with the value 64 (before including *any* header files) will turn *off_t* into a 64-bit type.

RETURN VALUE

On successful completion, **fseeko**() returns 0, while **ftello**() returns the current offset. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

See the ERRORS in *fseek*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fseeko(), **ftello**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001, SUSv2.

NOTES

The declarations of these functions can also be obtained by defining the obsolete **_LARGEFILE_SOURCE** feature test macro.

SEE ALSO

fseek(3)

NAME

ftime − return date and time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/timeb.h>

int ftime(struct timeb **tp*);

DESCRIPTION

NOTE: This function is no longer provided by the GNU C library. Use *clock_gettime*(2) instead.

This function returns the current time as seconds and milliseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). The time is returned in *tp*, which is declared as follows:

```
struct timeb {
    time_t          time;
    unsigned short millitm;
    short           timezone;
    short           dstflag;
};
```

Here *time* is the number of seconds since the Epoch, and *millitm* is the number of milliseconds since *time* seconds since the Epoch. The *timezone* field is the local timezone measured in minutes of time west of Greenwich (with a negative value indicating minutes east of Greenwich). The *dstflag* field is a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

POSIX.1-2001 says that the contents of the *timezone* and *dstflag* fields are unspecified; avoid relying on them.

RETURN VALUE

This function always returns 0. (POSIX.1-2001 specifies, and some systems document, a −1 error return.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ftime()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

Removed in glibc 2.33. 4.2BSD, POSIX.1-2001. Removed in POSIX.1-2008.

This function is obsolete. Don’t use it. If the time in seconds suffices, *time*(2) can be used; *gettimeofday*(2) gives microseconds; *clock_gettime*(2) gives nanoseconds but is not as widely available.

BUGS

Early glibc2 is buggy and returns 0 in the *millitm* field; glibc 2.1.1 is correct again.

SEE ALSO

gettimeofday(2), *time*(2)

NAME

ftok − convert a pathname and a project identifier to a System V IPC key

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/ipc.h>

key_t ftok(const char ****pathname*, int** *proj_id***);**

DESCRIPTION

The **ftok**() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file) and the least significant 8 bits of *proj_id* (which must be nonzero) to generate a *key_t* type System V IPC key, suitable for use with *msgget*(2), *semget*(2), or *shmget*(2).

The resulting value is the same for all pathnames that name the same file, when the same value of *proj_id* is used. The value returned should be different when the (simultaneously existing) files or the project IDs differ.

RETURN VALUE

On success, the generated *key_t* value is returned. On failure −1 is returned, with *errno* indicating the error as for the *stat*(2) system call.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ftok()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

On some ancient systems, the prototype was:

```
key_t ftok(char *pathname, char proj_id);
```

Today, *proj_id* is an *int*, but still only 8 bits are used. Typical usage has an ASCII character *proj_id*, that is why the behavior is said to be undefined when *proj_id* is zero.

Of course, no guarantee can be given that the resulting *key_t* is unique. Typically, a best-effort attempt combines the given *proj_id* byte, the lower 16 bits of the inode number, and the lower 8 bits of the device number into a 32-bit result. Collisions may easily happen, for example between files on */dev/hda1* and files on */dev/sda1*.

EXAMPLES

See *semget*(2).

SEE ALSO

msgget(2), *semget*(2), *shmget*(2), *stat*(2), *sysvipc*(7)

NAME

fts, fts_open, fts_read, fts_children, fts_set, fts_close − traverse a file hierarchy

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <fts.h>

FTS *fts_open(char *const * *path_argv***, int** *options***,**
int (*_Nullable *compar***)(const FTSENT **, const FTSENT **));**

FTSENT *fts_read(FTS * *ftsp***);**

FTSENT *fts_children(FTS * *ftsp***, int** *instr***);**

int fts_set(FTS * *ftsp***, FTSENT *** *f* **, int** *instr***);**

int fts_close(FTS * *ftsp***);**

DESCRIPTION

The fts functions are provided for traversing file hierarchies. A simple overview is that the **fts_open**() function returns a "handle" (of type *FTS **) that refers to a file hierarchy "stream". This handle is then supplied to the other fts functions. The function **fts_read**() returns a pointer to a structure describing one of the files in the file hierarchy. The function **fts_children**() returns a pointer to a linked list of structures, each of which describes one of the files contained in a directory in the hierarchy.

In general, directories are visited two distinguishable times; in preorder (before any of their descendants are visited) and in postorder (after all of their descendants have been visited). Files are visited once. It is possible to walk the hierarchy "logically" (visiting the files that symbolic links point to) or physically (visiting the symbolic links themselves), order the walk of the hierarchy or prune and/or revisit portions of the hierarchy.

Two structures (and associated types) are defined in the include file <*fts.h*>. The first type is *FTS*, the structure that represents the file hierarchy itself. The second type is *FTSENT*, the structure that represents a file in the file hierarchy. Normally, an *FTSENT* structure is returned for every file in the file hierarchy. In this manual page, "file" and "FTSENT structure" are generally interchangeable.

The *FTSENT* structure contains fields describing a file. The structure contains at least the following fields (there are additional fields that should be considered private to the implementation):

```
typedef struct _ftsent {
    unsigned short  fts_info;     /* flags for FTSENT structure */
    char           *fts_accpath;  /* access path */
    char           *fts_path;     /* root path */
    short           fts_pathlen;  /* strlen(fts_path) +
                                     strlen(fts_name) */
    char           *fts_name;     /* filename */
    short           fts_namelen;  /* strlen(fts_name) */
    short           fts_level;    /* depth (-1 to N) */
    int             fts_errno;    /* file errno */
    long            fts_number;   /* local numeric value */
    void           *fts_pointer;  /* local address value */
    struct _ftsent *fts_parent;   /* parent directory */
    struct _ftsent *fts_link;     /* next file structure */
    struct _ftsent *fts_cycle;    /* cycle structure */
    struct stat    *fts_statp;    /* [l]stat(2) information */
} FTSENT;
```

These fields are defined as follows:

fts_info

One of the following values describing the returned *FTSENT* structure and the file it represents. With the exception of directories without errors (**FTS_D**), all of these entries are terminal, that is, they will not be revisited, nor will any of their descendants be visited.

FTS_D A directory being visited in preorder.

FTS_DC
A directory that causes a cycle in the tree. (The *fts_cycle* field of the *FTSENT* structure will be filled in as well.)

FTS_DEFAULT
Any *FTSENT* structure that represents a file type not explicitly described by one of the other *fts_info* values.

FTS_DNR
A directory which cannot be read. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_DOT
A file named "." or ".." which was not specified as a filename to **fts_open**() (see **FTS_SEEDOT**).

FTS_DP
A directory being visited in postorder. The contents of the *FTSENT* structure will be unchanged from when it was returned in preorder, that is, with the *fts_info* field set to **FTS_D**.

FTS_ERR
This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_F A regular file.

FTS_NS
A file for which no [**l**]*stat*(2) information was available. The contents of the *fts_statp* field are undefined. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_NSOK
A file for which no [**l**]*stat*(2) information was requested. The contents of the *fts_statp* field are undefined.

FTS_SL
A symbolic link.

FTS_SLNONE
A symbolic link with a nonexistent target. The contents of the *fts_statp* field reference the file characteristic information for the symbolic link itself.

fts_accpath
A path for accessing the file from the current directory.

fts_path
The path for the file relative to the root of the traversal. This path contains the path specified to **fts_open**() as a prefix.

fts_pathlen
The sum of the lengths of the strings referenced by *fts_path* and *fts_name*.

fts_name
The name of the file.

fts_namelen
The length of the string referenced by *fts_name*.

fts_level
The depth of the traversal, numbered from −1 to N, where this file was found. The *FTSENT* structure representing the parent of the starting point (or root) of the traversal is numbered −1, and the *FTSENT* structure for the root itself is numbered 0.

fts_errno
If **fts_children**() or **fts_read**() returns an *FTSENT* structure whose *fts_info* field is set to **FTS_DNR**, **FTS_ERR**, or **FTS_NS**, the *fts_errno* field contains the error number (i.e., the

errno value) specifying the cause of the error. Otherwise, the contents of the *fts_errno* field are undefined.

fts_number
This field is provided for the use of the application program and is not modified by the fts functions. It is initialized to 0.

fts_pointer
This field is provided for the use of the application program and is not modified by the fts functions. It is initialized to NULL.

fts_parent
A pointer to the *FTSENT* structure referencing the file in the hierarchy immediately above the current file, that is, the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the *fts_level*, *fts_number*, and *fts_pointer* fields are guaranteed to be initialized.

fts_link
Upon return from the **fts_children**() function, the *fts_link* field points to the next structure in the NULL-terminated linked list of directory members. Otherwise, the contents of the *fts_link* field are undefined.

fts_cycle
If a directory causes a cycle in the hierarchy (see **FTS_DC**), either because of a hard link between two directories, or a symbolic link pointing to a directory, the *fts_cycle* field of the structure will point to the *FTSENT* structure in the hierarchy that references the same file as the current *FTSENT* structure. Otherwise, the contents of the *fts_cycle* field are undefined.

fts_statp
A pointer to [**l**]*stat*(2) information for the file.

A single buffer is used for all of the paths of all of the files in the file hierarchy. Therefore, the *fts_path* and *fts_accpath* fields are guaranteed to be null-terminated *only* for the file most recently returned by **fts_read**(). To use these fields to reference any files represented by other *FTSENT* structures will require that the path buffer be modified using the information contained in that *FTSENT* structure's *fts_pathlen* field. Any such modifications should be undone before further calls to **fts_read**() are attempted. The *fts_name* field is always null-terminated.

fts_open()
The **fts_open**() function takes a pointer to an array of character pointers naming one or more paths which make up a logical file hierarchy to be traversed. The array must be terminated by a null pointer.

There are a number of options, at least one of which (either **FTS_LOGICAL** or **FTS_PHYSICAL**) must be specified. The options are selected by ORing the following values:

FTS_LOGICAL
This option causes the fts routines to return *FTSENT* structures for the targets of symbolic links instead of the symbolic links themselves. If this option is set, the only symbolic links for which *FTSENT* structures are returned to the application are those referencing nonexistent files: the *fts_statp* field is obtained via *stat*(2) with a fallback to *lstat*(2).

FTS_PHYSICAL
This option causes the fts routines to return *FTSENT* structures for symbolic links themselves instead of the target files they point to. If this option is set, *FTSENT* structures for all symbolic links in the hierarchy are returned to the application: the *fts_statp* field is obtained via *lstat*(2).

FTS_COMFOLLOW
This option causes any symbolic link specified as a root path to be followed immediately, as if via **FTS_LOGICAL**, regardless of the primary mode.

FTS_NOCHDIR
As a performance optimization, the fts functions change directories as they walk the file hierarchy. This has the side-effect that an application cannot rely on being in any particular directory during the traversal. This option turns off this optimization, and the fts functions will not change the current directory. Note that applications should not themselves change their

current directory and try to access files unless **FTS_NOCHDIR** is specified and absolute pathnames were provided as arguments to **fts_open**().

FTS_NOSTAT
By default, returned *FTSENT* structures reference file characteristic information (the *fts_statp* field) for each file visited. This option relaxes that requirement as a performance optimization, allowing the fts functions to set the *fts_info* field to **FTS_NSOK** and leave the contents of the *fts_statp* field undefined.

FTS_SEEDOT
By default, unless they are specified as path arguments to **fts_open**(), any files named "." or ".." encountered in the file hierarchy are ignored. This option causes the fts routines to return *FTSENT* structures for them.

FTS_XDEV
This option prevents fts from descending into directories that have a different device number than the file from which the descent began.

The argument **compar**() specifies a user-defined function which may be used to order the traversal of the hierarchy. It takes two pointers to pointers to *FTSENT* structures as arguments and should return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before, in any order with respect to, or after, the file referenced by its second argument. The *fts_accpath*, *fts_path*, and *fts_pathlen* fields of the *FTSENT* structures may *never* be used in this comparison. If the *fts_info* field is set to **FTS_NS** or **FTS_NSOK**, the *fts_statp* field may not either. If the **compar**() argument is NULL, the directory traversal order is in the order listed in *path_argv* for the root paths, and in the order listed in the directory for everything else.

fts_read()
The **fts_read**() function returns a pointer to an *FTSENT* structure describing a file in the hierarchy. Directories (that are readable and do not cause cycles) are visited at least twice, once in preorder and once in postorder. All other files are visited at least once. (Hard links between directories that do not cause cycles or symbolic links to symbolic links may cause files to be visited more than once, or directories more than twice.)

If all the members of the hierarchy have been returned, **fts_read**() returns NULL and sets *errno* to 0. If an error unrelated to a file in the hierarchy occurs, **fts_read**() returns NULL and sets *errno* to indicate the error. If an error related to a returned file occurs, a pointer to an *FTSENT* structure is returned, and *errno* may or may not have been set (see *fts_info*).

The *FTSENT* structures returned by **fts_read**() may be overwritten after a call to **fts_close**() on the same file hierarchy stream, or, after a call to **fts_read**() on the same file hierarchy stream unless they represent a file of type directory, in which case they will not be overwritten until after a call to **fts_read**() after the *FTSENT* structure has been returned by the function **fts_read**() in postorder.

fts_children()
The **fts_children**() function returns a pointer to an *FTSENT* structure describing the first entry in a NULL-terminated linked list of the files in the directory represented by the *FTSENT* structure most recently returned by **fts_read**(). The list is linked through the *fts_link* field of the *FTSENT* structure, and is ordered by the user-specified comparison function, if any. Repeated calls to **fts_children**() will re-create this linked list.

As a special case, if **fts_read**() has not yet been called for a hierarchy, **fts_children**() will return a pointer to the files in the logical directory specified to **fts_open**(), that is, the arguments specified to **fts_open**(). Otherwise, if the *FTSENT* structure most recently returned by **fts_read**() is not a directory being visited in preorder, or the directory does not contain any files, **fts_children**() returns NULL and sets *errno* to zero. If an error occurs, **fts_children**() returns NULL and sets *errno* to indicate the error.

The *FTSENT* structures returned by **fts_children**() may be overwritten after a call to **fts_children**(), **fts_close**(), or **fts_read**() on the same file hierarchy stream.

The *instr* argument is either zero or the following value:

FTS_NAMEONLY
Only the names of the files are needed. The contents of all the fields in the returned linked list of structures are undefined with the exception of the *fts_name* and *fts_namelen* fields.

fts_set()
The function **fts_set**() allows the user application to determine further processing for the file *f* of the stream *ftsp*. The **fts_set**() function returns 0 on success, and −1 if an error occurs.

The *instr* argument is either 0 (meaning "do nothing") or one of the following values:

FTS_AGAIN
Revisit the file; any file type may be revisited. The next call to **fts_read**() will return the referenced file. The *fts_stat* and *fts_info* fields of the structure will be reinitialized at that time, but no other fields will have been changed. This option is meaningful only for the most recently returned file from **fts_read**(). Normal use is for postorder directory visits, where it causes the directory to be revisited (in both preorder and postorder) as well as all of its descendants.

FTS_FOLLOW
The referenced file must be a symbolic link. If the referenced file is the one most recently returned by **fts_read**(), the next call to **fts_read**() returns the file with the *fts_info* and *fts_statp* fields reinitialized to reflect the target of the symbolic link instead of the symbolic link itself. If the file is one of those most recently returned by **fts_children**(), the *fts_info* and *fts_statp* fields of the structure, when returned by **fts_read**(), will reflect the target of the symbolic link instead of the symbolic link itself. In either case, if the target of the symbolic link does not exist, the fields of the returned structure will be unchanged and the *fts_info* field will be set to **FTS_SLNONE**.

If the target of the link is a directory, the preorder return, followed by the return of all of its descendants, followed by a postorder return, is done.

FTS_SKIP
No descendants of this file are visited. The file may be one of those most recently returned by either **fts_children**() or **fts_read**().

fts_close()
The **fts_close**() function closes the file hierarchy stream referred to by *ftsp* and restores the current directory to the directory from which **fts_open**() was called to open *ftsp*. The **fts_close**() function returns 0 on success, and −1 if an error occurs.

ERRORS
The function **fts_open**() may fail and set *errno* for any of the errors specified for *open*(2) and *malloc*(3).

In addition, **fts_open**() may fail and set *errno* as follows:

ENOENT
Any element of *path_argv* was an empty string.

The function **fts_close**() may fail and set *errno* for any of the errors specified for *chdir*(2) and *close*(2).

The functions **fts_read**() and **fts_children**() may fail and set *errno* for any of the errors specified for *chdir*(2), *malloc*(3), *opendir*(3), *readdir*(3), and [**l**]*stat*(2).

In addition, **fts_children**(), **fts_open**(), and **fts_set**() may fail and set *errno* as follows:

EINVAL
options or *instr* was invalid.

ATTRIBUTES
For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fts_open(), **fts_set**(), **fts_close**()	Thread safety	MT-Safe
fts_read(), **fts_children**()	Thread safety	MT-Unsafe

STANDARDS
None.

HISTORY
glibc 2. 4.4BSD.

BUGS

Before glibc 2.23, all of the APIs described in this man page are not safe when compiling a program using the LFS APIs (e.g., when compiling with *−D_FILE_OFFSET_BITS=64*).

SEE ALSO

find(1), *chdir*(2), *lstat*(2), *stat*(2), *ftw*(3), *qsort*(3)

NAME

ftw, nftw − file tree walk

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <ftw.h>

int nftw(const char *dirpath,
    int (* fn)(const char * fpath, const struct stat *sb,
        int typeflag, struct FTW * ftwbuf ),
    int nopenfd, int  flags);

[[deprecated]]
int ftw(const char *dirpath,
    int (* fn)(const char * fpath, const struct stat *sb,
        int typeflag),
    int nopenfd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nftw():
_XOPEN_SOURCE >= 500

DESCRIPTION

nftw() walks through the directory tree that is located under the directory *dirpath*, and calls *fn*() once for each entry in the tree. By default, directories are handled before the files and subdirectories they contain (preorder traversal).

To avoid using up all of the calling process's file descriptors, *nopenfd* specifies the maximum number of directories that **nftw**() will hold open simultaneously. When the search depth exceeds this, **nftw**() will become slower because directories have to be closed and reopened. **nftw**() uses at most one file descriptor for each level in the directory tree.

For each entry found in the tree, **nftw**() calls *fn*() with four arguments: *fpath*, *sb*, *typeflag*, and *ftwbuf*. *fpath* is the pathname of the entry, and is expressed either as a pathname relative to the calling process's current working directory at the time of the call to **nftw**(), if *dirpath* was expressed as a relative pathname, or as an absolute pathname, if *dirpath* was expressed as an absolute pathname. *sb* is a pointer to the *stat* structure returned by a call to *stat*(2) for *fpath*.

The *typeflag* argument passed to *fn*() is an integer that has one of the following values:

FTW_F
: *fpath* is a regular file.

FTW_D
: *fpath* is a directory.

FTW_DNR
: *fpath* is a directory which can't be read.

FTW_DP
: *fpath* is a directory, and **FTW_DEPTH** was specified in *flags*. (If **FTW_DEPTH** was not specified in *flags*, then directories will always be visited with *typeflag* set to **FTW_D**.) All of the files and subdirectories within *fpath* have been processed.

FTW_NS
: The *stat*(2) call failed on *fpath*, which is not a symbolic link. The probable cause for this is that the caller had read permission on the parent directory, so that the filename *fpath* could be seen, but did not have execute permission, so that the file could not be reached for *stat*(2). The contents of the buffer pointed to by *sb* are undefined.

FTW_SL
: *fpath* is a symbolic link, and **FTW_PHYS** was set in *flags*.

FTW_SLN
: *fpath* is a symbolic link pointing to a nonexistent file. (This occurs only if **FTW_PHYS** is not set.) In this case the *sb* argument passed to *fn*() contains information returned by

performing *lstat*(2) on the "dangling" symbolic link. (But see BUGS.)

The fourth argument (*ftwbuf*) that **nftw**() supplies when calling *fn*() is a pointer to a structure of type *FTW*:

```
struct FTW {
    int base;
    int level;
};
```

base is the offset of the filename (i.e., basename component) in the pathname given in *fpath*. *level* is the depth of *fpath* in the directory tree, relative to the root of the tree (*dirpath*, which has depth 0).

To stop the tree walk, *fn*() returns a nonzero value; this value will become the return value of **nftw**(). As long as *fn*() returns 0, **nftw**() will continue either until it has traversed the entire tree, in which case it will return zero, or until it encounters an error (such as a *malloc*(3) failure), in which case it will return −1.

Because **nftw**() uses dynamic data structures, the only safe way to exit out of a tree walk is to return a nonzero value from *fn*(). To allow a signal to terminate the walk without causing a memory leak, have the handler set a global flag that is checked by *fn*(). *Don't* use *longjmp*(3) unless the program is going to terminate.

The *flags* argument of **nftw**() is formed by ORing zero or more of the following flags:

FTW_ACTIONRETVAL (since glibc 2.3.3)
: If this glibc-specific flag is set, then **nftw**() handles the return value from *fn*() differently. *fn*() should return one of the following values:

  FTW_CONTINUE
  : Instructs **nftw**() to continue normally.

  FTW_SKIP_SIBLINGS
  : If *fn*() returns this value, then siblings of the current entry will be skipped, and processing continues in the parent.

  FTW_SKIP_SUBTREE
  : If *fn*() is called with an entry that is a directory (*typeflag* is **FTW_D**), this return value will prevent objects within that directory from being passed as arguments to *fn*(). **nftw**() continues processing with the next sibling of the directory.

  FTW_STOP
  : Causes **nftw**() to return immediately with the return value **FTW_STOP**.

  Other return values could be associated with new actions in the future; *fn*() should not return values other than those listed above.

  The feature test macro **_GNU_SOURCE** must be defined (before including *any* header files) in order to obtain the definition of **FTW_ACTIONRETVAL** from <*ftw.h*>.

FTW_CHDIR
: If set, do a *chdir*(2) to each directory before handling its contents. This is useful if the program needs to perform some action in the directory in which *fpath* resides. (Specifying this flag has no effect on the pathname that is passed in the *fpath* argument of *fn*.)

FTW_DEPTH
: If set, do a post-order traversal, that is, call *fn*() for the directory itself *after* handling the contents of the directory and its subdirectories. (By default, each directory is handled *before* its contents.)

FTW_MOUNT
: If set, stay within the same filesystem (i.e., do not cross mount points).

FTW_PHYS
: If set, do not follow symbolic links. (This is what you want.) If not set, symbolic links are followed, but no file is reported twice.

  If **FTW_PHYS** is not set, but **FTW_DEPTH** is set, then the function *fn*() is never called for a directory that would be a descendant of itself.

ftw()

ftw() is an older function that offers a subset of the functionality of **nftw**(). The notable differences are as follows:

- **ftw**() has no *flags* argument. It behaves the same as when **nftw**() is called with *flags* specified as zero.
- The callback function, *fn*(), is not supplied with a fourth argument.
- The range of values that is passed via the *typeflag* argument supplied to *fn*() is smaller: just **FTW_F**, **FTW_D**, **FTW_DNR**, **FTW_NS**, and (possibly) **FTW_SL**.

RETURN VALUE

These functions return 0 on success, and −1 if an error occurs.

If *fn*() returns nonzero, then the tree walk is terminated and the value returned by *fn*() is returned as the result of **ftw**() or **nftw**().

If **nftw**() is called with the **FTW_ACTIONRETVAL** flag, then the only nonzero value that should be used by *fn*() to terminate the tree walk is **FTW_STOP**, and that value is returned as the result of **nftw**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nftw()	Thread safety	MT-Safe cwd
ftw()	Thread safety	MT-Safe

VERSIONS

In some implementations (e.g., glibc), **ftw**() will never use **FTW_SL**; on other systems **FTW_SL** occurs only for symbolic links that do not point to an existing file; and again on other systems **ftw**() will use **FTW_SL** for each symbolic link. If *fpath* is a symbolic link and *stat*(2) failed, POSIX.1-2008 states that it is undefined whether **FTW_NS** or **FTW_SL** is passed in *typeflag*. For predictable results, use **nftw**().

STANDARDS

POSIX.1-2008.

HISTORY

ftw() POSIX.1-2001, SVr4, SUSv1. POSIX.1-2008 marks it as obsolete.

nftw() glibc 2.1. POSIX.1-2001, SUSv1.

FTW_SL
POSIX.1-2001, SUSv1.

NOTES

POSIX.1-2008 notes that the results are unspecified if *fn* does not preserve the current working directory.

BUGS

According to POSIX.1-2008, when the *typeflag* argument passed to *fn*() contains **FTW_SLN**, the buffer pointed to by *sb* should contain information about the dangling symbolic link (obtained by calling *lstat*(2) on the link). Early glibc versions correctly followed the POSIX specification on this point. However, as a result of a regression introduced in glibc 2.4, the contents of the buffer pointed to by *sb* were undefined when **FTW_SLN** is passed in *typeflag*. (More precisely, the contents of the buffer were left unchanged in this case.) This regression was eventually fixed in glibc 2.30, so that the glibc implementation (once more) follows the POSIX specification.

EXAMPLES

The following program traverses the directory tree under the path named in its first command-line argument, or under the current directory if no argument is supplied. It displays various information about each file. The second command-line argument can be used to specify characters that control the value assigned to the *flags* argument when calling **nftw**().

Program source

```
#define _XOPEN_SOURCE 500
```

```
#include <ftw.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int
display_info(const char *fpath, const struct stat *sb,
             int tflag, struct FTW *ftwbuf)
{
    printf("%-3s %2d ",
           (tflag == FTW_D) ?   "d"   : (tflag == FTW_DNR) ? "dnr" :
           (tflag == FTW_DP) ?  "dp"  : (tflag == FTW_F) ?   "f" :
           (tflag == FTW_NS) ?  "ns"  : (tflag == FTW_SL) ?  "sl" :
           (tflag == FTW_SLN) ? "sln" : "???",
           ftwbuf->level);

    if (tflag == FTW_NS)
        printf("-------");
    else
        printf("%7jd", (intmax_t) sb->st_size);

    printf("   %-40s %d %s\n",
           fpath, ftwbuf->base, fpath + ftwbuf->base);

    return 0;           /* To tell nftw() to continue */
}

int
main(int argc, char *argv[])
{
    int flags = 0;

    if (argc > 2 && strchr(argv[2], 'd') != NULL)
        flags |= FTW_DEPTH;
    if (argc > 2 && strchr(argv[2], 'p') != NULL)
        flags |= FTW_PHYS;

    if (nftw((argc < 2) ? "." : argv[1], display_info, 20, flags)
        == -1)
    {
        perror("nftw");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

stat(2), *fts*(3), *readdir*(3)

NAME

futimes, lutimes − change file timestamps

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/time.h>

int futimes(int *fd***, const struct timeval** *tv***[2]);**
int lutimes(const char * *filename***, const struct timeval** *tv***[2]);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

futimes(), **lutimes**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

futimes() changes the access and modification times of a file in the same way as *utimes*(2), with the difference that the file whose timestamps are to be changed is specified via a file descriptor, *fd*, rather than via a pathname.

lutimes() changes the access and modification times of a file in the same way as *utimes*(2), with the difference that if *filename* refers to a symbolic link, then the link is not dereferenced: instead, the timestamps of the symbolic link are changed.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

Errors are as for *utimes*(2), with the following additions for **futimes**():

EBADF
fd is not a valid file descriptor.

ENOSYS
The */proc* filesystem could not be accessed.

The following additional error may occur for **lutimes**():

ENOSYS
The kernel does not support this call; Linux 2.6.22 or later is required.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
futimes(), **lutimes**()	Thread safety	MT-Safe

STANDARDS

Linux, BSD.

HISTORY

futimes()
glibc 2.3.

lutimes()
glibc 2.6.

NOTES

lutimes() is implemented using the *utimensat*(2) system call.

SEE ALSO

utime(2), *utimensat*(2), *symlink*(7)

NAME

fwide − set and determine the orientation of a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int fwide(FILE ****stream*, int** *mode***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fwide():
_XOPEN_SOURCE >= 500 || _ISOC99_SOURCE
|| _POSIX_C_SOURCE >= 200112L

DESCRIPTION

When *mode* is zero, the **fwide**() function determines the current orientation of *stream*. It returns a positive value if *stream* is wide-character oriented, that is, if wide-character I/O is permitted but char I/O is disallowed. It returns a negative value if *stream* is byte oriented—that is, if char I/O is permitted but wide-character I/O is disallowed. It returns zero if *stream* has no orientation yet; in this case the next I/O operation might change the orientation (to byte oriented if it is a char I/O operation, or to wide-character oriented if it is a wide-character I/O operation).

Once a stream has an orientation, it cannot be changed and persists until the stream is closed.

When *mode* is nonzero, the **fwide**() function first attempts to set *stream*'s orientation (to wide-character oriented if *mode* is greater than 0, or to byte oriented if *mode* is less than 0). It then returns a value denoting the current orientation, as above.

RETURN VALUE

The **fwide**() function returns the stream's orientation, after possibly changing it. A positive return value means wide-character oriented. A negative return value means byte oriented. A return value of zero means undecided.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

Wide-character output to a byte oriented stream can be performed through the *fprintf*(3) function with the **%lc** and **%ls** directives.

Char oriented output to a wide-character oriented stream can be performed through the *fwprintf*(3) function with the **%c** and **%s** directives.

SEE ALSO

fprintf(3), *fwprintf*(3)

NAME

gamma, gammaf, gammal − (logarithm of the) gamma function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

[[deprecated]] double gamma(double x);
[[deprecated]] float gammaf(float x);
[[deprecated]] long double gammal(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gamma():
_XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

gammaf(), **gammal**():
_XOPEN_SOURCE >= 600 || (_XOPEN_SOURCE && _ISOC99_SOURCE)
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions are deprecated: instead, use either the *tgamma*(3) or the *lgamma*(3) functions, as appropriate.

For the definition of the Gamma function, see *tgamma*(3).

***BSD version**

The libm in 4.4BSD and some versions of FreeBSD had a **gamma**() function that computes the Gamma function, as one would expect.

glibc version

glibc has a **gamma**() function that is equivalent to *lgamma*(3) and computes the natural logarithm of the Gamma function.

RETURN VALUE

See *lgamma*(3).

ERRORS

See *lgamma*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gamma(), **gammaf**(), **gammal**()	Thread safety	MT-Unsafe race:signgam

STANDARDS

None.

HISTORY

SVID 2.

Because of historical variations in behavior across systems, this function is not specified in any recent standard.

4.2BSD had a **gamma**() that computed ln(|Gamma(|*x*|)|), leaving the sign of Gamma(|*x*|) in the external integer *signgam*. In 4.3BSD the name was changed to *lgamma*(3), and the man page promises

> "At some time in the future the name gamma will be rehabilitated and used for the Gamma function"

This did indeed happen in 4.4BSD, where **gamma**() computes the Gamma function (with no effect on *signgam*). However, this came too late, and we now have *tgamma*(3), the "true gamma" function.

SEE ALSO

lgamma(3), *signgam*(3), *tgamma*(3)

NAME

gcvt − convert a floating-point number to a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

char *gcvt(double number, int ndigit, char *buf);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gcvt():
- Since glibc 2.17
  - (_XOPEN_SOURCE >= 500 && ! (_POSIX_C_SOURCE >= 200809L))
    - || /* glibc >= 2.20 */ _DEFAULT_SOURCE
    - || /* glibc <= 2.19 */ _SVID_SOURCE
- glibc 2.12 to glibc 2.16:
  - (_XOPEN_SOURCE >= 500 && ! (_POSIX_C_SOURCE >= 200112L))
    - || _SVID_SOURCE
- Before glibc 2.12:
  - _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

The **gcvt**() function converts *number* to a minimal length null-terminated ASCII string and stores the result in *buf*. It produces *ndigit* significant digits in either *printf*(3) F format or E format.

RETURN VALUE

The **gcvt**() function returns *buf*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gcvt()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

Marked as LEGACY in POSIX.1-2001. POSIX.1-2008 removed it, recommending the use of *sprintf*(3) instead (though *snprintf*(3) may be preferable).

SEE ALSO

ecvt(3), *fcvt*(3), *sprintf*(3)

NAME

_Generic − type-generic selection

SYNOPSIS

_Generic(*expression***, type1:** e1**, ...** /***, default:** e */**);**

DESCRIPTION

_Generic() evaluates the path of code under the type selector that is compatible with the type of the controlling *expression*, or **default:** if no type is compatible.

expression is not evaluated.

This is especially useful for writing type-generic macros, that will behave differently depending on the type of the argument.

STANDARDS

C11.

HISTORY

C11.

EXAMPLES

The following program demonstrates how to write a replacement for the standard *imaxabs*(3) function, which being a function can't really provide what it promises: seamlessly upgrading to the widest available type.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define my_imaxabs  _Generic(INTMAX_C(0),  \
    long:           labs,                  \
    long long:      llabs                  \
/*  long long long: lllabs */              \
)

int
main(void)
{
    off_t  a;

    a = -42;
    printf("imaxabs(%jd) == %jd\n", (intmax_t) a, my_imaxabs(a));
    printf("&imaxabs == %p\n", &my_imaxabs);
    printf("&labs    == %p\n", &labs);
    printf("&llabs   == %p\n", &llabs);

    exit(EXIT_SUCCESS);
}
```

NAME

get_nprocs, get_nprocs_conf − get number of processors

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sysinfo.h>

int get_nprocs(void);
int get_nprocs_conf(void);

DESCRIPTION

The function **get_nprocs_conf**() returns the number of processors configured by the operating system.

The function **get_nprocs**() returns the number of processors currently available in the system. This may be less than the number returned by **get_nprocs_conf**() because processors may be offline (e.g., on hotpluggable systems).

RETURN VALUE

As given in DESCRIPTION.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
get_nprocs(), **get_nprocs_conf**()	Thread safety	MT-Safe

STANDARDS

GNU.

NOTES

The current implementation of these functions is rather expensive, since they open and parse files in the */sys* filesystem each time they are called.

The following *sysconf*(3) calls make use of the functions documented on this page to return the same information.

```
np = sysconf(_SC_NPROCESSORS_CONF);     /* processors configured */
np = sysconf(_SC_NPROCESSORS_ONLN);     /* processors available */
```

EXAMPLES

The following example shows how **get_nprocs**() and **get_nprocs_conf**() can be used.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>

int
main(void)
{
    printf("This system has %d processors configured and "
           "%d processors available.\n",
           get_nprocs_conf(), get_nprocs());
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

nproc(1)

NAME

get_phys_pages, get_avphys_pages − get total and available physical page counts

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sysinfo.h>

long get_phys_pages(void);
long get_avphys_pages(void);

DESCRIPTION

The function **get_phys_pages**() returns the total number of physical pages of memory available on the system.

The function **get_avphys_pages**() returns the number of currently available physical pages of memory on the system.

RETURN VALUE

On success, these functions return a nonnegative value as given in DESCRIPTION. On failure, they return −1 and set *errno* to indicate the error.

ERRORS

ENOSYS

The system could not provide the required information (possibly because the */proc* filesystem was not mounted).

STANDARDS

GNU.

HISTORY

Before glibc 2.23, these functions obtained the required information by scanning the *MemTotal* and *MemFree* fields of */proc/meminfo*. Since glibc 2.23, these functions obtain the required information by calling *sysinfo*(2).

NOTES

The following *sysconf*(3) calls provide a portable means of obtaining the same information as the functions described on this page.

```
total_pages = sysconf(_SC_PHYS_PAGES);    /* total pages */
avl_pages = sysconf(_SC_AVPHYS_PAGES);    /* available pages */
```

EXAMPLES

The following example shows how **get_phys_pages**() and **get_avphys_pages**() can be used.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>

int
main(void)
{
    printf("This system has %ld pages of physical memory and "
           "%ld pages of physical memory available.\n",
           get_phys_pages(), get_avphys_pages());
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

sysconf(3)

NAME

getaddrinfo, freeaddrinfo, gai_strerror − network address and service translation

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict node,
                const char *restrict service,
                const struct addrinfo *restrict hints,
                struct addrinfo **restrict res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getaddrinfo(), **freeaddrinfo**(), **gai_strerror**():
Since glibc 2.22:
_POSIX_C_SOURCE >= 200112L
glibc 2.21 and earlier:
_POSIX_C_SOURCE

DESCRIPTION

Given *node* and *service*, which identify an Internet host and a service, **getaddrinfo**() returns one or more *addrinfo* structures, each of which contains an Internet address that can be specified in a call to *bind*(2) or *connect*(2). The **getaddrinfo**() function combines the functionality provided by the *gethostbyname*(3) and *getservbyname*(3) functions into a single interface, but unlike the latter functions, **getaddrinfo**() is reentrant and allows programs to eliminate IPv4-versus-IPv6 dependencies.

The *addrinfo* structure used by **getaddrinfo**() contains the following fields:

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

The *hints* argument points to an *addrinfo* structure that specifies criteria for selecting the socket address structures returned in the list pointed to by *res*. If *hints* is not NULL it points to an *addrinfo* structure whose *ai_family*, *ai_socktype*, and *ai_protocol* specify criteria that limit the set of socket addresses returned by **getaddrinfo**(), as follows:

ai_family
: This field specifies the desired address family for the returned addresses. Valid values for this field include **AF_INET** and **AF_INET6**. The value **AF_UNSPEC** indicates that **getaddrinfo**() should return socket addresses for any address family (either IPv4 or IPv6, for example) that can be used with *node* and *service*.

ai_socktype
: This field specifies the preferred socket type, for example **SOCK_STREAM** or **SOCK_DGRAM**. Specifying 0 in this field indicates that socket addresses of any type can be returned by **getaddrinfo**().

ai_protocol
: This field specifies the protocol for the returned socket addresses. Specifying 0 in this field indicates that socket addresses with any protocol can be returned by **getaddrinfo**().

ai_flags
: This field specifies additional options, described below. Multiple flags are specified by bitwise OR-ing them together.

All the other fields in the structure pointed to by *hints* must contain either 0 or a null pointer, as appropriate.

Specifying *hints* as NULL is equivalent to setting *ai_socktype* and *ai_protocol* to 0; *ai_family* to **AF_UNSPEC**; and *ai_flags* to **(AI_V4MAPPED | AI_ADDRCONFIG)**. (POSIX specifies different defaults for *ai_flags*; see NOTES.) *node* specifies either a numerical network address (for IPv4, numbers-and-dots notation as supported by *inet_aton*(3); for IPv6, hexadecimal string format as supported by *inet_pton*(3)), or a network hostname, whose network addresses are looked up and resolved. If *hints.ai_flags* contains the **AI_NUMERICHOST** flag, then *node* must be a numerical network address. The **AI_NUMERICHOST** flag suppresses any potentially lengthy network host address lookups.

If the **AI_PASSIVE** flag is specified in *hints.ai_flags*, and *node* is NULL, then the returned socket addresses will be suitable for *bind*(2)ing a socket that will *accept*(2) connections. The returned socket address will contain the "wildcard address" (**INADDR_ANY** for IPv4 addresses, **IN6ADDR_ANY_INIT** for IPv6 address). The wildcard address is used by applications (typically servers) that intend to accept connections on any of the host's network addresses. If *node* is not NULL, then the **AI_PASSIVE** flag is ignored.

If the **AI_PASSIVE** flag is not set in *hints.ai_flags*, then the returned socket addresses will be suitable for use with *connect*(2), *sendto*(2), or *sendmsg*(2). If *node* is NULL, then the network address will be set to the loopback interface address (**INADDR_LOOPBACK** for IPv4 addresses, **IN6ADDR_LOOPBACK_INIT** for IPv6 address); this is used by applications that intend to communicate with peers running on the same host.

service sets the port in each returned address structure. If this argument is a service name (see *services*(5)), it is translated to the corresponding port number. This argument can also be specified as a decimal number, which is simply converted to binary. If *service* is NULL, then the port number of the returned socket addresses will be left uninitialized. If **AI_NUMERICSERV** is specified in *hints.ai_flags* and *service* is not NULL, then *service* must point to a string containing a numeric port number. This flag is used to inhibit the invocation of a name resolution service in cases where it is known not to be required.

Either *node* or *service*, but not both, may be NULL.

The **getaddrinfo**() function allocates and initializes a linked list of *addrinfo* structures, one for each network address that matches *node* and *service*, subject to any restrictions imposed by *hints*, and returns a pointer to the start of the list in *res*. The items in the linked list are linked by the *ai_next* field.

There are several reasons why the linked list may have more than one *addrinfo* structure, including: the network host is multihomed, accessible over multiple protocols (e.g., both **AF_INET** and **AF_INET6**); or the same service is available from multiple socket types (one **SOCK_STREAM** address and another **SOCK_DGRAM** address, for example). Normally, the application should try using the addresses in the order in which they are returned. The sorting function used within **getaddrinfo**() is defined in RFC 3484; the order can be tweaked for a particular system by editing */etc/gai.conf* (available since glibc 2.5).

If *hints.ai_flags* includes the **AI_CANONNAME** flag, then the *ai_canonname* field of the first of the *addrinfo* structures in the returned list is set to point to the official name of the host.

The remaining fields of each returned *addrinfo* structure are initialized as follows:

- The *ai_family*, *ai_socktype*, and *ai_protocol* fields return the socket creation parameters (i.e., these fields have the same meaning as the corresponding arguments of *socket*(2)). For example, *ai_family* might return **AF_INET** or **AF_INET6**; *ai_socktype* might return **SOCK_DGRAM** or **SOCK_STREAM**; and *ai_protocol* returns the protocol for the socket.
- A pointer to the socket address is placed in the *ai_addr* field, and the length of the socket address, in bytes, is placed in the *ai_addrlen* field.

If *hints.ai_flags* includes the **AI_ADDRCONFIG** flag, then IPv4 addresses are returned in the list pointed to by *res* only if the local system has at least one IPv4 address configured, and IPv6 addresses

are returned only if the local system has at least one IPv6 address configured. The loopback address is not considered for this case as valid as a configured address. This flag is useful on, for example, IPv4-only systems, to ensure that **getaddrinfo**() does not return IPv6 socket addresses that would always fail in *connect*(2) or *bind*(2).

If *hints.ai_flags* specifies the **AI_V4MAPPED** flag, and *hints.ai_family* was specified as **AF_INET6**, and no matching IPv6 addresses could be found, then return IPv4-mapped IPv6 addresses in the list pointed to by *res*. If both **AI_V4MAPPED** and **AI_ALL** are specified in *hints.ai_flags*, then return both IPv6 and IPv4-mapped IPv6 addresses in the list pointed to by *res*. **AI_ALL** is ignored if **AI_V4MAPPED** is not also specified.

The **freeaddrinfo**() function frees the memory that was allocated for the dynamically allocated linked list *res*.

Extensions to getaddrinfo() for Internationalized Domain Names

Starting with glibc 2.3.4, **getaddrinfo**() has been extended to selectively allow the incoming and outgoing hostnames to be transparently converted to and from the Internationalized Domain Name (IDN) format (see RFC 3490, *Internationalizing Domain Names in Applications (IDNA)*). Four new flags are defined:

AI_IDN
: If this flag is specified, then the node name given in *node* is converted to IDN format if necessary. The source encoding is that of the current locale.

  If the input name contains non-ASCII characters, then the IDN encoding is used. Those parts of the node name (delimited by dots) that contain non-ASCII characters are encoded using ASCII Compatible Encoding (ACE) before being passed to the name resolution functions.

AI_CANONIDN
: After a successful name lookup, and if the **AI_CANONNAME** flag was specified, **getaddrinfo**() will return the canonical name of the node corresponding to the *addrinfo* structure value passed back. The return value is an exact copy of the value returned by the name resolution function.

  If the name is encoded using ACE, then it will contain the *xn−−* prefix for one or more components of the name. To convert these components into a readable form the **AI_CANONIDN** flag can be passed in addition to **AI_CANONNAME**. The resulting string is encoded using the current locale's encoding.

AI_IDN_ALLOW_UNASSIGNED
AI_IDN_USE_STD3_ASCII_RULES
: Setting these flags will enable the IDNA_ALLOW_UNASSIGNED (allow unassigned Unicode code points) and IDNA_USE_STD3_ASCII_RULES (check output to make sure it is a STD3 conforming hostname) flags respectively to be used in the IDNA handling.

RETURN VALUE

getaddrinfo() returns 0 if it succeeds, or one of the following nonzero error codes:

EAI_ADDRFAMILY
: The specified network host does not have any network addresses in the requested address family.

EAI_AGAIN
: The name server returned a temporary failure indication. Try again later.

EAI_BADFLAGS
: *hints.ai_flags* contains invalid flags; or, *hints.ai_flags* included **AI_CANONNAME** and *node* was NULL.

EAI_FAIL
: The name server returned a permanent failure indication.

EAI_FAMILY
: The requested address family is not supported.

EAI_MEMORY
: Out of memory.

EAI_NODATA
The specified network host exists, but does not have any network addresses defined.

EAI_NONAME
The *node* or *service* is not known; or both *node* and *service* are NULL; or **AI_NUMERICSERV** was specified in *hints.ai_flags* and *service* was not a numeric port-number string.

EAI_SERVICE
The requested service is not available for the requested socket type. It may be available through another socket type. For example, this error could occur if *service* was "shell" (a service available only on stream sockets), and either *hints.ai_protocol* was **IPPROTO_UDP**, or *hints.ai_socktype* was **SOCK_DGRAM**; or the error could occur if *service* was not NULL, and *hints.ai_socktype* was **SOCK_RAW** (a socket type that does not support the concept of services).

EAI_SOCKTYPE
The requested socket type is not supported. This could occur, for example, if *hints.ai_socktype* and *hints.ai_protocol* are inconsistent (e.g., **SOCK_DGRAM** and **IPPROTO_TCP**, respectively).

EAI_SYSTEM
Other system error; *errno* is set to indicate the error.

The **gai_strerror**() function translates these error codes to a human readable string, suitable for error reporting.

FILES

/etc/gai.conf

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getaddrinfo()	Thread safety	MT-Safe env locale
freeaddrinfo(), **gai_strerror**()	Thread safety	MT-Safe

VERSIONS

According to POSIX.1, specifying *hints* as NULL should cause *ai_flags* to be assumed as 0. The GNU C library instead assumes a value of **(AI_V4MAPPED | AI_ADDRCONFIG)** for this case, since this value is considered an improvement on the specification.

STANDARDS

POSIX.1-2008.

getaddrinfo()
RFC 2553.

HISTORY

POSIX.1-2001.

AI_ADDRCONFIG
AI_ALL
AI_V4MAPPED
glibc 2.3.3.

AI_NUMERICSERV
glibc 2.3.4.

NOTES

getaddrinfo() supports the *address***%***scope-id* notation for specifying the IPv6 scope-ID.

EXAMPLES

The following programs demonstrate the use of **getaddrinfo**(), **gai_strerror**(), **freeaddrinfo**(), and *getnameinfo*(3). The programs are an echo server and client for UDP datagrams.

Server program

```
#include <netdb.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define BUF_SIZE 500

int
main(int argc, char *argv[])
{
    int                     sfd, s;
    char                    buf[BUF_SIZE];
    ssize_t                 nread;
    socklen_t               peer_addrlen;
    struct addrinfo         hints;
    struct addrinfo         *result, *rp;
    struct sockaddr_storage  peer_addr;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
    hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
    hints.ai_protocol = 0;          /* Any protocol */
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;

    s = getaddrinfo(NULL, argv[1], &hints, &result);
    if (s != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(EXIT_FAILURE);
    }

    /* getaddrinfo() returns a list of address structures.
       Try each address until we successfully bind(2).
       If socket(2) (or bind(2)) fails, we (close the socket
       and) try the next address. */

    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype,
                     rp->ai_protocol);
        if (sfd == -1)
            continue;

        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
            break;                  /* Success */

        close(sfd);
    }

    freeaddrinfo(result);           /* No longer needed */

    if (rp == NULL) {               /* No address succeeded */
```

```
            fprintf(stderr, "Could not bind\n");
            exit(EXIT_FAILURE);
        }

        /* Read datagrams and echo them back to sender. */

        for (;;) {
            char host[NI_MAXHOST], service[NI_MAXSERV];

            peer_addrlen = sizeof(peer_addr);
            nread = recvfrom(sfd, buf, BUF_SIZE, 0,
                             (struct sockaddr *) &peer_addr, &peer_addrlen);
            if (nread == -1)
                continue;               /* Ignore failed request */

            s = getnameinfo((struct sockaddr *) &peer_addr,
                            peer_addrlen, host, NI_MAXHOST,
                            service, NI_MAXSERV, NI_NUMERICSERV);
            if (s == 0)
                printf("Received %zd bytes from %s:%s\n",
                       nread, host, service);
            else
                fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));

            if (sendto(sfd, buf, nread, 0, (struct sockaddr *) &peer_addr,
                       peer_addrlen) != nread)
            {
                fprintf(stderr, "Error sending response\n");
            }
        }
    }
```

Client program

```
    #include <netdb.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <sys/socket.h>
    #include <sys/types.h>
    #include <unistd.h>

    #define BUF_SIZE 500

    int
    main(int argc, char *argv[])
    {
        int              sfd, s;
        char             buf[BUF_SIZE];
        size_t           len;
        ssize_t          nread;
        struct addrinfo  hints;
        struct addrinfo  *result, *rp;

        if (argc < 3) {
            fprintf(stderr, "Usage: %s host port msg...\n", argv[0]);
            exit(EXIT_FAILURE);
        }
```

```
/* Obtain address(es) matching host/port. */

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
hints.ai_flags = 0;
hints.ai_protocol = 0;          /* Any protocol */

s = getaddrinfo(argv[1], argv[2], &hints, &result);
if (s != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures.
   Try each address until we successfully connect(2).
   If socket(2) (or connect(2)) fails, we (close the socket
   and) try the next address. */

for (rp = result; rp != NULL; rp = rp->ai_next) {
    sfd = socket(rp->ai_family, rp->ai_socktype,
                 rp->ai_protocol);
    if (sfd == -1)
        continue;

    if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
        break;                  /* Success */

    close(sfd);
}

freeaddrinfo(result);           /* No longer needed */

if (rp == NULL) {               /* No address succeeded */
    fprintf(stderr, "Could not connect\n");
    exit(EXIT_FAILURE);
}

/* Send remaining command-line arguments as separate
   datagrams, and read responses from server. */

for (size_t j = 3; j < argc; j++) {
    len = strlen(argv[j]) + 1;
            /* +1 for terminating null byte */

    if (len > BUF_SIZE) {
        fprintf(stderr,
                "Ignoring long message in argument %zu\n", j);
        continue;
    }

    if (write(sfd, argv[j], len) != len) {
        fprintf(stderr, "partial/failed write\n");
        exit(EXIT_FAILURE);
    }

    nread = read(sfd, buf, BUF_SIZE);
    if (nread == -1) {
        perror("read");
```

```
                exit(EXIT_FAILURE);
            }

            printf("Received %zd bytes: %s\n", nread, buf);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getaddrinfo_a(3), *gethostbyname*(3), *getnameinfo*(3), *inet*(3), *gai.conf*(5), *hostname*(7), *ip*(7)

NAME

getaddrinfo_a, gai_suspend, gai_error, gai_cancel − asynchronous network address and service translation

LIBRARY

Asynchronous name lookup library (*libanl*, *−lanl*)

SYNOPSIS

```
#define _GNU_SOURCE       /* See feature_test_macros(7) */
#include <netdb.h>

int getaddrinfo_a(int mode, struct gaicb *list[restrict],
                  int nitems, struct sigevent *restrict sevp);
int gai_suspend(const struct gaicb *const list[], int nitems,
                  const struct timespec *timeout);

int gai_error(struct gaicb *req);
int gai_cancel(struct gaicb *req);
```

DESCRIPTION

The **getaddrinfo_a**() function performs the same task as *getaddrinfo*(3), but allows multiple name look-ups to be performed asynchronously, with optional notification on completion of look-up operations.

The *mode* argument has one of the following values:

GAI_WAIT
: Perform the look-ups synchronously. The call blocks until the look-ups have completed.

GAI_NOWAIT
: Perform the look-ups asynchronously. The call returns immediately, and the requests are resolved in the background. See the discussion of the *sevp* argument below.

The array *list* specifies the look-up requests to process. The *nitems* argument specifies the number of elements in *list*. The requested look-up operations are started in parallel. NULL elements in *list* are ignored. Each request is described by a *gaicb* structure, defined as follows:

```
struct gaicb {
    const char            *ar_name;
    const char            *ar_service;
    const struct addrinfo *ar_request;
    struct addrinfo       *ar_result;
};
```

The elements of this structure correspond to the arguments of *getaddrinfo*(3). Thus, *ar_name* corresponds to the *node* argument and *ar_service* to the *service* argument, identifying an Internet host and a service. The *ar_request* element corresponds to the *hints* argument, specifying the criteria for selecting the returned socket address structures. Finally, *ar_result* corresponds to the *res* argument; you do not need to initialize this element, it will be automatically set when the request is resolved. The *addrinfo* structure referenced by the last two elements is described in *getaddrinfo*(3).

When *mode* is specified as **GAI_NOWAIT**, notifications about resolved requests can be obtained by employing the *sigevent* structure pointed to by the *sevp* argument. For the definition and general details of this structure, see *sigevent*(3type). The *sevp−>sigev_notify* field can have the following values:

SIGEV_NONE
: Don't provide any notification.

SIGEV_SIGNAL
: When a look-up completes, generate the signal *sigev_signo* for the process. See *sigevent*(3type) for general details. The *si_code* field of the *siginfo_t* structure will be set to **SI_ASYNCNL**.

SIGEV_THREAD
: When a look-up completes, invoke *sigev_notify_function* as if it were the start function of a new thread. See *sigevent*(3type) for details.

For **SIGEV_SIGNAL** and **SIGEV_THREAD**, it may be useful to point *sevp−>sigev_value.sival_ptr*

to *list*.

The **gai_suspend**() function suspends execution of the calling thread, waiting for the completion of one or more requests in the array *list*. The *nitems* argument specifies the size of the array *list*. The call blocks until one of the following occurs:

- One or more of the operations in *list* completes.
- The call is interrupted by a signal that is caught.
- The time interval specified in *timeout* elapses. This argument specifies a timeout in seconds plus nanoseconds (see *nanosleep*(2) for details of the *timespec* structure). If *timeout* is NULL, then the call blocks indefinitely (until one of the events above occurs).

No explicit indication of which request was completed is given; you must determine which request(s) have completed by iterating with **gai_error**() over the list of requests.

The **gai_error**() function returns the status of the request *req*: either **EAI_INPROGRESS** if the request was not completed yet, 0 if it was handled successfully, or an error code if the request could not be resolved.

The **gai_cancel**() function cancels the request *req*. If the request has been canceled successfully, the error status of the request will be set to **EAI_CANCELED** and normal asynchronous notification will be performed. The request cannot be canceled if it is currently being processed; in that case, it will be handled as if **gai_cancel**() has never been called. If *req* is NULL, an attempt is made to cancel all outstanding requests that the process has made.

RETURN VALUE

The **getaddrinfo_a**() function returns 0 if all of the requests have been enqueued successfully, or one of the following nonzero error codes:

EAI_AGAIN
: The resources necessary to enqueue the look-up requests were not available. The application may check the error status of each request to determine which ones failed.

EAI_MEMORY
: Out of memory.

EAI_SYSTEM
: *mode* is invalid.

The **gai_suspend**() function returns 0 if at least one of the listed requests has been completed. Otherwise, it returns one of the following nonzero error codes:

EAI_AGAIN
: The given timeout expired before any of the requests could be completed.

EAI_ALLDONE
: There were no actual requests given to the function.

EAI_INTR
: A signal has interrupted the function. Note that this interruption might have been caused by signal notification of some completed look-up request.

The **gai_error**() function can return **EAI_INPROGRESS** for an unfinished look-up request, 0 for a successfully completed look-up (as described above), one of the error codes that could be returned by *getaddrinfo*(3), or the error code **EAI_CANCELED** if the request has been canceled explicitly before it could be finished.

The **gai_cancel**() function can return one of these values:

EAI_CANCELED
: The request has been canceled successfully.

EAI_NOTCANCELED
: The request has not been canceled.

EAI_ALLDONE
: The request has already completed.

The *gai_strerror*(3) function translates these error codes to a human readable string, suitable for error

reporting.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getaddrinfo_a(), **gai_suspend**(), **gai_error**(), **gai_cancel**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.2.3.

The interface of **getaddrinfo_a**() was modeled after the *lio_listio*(3) interface.

EXAMPLES

Two examples are provided: a simple example that resolves several requests in parallel synchronously, and a complex example showing some of the asynchronous capabilities.

Synchronous example

The program below simply resolves several hostnames in parallel, giving a speed-up compared to resolving the hostnames sequentially using *getaddrinfo*(3). The program might be used like this:

```
$ ./a.out mirrors.kernel.org enoent.linuxfoundation.org gnu.org
mirrors.kernel.org: 139.178.88.99
enoent.linuxfoundation.org: Name or service not known
gnu.org: 209.51.188.116
```

Here is the program source code

```
#define _GNU_SOURCE
#include <err.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC(n, type)  ((type *) reallocarray(NULL, n, sizeof(type)))

int
main(int argc, char *argv[])
{
    int ret;
    struct gaicb *reqs[argc - 1];
    char host[NI_MAXHOST];
    struct addrinfo *res;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s HOST...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < argc - 1; i++) {
        reqs[i] = MALLOC(1, struct gaicb);
        if (reqs[i] == NULL)
            err(EXIT_FAILURE, "malloc");

        memset(reqs[i], 0, sizeof(*reqs[0]));
        reqs[i]->ar_name = argv[i + 1];
    }

    ret = getaddrinfo_a(GAI_WAIT, reqs, argc - 1, NULL);
    if (ret != 0) {
        fprintf(stderr, "getaddrinfo_a() failed: %s\n",
```

```
                gai_strerror(ret));
            exit(EXIT_FAILURE);
        }

        for (size_t i = 0; i < argc - 1; i++) {
            printf("%s: ", reqs[i]->ar_name);
            ret = gai_error(reqs[i]);
            if (ret == 0) {
                res = reqs[i]->ar_result;

                ret = getnameinfo(res->ai_addr, res->ai_addrlen,
                                  host, sizeof(host),
                                  NULL, 0, NI_NUMERICHOST);
                if (ret != 0) {
                    fprintf(stderr, "getnameinfo() failed: %s\n",
                            gai_strerror(ret));
                    exit(EXIT_FAILURE);
                }
                puts(host);

            } else {
                puts(gai_strerror(ret));
            }
        }
        exit(EXIT_SUCCESS);
    }
```

Asynchronous example

This example shows a simple interactive **getaddrinfo_a**() front-end. The notification facility is not demonstrated.

An example session might look like this:

```
$ ./a.out
> a mirrors.kernel.org enoent.linuxfoundation.org gnu.org
> c 2
[2] gnu.org: Request not canceled
> w 0 1
[00] mirrors.kernel.org: Finished
> l
[00] mirrors.kernel.org: 139.178.88.99
[01] enoent.linuxfoundation.org: Processing request in progress
[02] gnu.org: 209.51.188.116
> l
[00] mirrors.kernel.org: 139.178.88.99
[01] enoent.linuxfoundation.org: Name or service not known
[02] gnu.org: 209.51.188.116
```

The program source is as follows:

```
#define _GNU_SOURCE
#include <assert.h>
#include <err.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CALLOC(n, type)  ((type *) calloc(n, sizeof(type)))

#define REALLOCF(ptr, n, type)                                  \
({                                                              \
```

```
    static_assert(__builtin_types_compatible_p(typeof(ptr), type *));   \
                                                                        \
    (type *) reallocarrayf(ptr, n, sizeof(type));                       \
})

static struct gaicb **reqs = NULL;
static size_t nreqs = 0;

static inline void *
reallocarrayf(void *p, size_t nmemb, size_t size)
{
    void  *q;

    q = reallocarray(p, nmemb, size);
    if (q == NULL && nmemb != 0 && size != 0)
        free(p);
    return q;
}

static char *
getcmd(void)
{
    static char buf[256];

    fputs("> ", stdout); fflush(stdout);
    if (fgets(buf, sizeof(buf), stdin) == NULL)
        return NULL;

    if (buf[strlen(buf) - 1] == '\n')
        buf[strlen(buf) - 1] = 0;

    return buf;
}

/* Add requests for specified hostnames. */
static void
add_requests(void)
{
    size_t nreqs_base = nreqs;
    char *host;
    int ret;

    while ((host = strtok(NULL, " "))) {
        nreqs++;
        reqs = REALLOCF(reqs, nreqs, struct gaicb *);
        if (reqs == NULL)
            err(EXIT_FAILURE, "reallocf");

        reqs[nreqs - 1] = CALLOC(1, struct gaicb);
        if (reqs[nreqs - 1] == NULL)
            err(EXIT_FAILURE, "calloc");

        reqs[nreqs - 1]->ar_name = strdup(host);
    }

    /* Queue nreqs_base..nreqs requests. */

    ret = getaddrinfo_a(GAI_NOWAIT, &reqs[nreqs_base],
                        nreqs - nreqs_base, NULL);
```

```
        if (ret) {
            fprintf(stderr, "getaddrinfo_a() failed: %s\n",
                    gai_strerror(ret));
            exit(EXIT_FAILURE);
        }
    }

    /* Wait until at least one of specified requests completes. */
    static void
    wait_requests(void)
    {
        char *id;
        int ret;
        size_t n;
        struct gaicb const **wait_reqs;

        wait_reqs = CALLOC(nreqs, const struct gaicb *);
        if (wait_reqs == NULL)
            err(EXIT_FAILURE, "calloc");

                /* NULL elements are ignored by gai_suspend(). */

        while ((id = strtok(NULL, " ")) != NULL) {
            n = atoi(id);

            if (n >= nreqs) {
                printf("Bad request number: %s\n", id);
                return;
            }

            wait_reqs[n] = reqs[n];
        }

        ret = gai_suspend(wait_reqs, nreqs, NULL);
        if (ret) {
            printf("gai_suspend(): %s\n", gai_strerror(ret));
            return;
        }

        for (size_t i = 0; i < nreqs; i++) {
            if (wait_reqs[i] == NULL)
                continue;

            ret = gai_error(reqs[i]);
            if (ret == EAI_INPROGRESS)
                continue;

            printf("[%02zu] %s: %s\n", i, reqs[i]->ar_name,
                   ret == 0 ? "Finished" : gai_strerror(ret));
        }
    }

    /* Cancel specified requests. */
    static void
    cancel_requests(void)
    {
        char *id;
        int ret;
        size_t n;
```

```
        while ((id = strtok(NULL, " ")) != NULL) {
            n = atoi(id);

            if (n >= nreqs) {
                printf("Bad request number: %s\n", id);
                return;
            }

            ret = gai_cancel(reqs[n]);
            printf("[%s] %s: %s\n", id, reqs[atoi(id)]->ar_name,
                   gai_strerror(ret));
        }
    }

    /* List all requests. */
    static void
    list_requests(void)
    {
        int ret;
        char host[NI_MAXHOST];
        struct addrinfo *res;

        for (size_t i = 0; i < nreqs; i++) {
            printf("[%02zu] %s: ", i, reqs[i]->ar_name);
            ret = gai_error(reqs[i]);

            if (!ret) {
                res = reqs[i]->ar_result;

                ret = getnameinfo(res->ai_addr, res->ai_addrlen,
                                  host, sizeof(host),
                                  NULL, 0, NI_NUMERICHOST);
                if (ret) {
                    fprintf(stderr, "getnameinfo() failed: %s\n",
                            gai_strerror(ret));
                    exit(EXIT_FAILURE);
                }
                puts(host);
            } else {
                puts(gai_strerror(ret));
            }
        }
    }

    int
    main(void)
    {
        char *cmdline;
        char *cmd;

        while ((cmdline = getcmd()) != NULL) {
            cmd = strtok(cmdline, " ");

            if (cmd == NULL) {
                list_requests();
            } else {
                switch (cmd[0]) {
                case 'a':
```

```
                    add_requests();
                    break;
                case 'w':
                    wait_requests();
                    break;
                case 'c':
                    cancel_requests();
                    break;
                case 'l':
                    list_requests();
                    break;
                default:
                    fprintf(stderr, "Bad command: %c\n", cmd[0]);
                    break;
                }
            }
        }
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getaddrinfo(3), *inet*(3), *lio_listio*(3), *hostname*(7), *ip*(7), *sigevent*(3type)

NAME

getauxval − retrieve a value from the auxiliary vector

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/auxv.h>

unsigned long getauxval(unsigned long *type***);**

DESCRIPTION

The **getauxval**() function retrieves values from the auxiliary vector, a mechanism that the kernel's ELF binary loader uses to pass certain information to user space when a program is executed.

Each entry in the auxiliary vector consists of a pair of values: a type that identifies what this entry represents, and a value for that type. Given the argument *type*, **getauxval**() returns the corresponding value.

The value returned for each *type* is given in the following list. Not all *type* values are present on all architectures.

AT_BASE
: The base address of the program interpreter (usually, the dynamic linker).

AT_BASE_PLATFORM
: A pointer to a string (PowerPC and MIPS only). On PowerPC, this identifies the real platform; may differ from **AT_PLATFORM**. On MIPS, this identifies the ISA level (since Linux 5.7).

AT_CLKTCK
: The frequency with which *times*(2) counts. This value can also be obtained via *sysconf(_SC_CLK_TCK)*.

AT_DCACHEBSIZE
: The data cache block size.

AT_EGID
: The effective group ID of the thread.

AT_ENTRY
: The entry address of the executable.

AT_EUID
: The effective user ID of the thread.

AT_EXECFD
: File descriptor of program.

AT_EXECFN
: A pointer to a string containing the pathname used to execute the program.

AT_FLAGS
: Flags (unused).

AT_FPUCW
: Used FPU control word (SuperH architecture only). This gives some information about the FPU initialization performed by the kernel.

AT_GID
: The real group ID of the thread.

AT_HWCAP
: An architecture and ABI dependent bit-mask whose settings indicate detailed processor capabilities. The contents of the bit mask are hardware dependent (for example, see the kernel source file *arch/x86/include/asm/cpufeature.h* for details relating to the Intel x86 architecture; the value returned is the first 32-bit word of the array described there). A human-readable version of the same information is available via */proc/cpuinfo*.

AT_HWCAP2 (since glibc 2.18)
Further machine-dependent hints about processor capabilities.

AT_ICACHEBSIZE
The instruction cache block size.

AT_L1D_CACHEGEOMETRY
Geometry of the L1 data cache, encoded with the cache line size in bytes in the bottom 16 bits and the cache associativity in the next 16 bits. The associativity is such that if N is the 16-bit value, the cache is N-way set associative.

AT_L1D_CACHESIZE
The L1 data cache size.

AT_L1I_CACHEGEOMETRY
Geometry of the L1 instruction cache, encoded as for **AT_L1D_CACHEGEOMETRY**.

AT_L1I_CACHESIZE
The L1 instruction cache size.

AT_L2_CACHEGEOMETRY
Geometry of the L2 cache, encoded as for **AT_L1D_CACHEGEOMETRY**.

AT_L2_CACHESIZE
The L2 cache size.

AT_L3_CACHEGEOMETRY
Geometry of the L3 cache, encoded as for **AT_L1D_CACHEGEOMETRY**.

AT_L3_CACHESIZE
The L3 cache size.

AT_PAGESZ
The system page size (the same value returned by *sysconf(_SC_PAGESIZE)*).

AT_PHDR
The address of the program headers of the executable.

AT_PHENT
The size of program header entry.

AT_PHNUM
The number of program headers.

AT_PLATFORM
A pointer to a string that identifies the hardware platform that the program is running on. The dynamic linker uses this in the interpretation of *rpath* values.

AT_RANDOM
The address of sixteen bytes containing a random value.

AT_SECURE
Has a nonzero value if this executable should be treated securely. Most commonly, a nonzero value indicates that the process is executing a set-user-ID or set-group-ID binary (so that its real and effective UIDs or GIDs differ from one another), or that it gained capabilities by executing a binary file that has capabilities (see *capabilities*(7)). Alternatively, a nonzero value may be triggered by a Linux Security Module. When this value is nonzero, the dynamic linker disables the use of certain environment variables (see *ld-linux.so*(8)) and glibc changes other aspects of its behavior. (See also *secure_getenv*(3).)

AT_SYSINFO
The entry point to the system call function in the vDSO. Not present/needed on all architectures (e.g., absent on x86-64).

AT_SYSINFO_EHDR
The address of a page containing the virtual Dynamic Shared Object (vDSO) that the kernel creates in order to provide fast implementations of certain system calls.

AT_UCACHEBSIZE
The unified cache block size.

AT_UID
The real user ID of the thread.

RETURN VALUE

On success, **getauxval**() returns the value corresponding to *type*. If *type* is not found, 0 is returned.

ERRORS

ENOENT (since glibc 2.19)
No entry corresponding to *type* could be found in the auxiliary vector.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getauxval()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.16.

NOTES

The primary consumer of the information in the auxiliary vector is the dynamic linker, *ld-linux.so*(8). The auxiliary vector is a convenient and efficient shortcut that allows the kernel to communicate a certain set of standard information that the dynamic linker usually or always needs. In some cases, the same information could be obtained by system calls, but using the auxiliary vector is cheaper.

The auxiliary vector resides just above the argument list and environment in the process address space. The auxiliary vector supplied to a program can be viewed by setting the **LD_SHOW_AUXV** environment variable when running a program:

```
$ LD_SHOW_AUXV=1 sleep 1
```

The auxiliary vector of any process can (subject to file permissions) be obtained via */proc/*pid*/auxv*; see *proc*(5) for more information.

BUGS

Before the addition of the **ENOENT** error in glibc 2.19, there was no way to unambiguously distinguish the case where *type* could not be found from the case where the value corresponding to *type* was zero.

SEE ALSO

execve(2), *secure_getenv*(3), *vdso*(7), *ld-linux.so*(8)

NAME

getcontext, setcontext − get or set the user context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <ucontext.h>

int getcontext(ucontext_t *_ucp_);
int setcontext(const ucontext_t *_ucp_);

DESCRIPTION

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), *makecontext*(3), and *swapcontext*(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext_t {
    struct ucontext_t *uc_link;
    sigset_t          uc_sigmask;
    stack_t           uc_stack;
    mcontext_t        uc_mcontext;
    ...
} ucontext_t;
```

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using *makecontext*(3)), *uc_sigmask* is the set of signals blocked in this context (see *sigprocmask*(2)), *uc_stack* is the stack used by this context (see *sigaltstack*(2)), and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed to by *ucp* to the currently active context.

The function **setcontext**() restores the user context pointed to by *ucp*. A successful call does not return. The context should have been obtained by a call of **getcontext**(), or *makecontext*(3), or received as the third argument to a signal handler (see the discussion of the **SA_SIGINFO** flag in *sigaction*(2)).

If the context was obtained by a call of **getcontext**(), program execution continues as if this call just returned.

If the context was obtained by a call of *makecontext*(3), program execution continues by a call to the function *func* specified as the second argument of that call to *makecontext*(3). When the function *func* returns, we continue with the *uc_link* member of the structure *ucp* specified as the first argument of that call to *makecontext*(3). When this member is NULL, the thread exits.

If the context was obtained by a call to a signal handler, then old standard text says that "program execution continues with the program instruction following the instruction interrupted by the signal". However, this sentence was removed in SUSv2, and the present verdict is "the result is unspecified".

RETURN VALUE

When successful, **getcontext**() returns 0 and **setcontext**() does not return. On error, both return −1 and set *errno* to indicate the error.

ERRORS

None defined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getcontext(), **setcontext**()	Thread safety	MT-Safe race:ucp

STANDARDS

None.

HISTORY

SUSv2, POSIX.1-2001.

POSIX.1-2008 removes these functions, citing portability issues, and recommending that applications be rewritten to use POSIX threads instead.

NOTES

The earliest incarnation of this mechanism was the *setjmp*(3)/*longjmp*(3) mechanism. Since that does not define the handling of the signal context, the next stage was the *sigsetjmp*(3)/*siglongjmp*(3) pair. The present mechanism gives much more control. On the other hand, there is no easy way to detect whether a return from **getcontext**() is from the first call, or via a **setcontext**() call. The user has to invent their own bookkeeping device, and a register variable won't do since registers are restored.

When a signal occurs, the current user context is saved and a new context is created by the kernel for the signal handler. Do not leave the handler using *longjmp*(3): it is undefined what would happen with contexts. Use *siglongjmp*(3) or **setcontext**() instead.

SEE ALSO

sigaction(2), *sigaltstack*(2), *sigprocmask*(2), *longjmp*(3), *makecontext*(3), *sigsetjmp*(3), *signal*(7)

NAME

getcwd, getwd, get_current_dir_name − get current working directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char buf[.size], size_t size);
char *get_current_dir_name(void);

[[deprecated]] char *getwd(char buf[PATH_MAX]);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

get_current_dir_name():
_GNU_SOURCE

getwd():
Since glibc 2.12:
(_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE
Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument *buf*, if present.

The **getcwd**() function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds *size* bytes, NULL is returned, and *errno* is set to **ERANGE**; an application should check for this error, and allocate a larger buffer if necessary.

As an extension to the POSIX.1-2001 standard, glibc's **getcwd**() allocates the buffer dynamically using *malloc*(3) if *buf* is NULL. In this case, the allocated buffer has the length *size* unless *size* is zero, when *buf* is allocated as big as necessary. The caller should *free*(3) the returned buffer.

get_current_dir_name() will *malloc*(3) an array big enough to hold the absolute pathname of the current working directory. If the environment variable **PWD** is set, and its value is correct, then that value will be returned. The caller should *free*(3) the returned buffer.

getwd() does not *malloc*(3) any memory. The *buf* argument should be a pointer to an array at least **PATH_MAX** bytes long. If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds **PATH_MAX** bytes, NULL is returned, and *errno* is set to **ENAMETOOLONG**. (Note that on some systems, **PATH_MAX** may not be a compile-time constant; furthermore, its value may depend on the filesystem, see *pathconf*(3).) For portability and security reasons, use of **getwd**() is deprecated.

RETURN VALUE

On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case of **getcwd**() and **getwd**() this is the same value as *buf*.

On failure, these functions return NULL, and *errno* is set to indicate the error. The contents of the array pointed to by *buf* are undefined on error.

ERRORS

EACCES
Permission to read or search a component of the filename was denied.

EFAULT
buf points to a bad address.

EINVAL
The *size* argument is zero and *buf* is not a null pointer.

EINVAL
getwd(): *buf* is NULL.

ENAMETOOLONG
getwd(): The size of the null-terminated absolute pathname string exceeds **PATH_MAX** bytes.

ENOENT
The current working directory has been unlinked.

ENOMEM
Out of memory.

ERANGE
The *size* argument is less than the length of the absolute pathname of the working directory, including the terminating null byte. You need to allocate a bigger array and try again.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getcwd(), **getwd**()	Thread safety	MT-Safe
get_current_dir_name()	Thread safety	MT-Safe env

VERSIONS

POSIX.1-2001 leaves the behavior of **getcwd**() unspecified if *buf* is NULL.

POSIX.1-2001 does not define any errors for **getwd**().

VERSIONS

C library/kernel differences

On Linux, the kernel provides a **getcwd**() system call, which the functions described in this page will use if possible. The system call takes the same arguments as the library function of the same name, but is limited to returning at most **PATH_MAX** bytes. (Before Linux 3.12, the limit on the size of the returned pathname was the system page size. On many architectures, **PATH_MAX** and the system page size are both 4096 bytes, but a few architectures have a larger page size.) If the length of the pathname of the current working directory exceeds this limit, then the system call fails with the error **ENAMETOOLONG**. In this case, the library functions fall back to a (slower) alternative implementation that returns the full pathname.

Following a change in Linux 2.6.36, the pathname returned by the **getcwd**() system call will be prefixed with the string "(unreachable)" if the current directory is not below the root directory of the current process (e.g., because the process set a new filesystem root using *chroot*(2) without changing its current directory into the new root). Such behavior can also be caused by an unprivileged user by changing the current directory into another mount namespace. When dealing with pathname from untrusted sources, callers of the functions described in this page should consider checking whether the returned pathname starts with '/' or '(' to avoid misinterpreting an unreachable path as a relative pathname.

STANDARDS

getcwd()
POSIX.1-2008.

get_current_dir_name()
GNU.

getwd()
None.

HISTORY

getcwd()
POSIX.1-2001.

getwd()
POSIX.1-2001, but marked LEGACY. Removed in POSIX.1-2008. Use **getcwd**() instead.

Under Linux, these functions make use of the **getcwd**() system call (available since Linux 2.1.92). On older systems they would query */proc/self/cwd*. If both system call and proc filesystem are missing, a generic implementation is called. Only in that case can these calls fail under Linux with **EACCES**.

NOTES

These functions are often used to save the location of the current working directory for the purpose of returning to it later. Opening the current directory (".") and calling *fchdir*(2) to return is usually a faster and more reliable alternative when sufficiently many file descriptors are available, especially on platforms other than Linux.

BUGS

Since the Linux 2.6.36 change that added "(unreachable)" in the circumstances described above, the glibc implementation of **getcwd**() has failed to conform to POSIX and returned a relative pathname when the API contract requires an absolute pathname. With glibc 2.27 onwards this is corrected; calling **getcwd**() from such a pathname will now result in failure with **ENOENT**.

SEE ALSO

pwd(1), *chdir*(2), *fchdir*(2), *open*(2), *unlink*(2), *free*(3), *malloc*(3)

NAME

getdate, getdate_r − convert a date-plus-time string to broken-down time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <time.h>

struct tm *getdate(const char *_string_);

extern int getdate_err;

int getdate_r(const char *restrict *string***, struct tm *restrict** *res***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getdate():
_XOPEN_SOURCE >= 500

getdate_r():
_GNU_SOURCE

DESCRIPTION

The function **getdate**() converts a string representation of a date and time, contained in the buffer pointed to by *string*, into a broken-down time. The broken-down time is stored in a *tm* structure, and a pointer to this structure is returned as the function result. This *tm* structure is allocated in static storage, and consequently it will be overwritten by further calls to **getdate**().

In contrast to *strptime*(3), (which has a *format* argument), **getdate**() uses the formats found in the file whose full pathname is given in the environment variable **DATEMSK**. The first line in the file that matches the given input string is used for the conversion.

The matching is done case insensitively. Superfluous whitespace, either in the pattern or in the string to be converted, is ignored.

The conversion specifications that a pattern can contain are those given for *strptime*(3). One more conversion specification is specified in POSIX.1-2001:

%Z Timezone name. This is not implemented in glibc.

When **%Z** is given, the structure containing the broken-down time is initialized with values corresponding to the current time in the given timezone. Otherwise, the structure is initialized to the broken-down time corresponding to the current local time (as by a call to *localtime*(3)).

When only the day of the week is given, the day is taken to be the first such day on or after today.

When only the month is given (and no year), the month is taken to be the first such month equal to or after the current month. If no day is given, it is the first day of the month.

When no hour, minute, and second are given, the current hour, minute, and second are taken.

If no date is given, but we know the hour, then that hour is taken to be the first such hour equal to or after the current hour.

getdate_r() is a GNU extension that provides a reentrant version of **getdate**(). Rather than using a global variable to report errors and a static buffer to return the broken down time, it returns errors via the function result value, and returns the resulting broken-down time in the caller-allocated buffer pointed to by the argument *res*.

RETURN VALUE

When successful, **getdate**() returns a pointer to a *struct tm*. Otherwise, it returns NULL and sets the global variable *getdate_err* to one of the error numbers shown below. Changes to *errno* are unspecified.

On success **getdate_r**() returns 0; on error it returns one of the error numbers shown below.

ERRORS

The following errors are returned via *getdate_err* (for *getdate*()) or as the function result (for *getdate_r*()):

1 The **DATEMSK** environment variable is not defined, or its value is an empty string.

2 The template file specified by **DATEMSK** cannot be opened for reading.

3 Failed to get file status information.

4 The template file is not a regular file.

5 An error was encountered while reading the template file.

6 Memory allocation failed (not enough memory available).

7 There is no line in the file that matches the input.

8 Invalid input specification.

ENVIRONMENT

DATEMSK
File containing format patterns.

TZ
LC_TIME
Variables used by *strptime*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getdate()	Thread safety	MT-Unsafe race:getdate env locale
getdate_r()	Thread safety	MT-Safe env locale

VERSIONS

The POSIX.1 specification for *strptime*(3) contains conversion specifications using the **%E** or **%O** modifier, while such specifications are not given for **getdate**(). In glibc, **getdate**() is implemented using *strptime*(3), so that precisely the same conversions are supported by both.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The program below calls **getdate**() for each of its command-line arguments, and for each call displays the values in the fields of the returned *tm* structure. The following shell session demonstrates the operation of the program:

```
$ TFILE=$PWD/tfile
$ echo '%A' > $TFILE       # Full name of the day of the week
$ echo '%T' >> $TFILE      # Time (HH:MM:SS)
$ echo '%F' >> $TFILE      # ISO date (YYYY-MM-DD)
$ date
$ export DATEMSK=$TFILE
$ ./a.out Tuesday '2009-12-28' '12:22:33'
Sun Sep  7 06:03:36 CEST 2008
Call 1 ("Tuesday") succeeded:
    tm_sec   = 36
    tm_min   = 3
    tm_hour  = 6
    tm_mday  = 9
    tm_mon   = 8
    tm_year  = 108
    tm_wday  = 2
    tm_yday  = 252
    tm_isdst = 1
Call 2 ("2009-12-28") succeeded:
    tm_sec   = 36
    tm_min   = 3
```

```
            tm_hour  = 6
            tm_mday  = 28
            tm_mon   = 11
            tm_year  = 109
            tm_wday  = 1
            tm_yday  = 361
            tm_isdst = 0
        Call 3 ("12:22:33") succeeded:
            tm_sec   = 33
            tm_min   = 22
            tm_hour  = 12
            tm_mday  = 7
            tm_mon   = 8
            tm_year  = 108
            tm_wday  = 0
            tm_yday  = 250
            tm_isdst = 1
```

Program source

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    struct tm *tmp;

    for (size_t j = 1; j < argc; j++) {
        tmp = getdate(argv[j]);

        if (tmp == NULL) {
            printf("Call %zu failed; getdate_err = %d\n",
                   j, getdate_err);
            continue;
        }

        printf("Call %zu (\"%s\") succeeded:\n", j, argv[j]);
        printf("    tm_sec   = %d\n", tmp->tm_sec);
        printf("    tm_min   = %d\n", tmp->tm_min);
        printf("    tm_hour  = %d\n", tmp->tm_hour);
        printf("    tm_mday  = %d\n", tmp->tm_mday);
        printf("    tm_mon   = %d\n", tmp->tm_mon);
        printf("    tm_year  = %d\n", tmp->tm_year);
        printf("    tm_wday  = %d\n", tmp->tm_wday);
        printf("    tm_yday  = %d\n", tmp->tm_yday);
        printf("    tm_isdst = %d\n", tmp->tm_isdst);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

time(2), *localtime*(3), *setlocale*(3), *strftime*(3), *strptime*(3)

NAME

getdirentries − get directory entries in a filesystem-independent format

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <dirent.h>

ssize_t getdirentries(int fd, char buf[restrict .nbytes], size_t nbytes,
                      off_t *restrict basep);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getdirentries():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Read directory entries from the directory specified by *fd* into *buf*. At most *nbytes* are read. Reading starts at offset **basep*, and **basep* is updated with the new position after reading.

RETURN VALUE

getdirentries() returns the number of bytes read or zero when at the end of the directory. If an error occurs, −1 is returned, and *errno* is set to indicate the error.

ERRORS

See the Linux library source code for details.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getdirentries()	Thread safety	MT-Safe

STANDARDS

BSD.

NOTES

Use *opendir*(3) and *readdir*(3) instead.

SEE ALSO

lseek(2), *open*(2)

NAME

getdtablesize − get file descriptor table size

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int getdtablesize(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getdtablesize():
Since glibc 2.20:
_DEFAULT_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)
glibc 2.12 to glibc 2.19:
_BSD_SOURCE || ! (_POSIX_C_SOURCE >= 200112L)
Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

getdtablesize() returns the maximum number of files a process can have open, one more than the largest possible value for a file descriptor.

RETURN VALUE

The current limit on the number of open files per process.

ERRORS

On Linux, **getdtablesize**() can return any of the errors described for *getrlimit*(2); see NOTES below.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getdtablesize()	Thread safety	MT-Safe

VERSIONS

The glibc version of **getdtablesize**() calls *getrlimit*(2) and returns the current **RLIMIT_NOFILE** limit, or **OPEN_MAX** when that fails.

Portable applications should employ *sysconf(_SC_OPEN_MAX)* instead of this call.

STANDARDS

None.

HISTORY

SVr4, 4.4BSD (first appeared in 4.2BSD).

SEE ALSO

close(2), *dup*(2), *getrlimit*(2), *open*(2)

NAME

getentropy − fill a buffer with random bytes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int getentropy(void *buffer***[.**_length_**], size_t** *length***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getentropy():
_DEFAULT_SOURCE

DESCRIPTION

The **getentropy**() function writes *length* bytes of high-quality random data to the buffer starting at the location pointed to by *buffer*. The maximum permitted value for the *length* argument is 256.

A successful call to **getentropy**() always provides the requested number of bytes of entropy.

RETURN VALUE

On success, this function returns zero. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EFAULT
Part or all of the buffer specified by *buffer* and *length* is not in valid addressable memory.

EIO *length* is greater than 256.

EIO An unspecified error occurred while trying to overwrite *buffer* with random data.

ENOSYS
This kernel version does not implement the *getrandom*(2) system call required to implement this function.

STANDARDS

None.

HISTORY

glibc 2.25. OpenBSD.

NOTES

The **getentropy**() function is implemented using *getrandom*(2).

Whereas the glibc wrapper makes *getrandom*(2) a cancelation point, **getentropy**() is not a cancelation point.

getentropy() is also declared in **<sys/random.h>**. (No feature test macro need be defined to obtain the declaration from that header file.)

A call to **getentropy**() may block if the system has just booted and the kernel has not yet collected enough randomness to initialize the entropy pool. In this case, **getentropy**() will keep blocking even if a signal is handled, and will return only once the entropy pool has been initialized.

SEE ALSO

getrandom(2), *urandom*(4), *random*(7)

NAME

getenv, secure_getenv − get an environment variable

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
char *secure_getenv(const char *name);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

secure_getenv():
_GNU_SOURCE

DESCRIPTION

The **getenv**() function searches the environment list to find the environment variable *name*, and returns a pointer to the corresponding *value* string.

The GNU-specific **secure_getenv**() function is just like **getenv**() except that it returns NULL in cases where "secure execution" is required. Secure execution is required if one of the following conditions was true when the program run by the calling process was loaded:

- the process's effective user ID did not match its real user ID or the process's effective group ID did not match its real group ID (typically this is the result of executing a set-user-ID or set-group-ID program);
- the effective capability bit was set on the executable file; or
- the process has a nonempty permitted capability set.

Secure execution may also be required if triggered by some Linux security modules.

The **secure_getenv**() function is intended for use in general-purpose libraries to avoid vulnerabilities that could occur if set-user-ID or set-group-ID programs accidentally trusted the environment.

RETURN VALUE

The **getenv**() function returns a pointer to the value in the environment, or NULL if there is no match.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getenv(), **secure_getenv**()	Thread safety	MT-Safe env

STANDARDS

getenv()
C11, POSIX.1-2008.

secure_getenv()
GNU.

HISTORY

getenv()
POSIX.1-2001, C89, C99, SVr4, 4.3BSD.

secure_getenv()
glibc 2.17.

NOTES

The strings in the environment list are of the form *name=value*.

As typically implemented, **getenv**() returns a pointer to a string within the environment list. The caller must take care not to modify this string, since that would change the environment of the process.

The implementation of **getenv**() is not required to be reentrant. The string pointed to by the return value of **getenv**() may be statically allocated, and can be modified by a subsequent call to **getenv**(), *putenv*(3), *setenv*(3), or *unsetenv*(3).

The "secure execution" mode of **secure_getenv**() is controlled by the **AT_SECURE** flag contained in the auxiliary vector passed from the kernel to user space.

SEE ALSO

clearenv(3), *getauxval*(3), *putenv*(3), *setenv*(3), *unsetenv*(3), *capabilities*(7), *environ*(7)

NAME

getfsent, getfsspec, getfsfile, setfsent, endfsent − handle fstab entries

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <fstab.h>

int setfsent(void);
struct fstab *getfsent(void);
void endfsent(void);

struct fstab *getfsfile(const char *_mount_point_);
struct fstab *getfsspec(const char *_special_file_);

DESCRIPTION

These functions read from the file */etc/fstab*. The *struct fstab* is defined by:

```
struct fstab {
    char       *fs_spec;       /* block device name */
    char       *fs_file;       /* mount point */
    char       *fs_vfstype;    /* filesystem type */
    char       *fs_mntops;     /* mount options */
    const char *fs_type;       /* rw/rq/ro/sw/xx option */
    int         fs_freq;       /* dump frequency, in days */
    int         fs_passno;     /* pass number on parallel dump */
};
```

Here the field *fs_type* contains (on a *BSD system) one of the five strings "rw", "rq", "ro", "sw", "xx" (read-write, read-write with quota, read-only, swap, ignore).

The function **setfsent**() opens the file when required and positions it at the first line.

The function **getfsent**() parses the next line from the file. (After opening it when required.)

The function **endfsent**() closes the file when required.

The function **getfsspec**() searches the file from the start and returns the first entry found for which the *fs_spec* field matches the *special_file* argument.

The function **getfsfile**() searches the file from the start and returns the first entry found for which the *fs_file* field matches the *mount_point* argument.

RETURN VALUE

Upon success, the functions **getfsent**(), **getfsfile**(), and **getfsspec**() return a pointer to a *struct fstab*, while **setfsent**() returns 1. Upon failure or end-of-file, these functions return NULL and 0, respectively.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
endfsent(), **setfsent**()	Thread safety	MT-Unsafe race:fsent
getfsent(), **getfsspec**(), **getfsfile**()	Thread safety	MT-Unsafe race:fsent locale

VERSIONS

Several operating systems have these functions, for example, *BSD, SunOS, Digital UNIX, AIX (which also has a *getfstype*())HP-UX has functions of the same names, that however use a *struct checklist* instead of a *struct fstab*, and calls these functions obsolete, superseded by *getmntent*(3).

STANDARDS

None.

HISTORY

The **getfsent**() function appeared in 4.0BSD; the other four functions appeared in 4.3BSD.

NOTES

These functions are not thread-safe.

Since Linux allows mounting a block special device in several places, and since several devices can

have the same mount point, where the last device with a given mount point is the interesting one, while **getfsfile**() and **getfsspec**() only return the first occurrence, these two functions are not suitable for use under Linux.

SEE ALSO

getmntent(3), *fstab*(5)

NAME

getgrent, setgrent, endgrent − get group file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrent(void);

void setgrent(void);
void endgrent(void);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setgrent():
_XOPEN_SOURCE >= 500
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

getgrent(), **endgrent**():
Since glibc 2.22:
_XOPEN_SOURCE >= 500 || _DEFAULT_SOURCE
glibc 2.21 and earlier
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getgrent**() function returns a pointer to a structure containing the broken-out fields of a record in the group database (e.g., the local group file */etc/group*, NIS, and LDAP). The first time **getgrent**() is called, it returns the first entry; thereafter, it returns successive entries.

The **setgrent**() function rewinds to the beginning of the group database, to allow repeated scans.

The **endgrent**() function is used to close the group database after all processing has been performed.

The *group* structure is defined in <*grp.h*> as follows:

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* NULL-terminated array of pointers
                               to names of group members */
};
```

For more information about the fields of this structure, see *group*(5).

RETURN VALUE

The **getgrent**() function returns a pointer to a *group* structure, or NULL if there are no more entries or an error occurs.

Upon error, *errno* may be set. If one wants to check *errno* after the call, it should be set to zero before the call.

The return value may point to a static area, and may be overwritten by subsequent calls to **getgrent**(), *getgrgid*(3), or *getgrnam*(3). (Do not pass the returned pointer to *free*(3).)

ERRORS

EAGAIN
The service was temporarily unavailable; try again later. For NSS backends in glibc this indicates a temporary error talking to the backend. The error may correct itself, retrying later is suggested.

EINTR
A signal was caught; see *signal*(7).

EIO I/O error.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
A necessary input file cannot be found. For NSS backends in glibc this indicates the backend is not correctly configured.

ENOMEM
Insufficient memory to allocate *group* structure.

ERANGE
Insufficient buffer space supplied.

FILES

/etc/group
local group database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getgrent()	Thread safety	MT-Unsafe race:grent race:grentbuf locale
setgrent(), **endgrent**()	Thread safety	MT-Unsafe race:grent locale

In the above table, *grent* in *race:grent* signifies that if any of the functions **setgrent**(), **getgrent**(), or **endgrent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

fgetgrent(3), *getgrent_r*(3), *getgrgid*(3), *getgrnam*(3), *getgrouplist*(3), *putgrent*(3), *group*(5)

NAME

getgrent_r, fgetgrent_r − get group file entry reentrantly

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <grp.h>

int getgrent_r(struct group *restrict gbuf,
               char buf[restrict .buflen], size_t buflen,
               struct group **restrict gbufp);
int fgetgrent_r(FILE *restrict stream, struct group *restrict gbuf,
               char buf[restrict .buflen], size_t buflen,
               struct group **restrict gbufp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getgrent_r():
_GNU_SOURCE

fgetgrent_r():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

The functions **getgrent_r**() and **fgetgrent_r**() are the reentrant versions of *getgrent*(3) and *fgetgrent*(3). The former reads the next group entry from the stream initialized by *setgrent*(3). The latter reads the next group entry from *stream*.

The *group* structure is defined in <*grp.h*> as follows:

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* NULL-terminated array of pointers
                               to names of group members */
};
```

For more information about the fields of this structure, see *group*(5).

The nonreentrant functions return a pointer to static storage, where this static storage contains further pointers to group name, password, and members. The reentrant functions described here return all of that in caller-provided buffers. First of all there is the buffer *gbuf* that can hold a *struct group*. And next the buffer *buf* of size *buflen* that can hold additional strings. The result of these functions, the *struct group* read from the stream, is stored in the provided buffer **gbuf*, and a pointer to this *struct group* is returned in **gbufp*.

RETURN VALUE

On success, these functions return 0 and **gbufp* is a pointer to the *struct group*. On error, these functions return an error value and **gbufp* is NULL.

ERRORS

ENOENT
No more entries.

ERANGE
Insufficient buffer space supplied. Try again with larger buffer.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getgrent_r()	Thread safety	MT-Unsafe race:grent locale
fgetgrent_r()	Thread safety	MT-Safe

In the above table, *grent* in *race:grent* signifies that if any of the functions *setgrent*(3), *getgrent*(3), *endgrent*(3), or **getgrent_r**() are used in parallel in different threads of a program, then data races could occur.

VERSIONS

Other systems use the prototype

```
struct group *getgrent_r(struct group *grp, char *buf,
                         int buflen);
```

or, better,

```
int getgrent_r(struct group *grp, char *buf, int buflen,
               FILE **gr_fp);
```

STANDARDS

GNU.

HISTORY

These functions are done in a style resembling the POSIX version of functions like *getpwnam_r*(3).

NOTES

The function **getgrent_r**() is not really reentrant since it shares the reading position in the stream with all other threads.

EXAMPLES

```
#define _GNU_SOURCE
#include <grp.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFLEN 4096

int
main(void)
{
    struct group grp;
    struct group *grpp;
    char buf[BUFLEN];
    int i;

    setgrent();
    while (1) {
        i = getgrent_r(&grp, buf, sizeof(buf), &grpp);
        if (i)
            break;
        printf("%s (%jd):", grpp->gr_name, (intmax_t) grpp->gr_gid);
        for (size_t j = 0; ; j++) {
            if (grpp->gr_mem[j] == NULL)
                break;
            printf(" %s", grpp->gr_mem[j]);
        }
        printf("\n");
    }
    endgrent();
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fgetgrent(3), *getgrent*(3), *getgrgid*(3), *getgrnam*(3), *putgrent*(3), *group*(5)

NAME

getgrnam, getgrnam_r, getgrgid, getgrgid_r − get group file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);

int getgrnam_r(const char *restrict name, struct group *restrict grp,
        char buf[restrict .buflen], size_t buflen,
        struct group **restrict result);
int getgrgid_r(gid_t gid, struct group *restrict grp,
        char buf[restrict .buflen], size_t buflen,
        struct group **restrict result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getgrnam_r(), **getgrgid_r**():
_POSIX_C_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getgrnam**() function returns a pointer to a structure containing the broken-out fields of the record in the group database (e.g., the local group file */etc/group*, NIS, and LDAP) that matches the group name *name*.

The **getgrgid**() function returns a pointer to a structure containing the broken-out fields of the record in the group database that matches the group ID *gid*.

The *group* structure is defined in <*grp.h*> as follows:

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* NULL-terminated array of pointers
                               to names of group members */
};
```

For more information about the fields of this structure, see *group*(5).

The **getgrnam_r**() and **getgrgid_r**() functions obtain the same information as **getgrnam**() and **getgrgid**(), but store the retrieved *group* structure in the space pointed to by *grp*. The string fields pointed to by the members of the *group* structure are stored in the buffer *buf* of size *buflen*. A pointer to the result (in case of success) or NULL (in case no entry was found or an error occurred) is stored in **result*.

The call

```
sysconf(_SC_GETGR_R_SIZE_MAX)
```

returns either −1, without changing *errno*, or an initial suggested size for *buf*. (If this size is too small, the call fails with **ERANGE**, in which case the caller can retry with a larger buffer.)

RETURN VALUE

The **getgrnam**() and **getgrgid**() functions return a pointer to a *group* structure, or NULL if the matching entry is not found or an error occurs. If an error occurs, *errno* is set to indicate the error. If one wants to check *errno* after the call, it should be set to zero before the call.

The return value may point to a static area, and may be overwritten by subsequent calls to *getgrent*(3), **getgrgid**(), or **getgrnam**(). (Do not pass the returned pointer to *free*(3).)

On success, **getgrnam_r**() and **getgrgid_r**() return zero, and set **result* to *grp*. If no matching group record was found, these functions return 0 and store NULL in **result*. In case of error, an error number is returned, and NULL is stored in **result*.

ERRORS

0 or **ENOENT** or **ESRCH** or **EBADF** or **EPERM** or ...
The given *name* or *gid* was not found.

EINTR
A signal was caught; see *signal*(7).

EIO I/O error.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
Insufficient memory to allocate *group* structure.

ERANGE
Insufficient buffer space supplied.

FILES

/etc/group
local group database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getgrnam()	Thread safety	MT-Unsafe race:grnam locale
getgrgid()	Thread safety	MT-Unsafe race:grgid locale
getgrnam_r(), **getgrgid_r**()	Thread safety	MT-Safe locale

VERSIONS

The formulation given above under "RETURN VALUE" is from POSIX.1. It does not call "not found" an error, hence does not specify what value *errno* might have in this situation. But that makes it impossible to recognize errors. One might argue that according to POSIX *errno* should be left unchanged if an entry is not found. Experiments on various UNIX-like systems show that lots of different values occur in this situation: 0, ENOENT, EBADF, ESRCH, EWOULDBLOCK, EPERM, and probably others.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

endgrent(3), *fgetgrent*(3), *getgrent*(3), *getpwnam*(3), *setgrent*(3), *group*(5)

NAME

getgrouplist − get list of groups to which a user belongs

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <grp.h>

int getgrouplist(const char *user, gid_t group,
                 gid_t *groups, int *ngroups);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getgrouplist():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **getgrouplist**() function scans the group database (see *group*(5)) to obtain the list of groups that *user* belongs to. Up to **ngroups* of these groups are returned in the array *groups*.

If it was not among the groups defined for *user* in the group database, then *group* is included in the list of groups returned by **getgrouplist**(); typically this argument is specified as the group ID from the password record for *user*.

The *ngroups* argument is a value-result argument: on return it always contains the number of groups found for *user*, including *group*; this value may be greater than the number of groups stored in *groups*.

RETURN VALUE

If the number of groups of which *user* is a member is less than or equal to **ngroups*, then the value **ngroups* is returned.

If the user is a member of more than **ngroups* groups, then **getgrouplist**() returns −1. In this case, the value returned in **ngroups* can be used to resize the buffer passed to a further call to **getgrouplist**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getgrouplist()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

glibc 2.2.4.

BUGS

Before glibc 2.3.3, the implementation of this function contains a buffer-overrun bug: it returns the complete list of groups for *user* in the array *groups*, even when the number of groups exceeds **ngroups*.

EXAMPLES

The program below displays the group list for the user named in its first command-line argument. The second command-line argument specifies the *ngroups* value to be supplied to **getgrouplist**(). The following shell session shows examples of the use of this program:

```
$ ./a.out cecilia 0
getgrouplist() returned -1; ngroups = 3
$ ./a.out cecilia 3
ngroups = 3
16 (dialout)
33 (video)
100 (users)
```

Program source

```
#include <grp.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int            ngroups;
    gid_t          *groups;
    struct group   *gr;
    struct passwd  *pw;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <user> <ngroups>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    ngroups = atoi(argv[2]);

    groups = malloc(sizeof(*groups) * ngroups);
    if (groups == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    /* Fetch passwd structure (contains first group ID for user). */

    pw = getpwnam(argv[1]);
    if (pw == NULL) {
        perror("getpwnam");
        exit(EXIT_SUCCESS);
    }

    /* Retrieve group list. */

    if (getgrouplist(argv[1], pw->pw_gid, groups, &ngroups) == -1) {
        fprintf(stderr, "getgrouplist() returned -1; ngroups = %d\n",
                ngroups);
        exit(EXIT_FAILURE);
    }

    /* Display list of retrieved groups, along with group names. */

    fprintf(stderr, "ngroups = %d\n", ngroups);
    for (int j = 0; j < ngroups; j++) {
        printf("%d", groups[j]);
        gr = getgrgid(groups[j]);
        if (gr != NULL)
            printf(" (%s)", gr->gr_name);
        printf("\n");
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getgroups(2), *setgroups*(2), *getgrent*(3), *group_member*(3), *group*(5), *passwd*(5)

NAME

gethostbyname, gethostbyaddr, sethostent, gethostent, endhostent, h_errno, herror, hstrerror, gethostbyaddr_r, gethostbyname2, gethostbyname2_r, gethostbyname_r, gethostent_r − get network host entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

void sethostent(int stayopen);
void endhostent(void);

[[deprecated]] extern int h_errno;

[[deprecated]] struct hostent *gethostbyname(const char *name);
[[deprecated]] struct hostent *gethostbyaddr(const void addr[.len],
                                socklen_t len, int type);

[[deprecated]] void herror(const char *s);
[[deprecated]] const char *hstrerror(int err);

/* System V/POSIX extension */
struct hostent *gethostent(void);

/* GNU extensions */
[[deprecated]]
struct hostent *gethostbyname2(const char *name, int af );

int gethostent_r(struct hostent *restrict ret,
                 char buf[restrict .buflen], size_t buflen,
                 struct hostent **restrict result,
                 int *restrict h_errnop);

[[deprecated]]
int gethostbyaddr_r(const void addr[restrict .len], socklen_t len,
                 int type,
                 struct hostent *restrict ret,
                 char buf[restrict .buflen], size_t buflen,
                 struct hostent **restrict result,
                 int *restrict h_errnop);
[[deprecated]]
int gethostbyname_r(const char *restrict name,
                 struct hostent *restrict ret,
                 char buf[restrict .buflen], size_t buflen,
                 struct hostent **restrict result,
                 int *restrict h_errnop);
[[deprecated]]
int gethostbyname2_r(const char *restrict name, int af,
                 struct hostent *restrict ret,
                 char buf[restrict .buflen], size_t buflen,
                 struct hostent **restrict result,
                 int *restrict h_errnop);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gethostbyname2(), **gethostent_r**(), **gethostbyaddr_r**(), **gethostbyname_r**(), **gethostbyname2_r**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc up to and including 2.19:
_BSD_SOURCE || _SVID_SOURCE

herror(), **hstrerror**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.8 to glibc 2.19:

_BSD_SOURCE || _SVID_SOURCE
Before glibc 2.8:
none

h_errno:
Since glibc 2.19
_DEFAULT_SOURCE || _POSIX_C_SOURCE < 200809L
glibc 2.12 to glibc 2.19:
_BSD_SOURCE || _SVID_SOURCE || _POSIX_C_SOURCE < 200809L
Before glibc 2.12:
none

DESCRIPTION

The **gethostbyname***(), **gethostbyaddr***(), **herror**(), and **hstrerror**() functions are obsolete. Applications should use *getaddrinfo*(3), *getnameinfo*(3), and *gai_strerror*(3) instead.

The **sethostent**() function specifies, if *stayopen* is true (1), that a connected TCP socket should be used for the name server queries and that the connection should remain open during successive queries. Otherwise, name server queries will use UDP datagrams.

The **endhostent**() function ends the use of a TCP connection for name server queries.

The **gethostbyname**() function returns a structure of type *hostent* for the given host *name*. Here *name* is either a hostname or an IPv4 address in standard dot notation (as for *inet_addr*(3)). If *name* is an IPv4 address, no lookup is performed and **gethostbyname**() simply copies *name* into the *h_name* field and its *struct in_addr* equivalent into the *h_addr_list[0]* field of the returned *hostent* structure. If *name* doesn't end in a dot and the environment variable **HOSTALIASES** is set, the alias file pointed to by **HOSTALIASES** will first be searched for *name* (see *hostname*(7) for the file format). The current domain and its parents are searched unless *name* ends in a dot.

The **gethostbyaddr**() function returns a structure of type *hostent* for the given host address *addr* of length *len* and address type *type*. Valid address types are **AF_INET** and **AF_INET6** (defined in <*sys/socket.h*>). The host address argument is a pointer to a struct of a type depending on the address type, for example a *struct in_addr* * (probably obtained via a call to *inet_addr*(3)) for address type **AF_INET**.

The (obsolete) **herror**() function prints the error message associated with the current value of *h_errno* on *stderr*.

The (obsolete) **hstrerror**() function takes an error number (typically *h_errno*) and returns the corresponding message string.

The domain name queries carried out by **gethostbyname**() and **gethostbyaddr**() rely on the Name Service Switch (**nsswitch.conf**(5)) configured sources or a local name server (**named**(8)). The default action is to query the Name Service Switch (**nsswitch.conf**(5)) configured sources, failing that, a local name server (**named**(8)).

Historical

The *nsswitch.conf*(5) file is the modern way of controlling the order of host lookups.

In glibc 2.4 and earlier, the *order* keyword was used to control the order of host lookups as defined in */etc/host.conf* (**host.conf**(5)).

The *hostent* structure is defined in <*netdb.h*> as follows:

```
struct hostent {
    char  *h_name;            /* official name of host */
    char **h_aliases;         /* alias list */
    int    h_addrtype;        /* host address type */
    int    h_length;          /* length of address */
    char **h_addr_list;       /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

The members of the *hostent* structure are:

h_name
The official name of the host.

h_aliases
An array of alternative names for the host, terminated by a null pointer.

h_addrtype
The type of address; always **AF_INET** or **AF_INET6** at present.

h_length
The length of the address in bytes.

h_addr_list
An array of pointers to network addresses for the host (in network byte order), terminated by a null pointer.

h_addr The first address in *h_addr_list* for backward compatibility.

RETURN VALUE

The **gethostbyname**() and **gethostbyaddr**() functions return the *hostent* structure or a null pointer if an error occurs. On error, the *h_errno* variable holds an error number. When non-NULL, the return value may point at static data, see the notes below.

ERRORS

The variable *h_errno* can have the following values:

HOST_NOT_FOUND
The specified host is unknown.

NO_DATA
The requested name is valid but does not have an IP address. Another type of request to the name server for this domain may return an answer. The constant **NO_ADDRESS** is a synonym for **NO_DATA**.

NO_RECOVERY
A nonrecoverable name server error occurred.

TRY_AGAIN
A temporary error occurred on an authoritative name server. Try again later.

FILES

/etc/host.conf
resolver configuration file

/etc/hosts
host database file

/etc/nsswitch.conf
name service switch configuration

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gethostbyname()	Thread safety	MT-Unsafe race:hostbyname env locale
gethostbyaddr()	Thread safety	MT-Unsafe race:hostbyaddr env locale
sethostent(), **endhostent**(), **gethostent_r**()	Thread safety	MT-Unsafe race:hostent env locale
herror(), **hstrerror**()	Thread safety	MT-Safe
gethostent()	Thread safety	MT-Unsafe race:hostent race:hostentbuf env locale
gethostbyname2()	Thread safety	MT-Unsafe race:hostbyname2 env locale
gethostbyaddr_r(), **gethostbyname_r**(), **gethostbyname2_r**()	Thread safety	MT-Safe env locale

In the above table, *hostent* in *race:hostent* signifies that if any of the functions **sethostent**(), **gethostent**(), **gethostent_r**(), or **endhostent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

sethostent()
endhostent()
gethostent()
POSIX.1-2008.

gethostent_r()
GNU.

Others: None.

HISTORY

sethostent()
endhostent()
gethostent()
POSIX.1-2001.

gethostbyname()
gethostbyaddr()
h_errno
Marked obsolescent in POSIX.1-2001. Removed in POSIX.1-2008, recommending the use of *getaddrinfo*(3) and *getnameinfo*(3) instead.

NOTES

The functions **gethostbyname**() and **gethostbyaddr**() may return pointers to static data, which may be overwritten by later calls. Copying the *struct hostent* does not suffice, since it contains pointers; a deep copy is required.

In the original BSD implementation the *len* argument of **gethostbyname**() was an *int*. The SUSv2 standard is buggy and declares the *len* argument of **gethostbyaddr**() to be of type *size_t*. (That is wrong, because it has to be *int*, and *size_t* is not. POSIX.1-2001 makes it *socklen_t*, which is OK.) See also *accept*(2).

The BSD prototype for **gethostbyaddr**() uses *const char* * for the first argument.

System V/POSIX extension

POSIX requires the **gethostent**() call, which should return the next entry in the host data base. When using DNS/BIND this does not make much sense, but it may be reasonable if the host data base is a file that can be read line by line. On many systems, a routine of this name reads from the file */etc/hosts*. It may be available only when the library was built without DNS support. The glibc version will ignore ipv6 entries. This function is not reentrant, and glibc adds a reentrant version **gethostent_r**().

GNU extensions

glibc2 also has a **gethostbyname2**() that works like **gethostbyname**(), but permits to specify the address family to which the address must belong.

glibc2 also has reentrant versions **gethostent_r**(), **gethostbyaddr_r**(), **gethostbyname_r**(), and **gethostbyname2_r**(). The caller supplies a *hostent* structure *ret* which will be filled in on success, and a temporary work buffer *buf* of size *buflen*. After the call, *result* will point to the result on success. In case of an error or if no entry is found *result* will be NULL. The functions return 0 on success and a nonzero error number on failure. In addition to the errors returned by the nonreentrant versions of these functions, if *buf* is too small, the functions will return **ERANGE**, and the call should be retried with a larger buffer. The global variable *h_errno* is not modified, but the address of a variable in which to store error numbers is passed in *h_errnop*.

BUGS

gethostbyname() does not recognize components of a dotted IPv4 address string that are expressed in hexadecimal.

SEE ALSO

getaddrinfo(3), *getnameinfo*(3), *inet*(3), *inet_ntop*(3), *inet_pton*(3), *resolver*(3), *hosts*(5), *nsswitch.conf*(5), *hostname*(7), *named*(8)

NAME

gethostid, sethostid − get or set the unique identifier of the current host

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

long gethostid(void);
int sethostid(long hostid);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gethostid():
Since glibc 2.20:
_DEFAULT_SOURCE || _XOPEN_SOURCE >= 500
Up to and including glibc 2.19:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

sethostid():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

gethostid() and **sethostid**() respectively get or set a unique 32-bit identifier for the current machine. The 32-bit identifier was intended to be unique among all UNIX systems in existence. This normally resembles the Internet address for the local machine, as returned by *gethostbyname*(3), and thus usually never needs to be set.

The **sethostid**() call is restricted to the superuser.

RETURN VALUE

gethostid() returns the 32-bit identifier for the current host as set by **sethostid**().

On success, **sethostid**() returns 0; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

sethostid() can fail with the following errors:

EACCES
The caller did not have permission to write to the file used to store the host ID.

EPERM
The calling process's effective user or group ID is not the same as its corresponding real ID.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gethostid()	Thread safety	MT-Safe hostid env locale
sethostid()	Thread safety	MT-Unsafe const:hostid

VERSIONS

In the glibc implementation, the *hostid* is stored in the file */etc/hostid*. (Before glibc 2.2, the file */var/adm/hostid* was used.)

In the glibc implementation, if **gethostid**() cannot open the file containing the host ID, then it obtains the hostname using *gethostname*(2), passes that hostname to *gethostbyname_r*(3) in order to obtain the host's IPv4 address, and returns a value obtained by bit-twiddling the IPv4 address. (This value may not be unique.)

STANDARDS

gethostid()
POSIX.1-2008.

sethostid()
None.

HISTORY

4.2BSD; dropped in 4.4BSD. SVr4 and POSIX.1-2001 include **gethostid**() but not **sethostid**().

BUGS

It is impossible to ensure that the identifier is globally unique.

SEE ALSO

hostid(1), *gethostbyname*(3)

NAME

getifaddrs, freeifaddrs − get interface addresses

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <ifaddrs.h>

int getifaddrs(struct ifaddrs *****ifap*);**
void freeifaddrs(struct ifaddrs ****ifa*);**

DESCRIPTION

The **getifaddrs**() function creates a linked list of structures describing the network interfaces of the local system, and stores the address of the first item of the list in **ifap*. The list consists of *ifaddrs* structures, defined as follows:

```
struct ifaddrs {
    struct ifaddrs  *ifa_next;    /* Next item in list */
    char            *ifa_name;    /* Name of interface */
    unsigned int     ifa_flags;   /* Flags from SIOCGIFFLAGS */
    struct sockaddr *ifa_addr;    /* Address of interface */
    struct sockaddr *ifa_netmask; /* Netmask of interface */
    union {
        struct sockaddr *ifu_broadaddr;
                         /* Broadcast address of interface */
        struct sockaddr *ifu_dstaddr;
                         /* Point-to-point destination address */
    } ifa_ifu;
#define              ifa_broadaddr ifa_ifu.ifu_broadaddr
#define              ifa_dstaddr   ifa_ifu.ifu_dstaddr
    void            *ifa_data;    /* Address-specific data */
};
```

The *ifa_next* field contains a pointer to the next structure on the list, or NULL if this is the last item of the list.

The *ifa_name* points to the null-terminated interface name.

The *ifa_flags* field contains the interface flags, as returned by the **SIOCGIFFLAGS** *ioctl*(2) operation (see *netdevice*(7) for a list of these flags).

The *ifa_addr* field points to a structure containing the interface address. (The *sa_family* subfield should be consulted to determine the format of the address structure.) This field may contain a null pointer.

The *ifa_netmask* field points to a structure containing the netmask associated with *ifa_addr*, if applicable for the address family. This field may contain a null pointer.

Depending on whether the bit **IFF_BROADCAST** or **IFF_POINTOPOINT** is set in *ifa_flags* (only one can be set at a time), either *ifa_broadaddr* will contain the broadcast address associated with *ifa_addr* (if applicable for the address family) or *ifa_dstaddr* will contain the destination address of the point-to-point interface.

The *ifa_data* field points to a buffer containing address-family-specific data; this field may be NULL if there is no such data for this interface.

The data returned by **getifaddrs**() is dynamically allocated and should be freed using **freeifaddrs**() when no longer needed.

RETURN VALUE

On success, **getifaddrs**() returns zero; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

getifaddrs() may fail and set *errno* for any of the errors specified for *socket*(2), *bind*(2), *getsockname*(2), *recvmsg*(2), *sendto*(2), *malloc*(3), or *realloc*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getifaddrs(), **freeifaddrs**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

This function first appeared in BSDi and is present on the BSD systems, but with slightly different semantics documented—returning one entry per interface, not per address. This means *ifa_addr* and other fields can actually be NULL if the interface has no address, and no link-level address is returned if the interface has an IP address assigned. Also, the way of choosing either *ifa_broadaddr* or *ifa_dstaddr* differs on various systems.

getifaddrs() first appeared in glibc 2.3, but before glibc 2.3.3, the implementation supported only IPv4 addresses; IPv6 support was added in glibc 2.3.3. Support of address families other than IPv4 is available only on kernels that support netlink.

NOTES

The addresses returned on Linux will usually be the IPv4 and IPv6 addresses assigned to the interface, but also one **AF_PACKET** address per interface containing lower-level details about the interface and its physical layer. In this case, the *ifa_data* field may contain a pointer to a *struct rtnl_link_stats*, defined in <*linux/if_link.h*> (in Linux 2.4 and earlier, *struct net_device_stats*, defined in <*linux/netdevice.h*>), which contains various interface attributes and statistics.

EXAMPLES

The program below demonstrates the use of **getifaddrs**(), **freeifaddrs**(), and *getnameinfo*(3). Here is what we see when running this program on one system:

```
$ ./a.out
lo       AF_PACKET (17)
                tx_packets =        524; rx_packets =        524
                tx_bytes   =      38788; rx_bytes   =      38788
wlp3s0   AF_PACKET (17)
                tx_packets =     108391; rx_packets =     130245
                tx_bytes   =   30420659; rx_bytes   =   94230014
em1      AF_PACKET (17)
                tx_packets =          0; rx_packets =          0
                tx_bytes   =          0; rx_bytes   =          0
lo       AF_INET (2)
                address: <127.0.0.1>
wlp3s0   AF_INET (2)
                address: <192.168.235.137>
lo       AF_INET6 (10)
                address: <::1>
wlp3s0   AF_INET6 (10)
                address: <fe80::7ee9:d3ff:fef5:1a91%wlp3s0>
```

Program source

```
#define _GNU_SOURCE     /* To get defns of NI_MAXSERV and NI_MAXHOST */
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <ifaddrs.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/if_link.h>

int main(int argc, char *argv[])
{
```

```
        struct ifaddrs *ifaddr;
        int family, s;
        char host[NI_MAXHOST];

        if (getifaddrs(&ifaddr) == -1) {
            perror("getifaddrs");
            exit(EXIT_FAILURE);
        }

        /* Walk through linked list, maintaining head pointer so we
           can free list later. */

        for (struct ifaddrs *ifa = ifaddr; ifa != NULL;
                 ifa = ifa->ifa_next) {
            if (ifa->ifa_addr == NULL)
                continue;

            family = ifa->ifa_addr->sa_family;

            /* Display interface name and family (including symbolic
               form of the latter for the common families). */

            printf("%-8s %s (%d)\n",
                   ifa->ifa_name,
                   (family == AF_PACKET) ? "AF_PACKET" :
                   (family == AF_INET) ? "AF_INET" :
                   (family == AF_INET6) ? "AF_INET6" : "???",
                   family);

            /* For an AF_INET* interface address, display the address. */

            if (family == AF_INET || family == AF_INET6) {
                s = getnameinfo(ifa->ifa_addr,
                        (family == AF_INET) ? sizeof(struct sockaddr_in) :
                                              sizeof(struct sockaddr_in6),
                        host, NI_MAXHOST,
                        NULL, 0, NI_NUMERICHOST);
                if (s != 0) {
                    printf("getnameinfo() failed: %s\n", gai_strerror(s));
                    exit(EXIT_FAILURE);
                }

                printf("\t\taddress: <%s>\n", host);

            } else if (family == AF_PACKET && ifa->ifa_data != NULL) {
                struct rtnl_link_stats *stats = ifa->ifa_data;

                printf("\t\ttx_packets = %10u; rx_packets = %10u\n"
                       "\t\ttx_bytes   = %10u; rx_bytes   = %10u\n",
                       stats->tx_packets, stats->rx_packets,
                       stats->tx_bytes, stats->rx_bytes);
            }
        }

        freeifaddrs(ifaddr);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

bind(2), *getsockname*(2), *socket*(2), *packet*(7), *ifconfig*(8)

NAME

getipnodebyname, getipnodebyaddr, freehostent − get network hostnames and addresses

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

[[deprecated]] struct hostent *getipnodebyname(const char *name, int af,
                        int flags, int *error_num);
[[deprecated]] struct hostent *getipnodebyaddr(const void addr[.len],
                        size_t len, int af,
                        int *error_num);
[[deprecated]] void freehostent(struct hostent *ip);
```

DESCRIPTION

These functions are deprecated (and unavailable in glibc). Use *getaddrinfo*(3) and *getnameinfo*(3) instead.

The **getipnodebyname**() and **getipnodebyaddr**() functions return the names and addresses of a network host. These functions return a pointer to the following structure:

```
struct hostent {
    char  *h_name;
    char **h_aliases;
    int    h_addrtype;
    int    h_length;
    char **h_addr_list;
};
```

These functions replace the *gethostbyname*(3) and *gethostbyaddr*(3) functions, which could access only the IPv4 network address family. The **getipnodebyname**() and **getipnodebyaddr**() functions can access multiple network address families.

Unlike the **gethostby** functions, these functions return pointers to dynamically allocated memory. The **freehostent**() function is used to release the dynamically allocated memory after the caller no longer needs the *hostent* structure.

getipnodebyname() arguments

The **getipnodebyname**() function looks up network addresses for the host specified by the *name* argument. The *af* argument specifies one of the following values:

AF_INET
: The *name* argument points to a dotted-quad IPv4 address or a name of an IPv4 network host.

AF_INET6
: The *name* argument points to a hexadecimal IPv6 address or a name of an IPv6 network host.

The *flags* argument specifies additional options. More than one option can be specified by bitwise ORing them together. *flags* should be set to 0 if no options are desired.

AI_V4MAPPED
: This flag is used with **AF_INET6** to request a query for IPv4 addresses instead of IPv6 addresses; the IPv4 addresses will be mapped to IPv6 addresses.

AI_ALL
: This flag is used with **AI_V4MAPPED** to request a query for both IPv4 and IPv6 addresses. Any IPv4 address found will be mapped to an IPv6 address.

AI_ADDRCONFIG
: This flag is used with **AF_INET6** to further request that queries for IPv6 addresses should not be made unless the system has at least one IPv6 address assigned to a network interface, and that queries for IPv4 addresses should not be made unless the system has at least one IPv4 address assigned to a network interface. This flag may be used by itself or with the **AI_V4MAPPED** flag.

AI_DEFAULT
This flag is equivalent to **(AI_ADDRCONFIG | AI_V4MAPPED)**.

getipnodebyaddr() arguments
The **getipnodebyaddr**() function looks up the name of the host whose network address is specified by the *addr* argument. The *af* argument specifies one of the following values:

AF_INET
The *addr* argument points to a *struct in_addr* and *len* must be set to *sizeof(struct in_addr)*.

AF_INET6
The *addr* argument points to a *struct in6_addr* and *len* must be set to *sizeof(struct in6_addr)*.

RETURN VALUE
NULL is returned if an error occurred, and *error_num* will contain an error code from the following list:

HOST_NOT_FOUND
The hostname or network address was not found.

NO_ADDRESS
The domain name server recognized the network address or name, but no answer was returned. This can happen if the network host has only IPv4 addresses and a request has been made for IPv6 information only, or vice versa.

NO_RECOVERY
The domain name server returned a permanent failure response.

TRY_AGAIN
The domain name server returned a temporary failure response. You might have better luck next time.

A successful query returns a pointer to a *hostent* structure that contains the following fields:

h_name
This is the official name of this network host.

h_aliases
This is an array of pointers to unofficial aliases for the same host. The array is terminated by a null pointer.

h_addrtype
This is a copy of the *af* argument to **getipnodebyname**() or **getipnodebyaddr**(). *h_addrtype* will always be **AF_INET** if the *af* argument was **AF_INET**. *h_addrtype* will always be **AF_INET6** if the *af* argument was **AF_INET6**.

h_length
This field will be set to *sizeof(struct in_addr)* if *h_addrtype* is **AF_INET**, and to *sizeof(struct in6_addr)* if *h_addrtype* is **AF_INET6**.

h_addr_list
This is an array of one or more pointers to network address structures for the network host. The array is terminated by a null pointer.

STANDARDS
None.

HISTORY
RFC 2553.

Present in glibc 2.1.91-95, but removed again. Several UNIX-like systems support them, but all call them deprecated.

SEE ALSO
getaddrinfo(3), *getnameinfo*(3), *inet_ntop*(3), *inet_pton*(3)

NAME

getline, getdelim − delimited string input

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

ssize_t getline(char **restrict lineptr, size_t *restrict n,
                FILE *restrict stream);
ssize_t getdelim(char **restrict lineptr, size_t *restrict n,
                 int delim, FILE *restrict stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getline(), **getdelim**():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

getline() reads an entire line from *stream*, storing the address of the buffer containing the text into **lineptr*. The buffer is null-terminated and includes the newline character, if one was found.

If **lineptr* is set to NULL before the call, then **getline**() will allocate a buffer for storing the line. This buffer should be freed by the user program even if **getline**() failed.

Alternatively, before calling **getline**(), **lineptr* can contain a pointer to a *malloc*(3)-allocated buffer **n* bytes in size. If the buffer is not large enough to hold the line, **getline**() resizes it with *realloc*(3), updating **lineptr* and **n* as necessary.

In either case, on a successful call, **lineptr* and **n* will be updated to reflect the buffer address and allocated size respectively.

getdelim() works like **getline**(), except that a line delimiter other than newline can be specified as the *delimiter* argument. As with **getline**(), a delimiter character is not added if one was not present in the input before end of file was reached.

RETURN VALUE

On success, **getline**() and **getdelim**() return the number of characters read, including the delimiter character, but not including the terminating null byte ('\0'). This value can be used to handle embedded null bytes in the line read.

Both functions return −1 on failure to read a line (including end-of-file condition). In the event of a failure, *errno* is set to indicate the error.

If **lineptr* was set to NULL before the call, then the buffer should be freed by the user program even on failure.

ERRORS

EINVAL
Bad arguments (*n* or *lineptr* is NULL, or *stream* is not valid).

ENOMEM
Allocation or reallocation of the line buffer failed.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getline(), **getdelim**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

GNU, POSIX.1-2008.

EXAMPLES

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    FILE *stream;
    char *line = NULL;
    size_t len = 0;
    ssize_t nread;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    stream = fopen(argv[1], "r");
    if (stream == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    while ((nread = getline(&line, &len, stream)) != -1) {
        printf("Retrieved line of length %zd:\n", nread);
        fwrite(line, nread, 1, stdout);
    }

    free(line);
    fclose(stream);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

read(2), *fgets*(3), *fopen*(3), *fread*(3), *scanf*(3)

NAME

getloadavg − get system load averages

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int getloadavg(double *loadavg[]***, int** *nelem***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getloadavg():
Since glibc 2.19:
_DEFAULT_SOURCE
In glibc up to and including 2.19:
_BSD_SOURCE

DESCRIPTION

The **getloadavg**() function returns the number of processes in the system run queue averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg[]*. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively.

RETURN VALUE

If the load average was unobtainable, −1 is returned; otherwise, the number of samples actually retrieved is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getloadavg()	Thread safety	MT-Safe

STANDARDS

BSD.

HISTORY

4.3BSD-Reno, Solaris. glibc 2.2.

SEE ALSO

uptime(1), *proc*(5)

NAME

getlogin, getlogin_r, cuserid − get username

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

char *getlogin(void);
int getlogin_r(char *buf* **[.***bufsize***], size_t** *bufsize***);**

#include <stdio.h>

char *cuserid(char * *string***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getlogin_r():
_POSIX_C_SOURCE >= 199506L

cuserid():
Since glibc 2.24:
(_XOPEN_SOURCE && ! (_POSIX_C_SOURCE >= 200112L)
|| _GNU_SOURCE
Up to and including glibc 2.23:
_XOPEN_SOURCE

DESCRIPTION

getlogin() returns a pointer to a string containing the name of the user logged in on the controlling terminal of the process, or a null pointer if this information cannot be determined. The string is statically allocated and might be overwritten on subsequent calls to this function or to **cuserid**().

getlogin_r() returns this same username in the array *buf* of size *bufsize*.

cuserid() returns a pointer to a string containing a username associated with the effective user ID of the process. If *string* is not a null pointer, it should be an array that can hold at least **L_cuserid** characters; the string is returned in this array. Otherwise, a pointer to a string in a static area is returned. This string is statically allocated and might be overwritten on subsequent calls to this function or to **getlogin**().

The macro **L_cuserid** is an integer constant that indicates how long an array you might need to store a username. **L_cuserid** is declared in <*stdio.h*>.

These functions let your program identify positively the user who is running (**cuserid**()) or the user who logged in this session (**getlogin**()). (These can differ when set-user-ID programs are involved.)

For most purposes, it is more useful to use the environment variable **LOGNAME** to find out who the user is. This is more flexible precisely because the user can set **LOGNAME** arbitrarily.

RETURN VALUE

getlogin() returns a pointer to the username when successful, and NULL on failure, with *errno* set to indicate the error. **getlogin_r**() returns 0 when successful, and nonzero on failure.

ERRORS

POSIX specifies:

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENXIO
The calling process has no controlling terminal.

ERANGE
(getlogin_r) The length of the username, including the terminating null byte ('\0'), is larger than *bufsize*.

Linux/glibc also has:

ENOENT
There was no corresponding entry in the utmp-file.

ENOMEM
Insufficient memory to allocate passwd structure.

ENOTTY
Standard input didn't refer to a terminal. (See BUGS.)

FILES

/etc/passwd
password database file

/var/run/utmp
(traditionally */etc/utmp*; some libc versions used */var/adm/utmp*)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getlogin()	Thread safety	MT-Unsafe race:getlogin race:utent sig:ALRM timer locale
getlogin_r()	Thread safety	MT-Unsafe race:utent sig:ALRM timer locale
cuserid()	Thread safety	MT-Unsafe race:cuserid/!string locale

In the above table, *utent* in *race:utent* signifies that if any of the functions *setutent*(3), *getutent*(3), or *endutent*(3) are used in parallel in different threads of a program, then data races could occur. **getlogin**() and **getlogin_r**() call those functions, so we use race:utent to remind users.

VERSIONS

OpenBSD has **getlogin**() and **setlogin**(), and a username associated with a session, even if it has no controlling terminal.

STANDARDS

getlogin()
getlogin_r()
POSIX.1-2008.

cuserid()
None.

STANDARDS

getlogin()
getlogin_r():
POSIX.1-2001. OpenBSD.

cuserid()
System V, POSIX.1-1988. Removed in POSIX.1-1990. SUSv2. Removed in POSIX.1-2001.

System V has a **cuserid**() function which uses the real user ID rather than the effective user ID.

BUGS

Unfortunately, it is often rather easy to fool **getlogin**(). Sometimes it does not work at all, because some program messed up the utmp file. Often, it gives only the first 8 characters of the login name. The user currently logged in on the controlling terminal of our program need not be the user who started it. Avoid **getlogin**() for security-related purposes.

Note that glibc does not follow the POSIX specification and uses *stdin* instead of */dev/tty*. A bug. (Other recent systems, like SunOS 5.8 and HP-UX 11.11 and FreeBSD 4.8 all return the login name also when *stdin* is redirected.)

Nobody knows precisely what **cuserid**() does; avoid it in portable programs. Or avoid it altogether: use *getpwuid(geteuid())* instead, if that is what you meant. **Do not use cuserid**().

SEE ALSO

logname(1), *geteuid*(2), *getuid*(2), *utmp*(5)

NAME

getmntent, setmntent, addmntent, endmntent, hasmntopt, getmntent_r − get filesystem descriptor file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <mntent.h>

FILE *setmntent(const char *filename, const char *type);

struct mntent *getmntent(FILE *stream);

int addmntent(FILE *restrict stream,
              const struct mntent *restrict mnt);

int endmntent(FILE *streamp);

char *hasmntopt(const struct mntent *mnt, const char *opt);

/* GNU extension */
#include <mntent.h>

struct mntent *getmntent_r(FILE *restrict streamp,
              struct mntent *restrict mntbuf,
              char buf[restrict .buflen], int buflen);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getmntent_r():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These routines are used to access the filesystem description file */etc/fstab* and the mounted filesystem description file */etc/mtab*.

The **setmntent**() function opens the filesystem description file *filename* and returns a file pointer which can be used by **getmntent**(). The argument *type* is the type of access required and can take the same values as the *mode* argument of *fopen*(3). The returned stream should be closed using **endmntent**() rather than *fclose*(3).

The **getmntent**() function reads the next line of the filesystem description file from *stream* and returns a pointer to a structure containing the broken out fields from a line in the file. The pointer points to a static area of memory which is overwritten by subsequent calls to **getmntent**().

The **addmntent**() function adds the *mntent* structure *mnt* to the end of the open *stream*.

The **endmntent**() function closes the *stream* associated with the filesystem description file.

The **hasmntopt**() function scans the *mnt_opts* field (see below) of the *mntent* structure *mnt* for a substring that matches *opt*. See <*mntent.h*> and *mount*(8) for valid mount options.

The reentrant **getmntent_r**() function is similar to **getmntent**(), but stores the *mntent* structure in the provided **mntbuf*, and stores the strings pointed to by the entries in that structure in the provided array *buf* of size *buflen*.

The *mntent* structure is defined in <*mntent.h*> as follows:

```
struct mntent {
    char *mnt_fsname;   /* name of mounted filesystem */
    char *mnt_dir;      /* filesystem path prefix */
    char *mnt_type;     /* mount type (see mntent.h) */
    char *mnt_opts;     /* mount options (see mntent.h) */
    int   mnt_freq;     /* dump frequency in days */
    int   mnt_passno;   /* pass number on parallel fsck */
};
```

Since fields in the mtab and fstab files are separated by whitespace, octal escapes are used to represent the characters space (\040), tab (\011), newline (\012), and backslash (\\) in those files when they occur in one of the four strings in a *mntent* structure. The routines **addmntent**() and **getmntent**() will convert from string representation to escaped representation and back. When converting from escaped representation, the sequence \134 is also converted to a backslash.

RETURN VALUE

The **getmntent**() and **getmntent_r**() functions return a pointer to the *mntent* structure or NULL on failure.

The **addmntent**() function returns 0 on success and 1 on failure.

The **endmntent**() function always returns 1.

The **hasmntopt**() function returns the address of the substring if a match is found and NULL otherwise.

FILES

/etc/fstab
: filesystem description file

/etc/mtab
: mounted filesystem description file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setmntent(), **endmntent**(), **hasmntopt**()	Thread safety	MT-Safe
getmntent()	Thread safety	MT-Unsafe race:mntentbuf locale
addmntent()	Thread safety	MT-Safe race:stream locale
getmntent_r()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

The nonreentrant functions are from SunOS 4.1.3. A routine **getmntent_r**() was introduced in HP-UX 10, but it returns an *int*. The prototype shown above is glibc-only.

System V also has a **getmntent**() function but the calling sequence differs, and the returned structure is different. Under System V */etc/mnttab* is used. 4.4BSD and Digital UNIX have a routine **getmntinfo**(), a wrapper around the system call **getfsstat**().

SEE ALSO

fopen(3), *fstab*(5), *mount*(8)

NAME

getnameinfo − address-to-name translation in protocol-independent manner

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr, socklen_t addrlen,
                char host[_Nullable restrict .hostlen],
                socklen_t hostlen,
                char serv[_Nullable restrict .servlen],
                socklen_t servlen,
                int  flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getnameinfo():
Since glibc 2.22:
_POSIX_C_SOURCE >= 200112L
glibc 2.21 and earlier:
_POSIX_C_SOURCE

DESCRIPTION

The **getnameinfo**() function is the inverse of *getaddrinfo*(3): it converts a socket address to a corresponding host and service, in a protocol-independent manner. It combines the functionality of *gethostbyaddr*(3) and *getservbyport*(3), but unlike those functions, **getnameinfo**() is reentrant and allows programs to eliminate IPv4-versus-IPv6 dependencies.

The *addr* argument is a pointer to a generic socket address structure (of type *sockaddr_in* or *sockaddr_in6*) of size *addrlen* that holds the input IP address and port number. The arguments *host* and *serv* are pointers to caller-allocated buffers (of size *hostlen* and *servlen* respectively) into which **getnameinfo**() places null-terminated strings containing the host and service names respectively.

The caller can specify that no hostname (or no service name) is required by providing a NULL *host* (or *serv*) argument or a zero *hostlen* (or *servlen*) argument. However, at least one of hostname or service name must be requested.

The *flags* argument modifies the behavior of **getnameinfo**() as follows:

NI_NAMEREQD
If set, then an error is returned if the hostname cannot be determined.

NI_DGRAM
If set, then the service is datagram (UDP) based rather than stream (TCP) based. This is required for the few ports (512–514) that have different services for UDP and TCP.

NI_NOFQDN
If set, return only the hostname part of the fully qualified domain name for local hosts.

NI_NUMERICHOST
If set, then the numeric form of the hostname is returned. (When not set, this will still happen in case the node's name cannot be determined.)

NI_NUMERICSERV
If set, then the numeric form of the service address is returned. (When not set, this will still happen in case the service's name cannot be determined.)

Extensions to getnameinfo() for Internationalized Domain Names

Starting with glibc 2.3.4, **getnameinfo**() has been extended to selectively allow hostnames to be transparently converted to and from the Internationalized Domain Name (IDN) format (see RFC 3490, *Internationalizing Domain Names in Applications (IDNA)*). Three new flags are defined:

NI_IDN
If this flag is used, then the name found in the lookup process is converted from IDN format to the locale's encoding if necessary. ASCII-only names are not affected by the conversion,

which makes this flag usable in existing programs and environments.

NI_IDN_ALLOW_UNASSIGNED
NI_IDN_USE_STD3_ASCII_RULES
Setting these flags will enable the IDNA_ALLOW_UNASSIGNED (allow unassigned Unicode code points) and IDNA_USE_STD3_ASCII_RULES (check output to make sure it is a STD3 conforming hostname) flags respectively to be used in the IDNA handling.

RETURN VALUE

On success, 0 is returned, and node and service names, if requested, are filled with null-terminated strings, possibly truncated to fit the specified buffer lengths. On error, one of the following nonzero error codes is returned:

EAI_AGAIN
The name could not be resolved at this time. Try again later.

EAI_BADFLAGS
The *flags* argument has an invalid value.

EAI_FAIL
A nonrecoverable error occurred.

EAI_FAMILY
The address family was not recognized, or the address length was invalid for the specified family.

EAI_MEMORY
Out of memory.

EAI_NONAME
The name does not resolve for the supplied arguments. **NI_NAMEREQD** is set and the host's name cannot be located, or neither hostname nor service name were requested.

EAI_OVERFLOW
The buffer pointed to by *host* or *serv* was too small.

EAI_SYSTEM
A system error occurred. The error code can be found in *errno*.

The *gai_strerror*(3) function translates these error codes to a human readable string, suitable for error reporting.

FILES

/etc/hosts
/etc/nsswitch.conf
/etc/resolv.conf

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getnameinfo()	Thread safety	MT-Safe env locale

STANDARDS

POSIX.1-2008. RFC 2553.

HISTORY

glibc 2.1. POSIX.1-2001.

Before glibc 2.2, the *hostlen* and *servlen* arguments were typed as *size_t*.

NOTES

In order to assist the programmer in choosing reasonable sizes for the supplied buffers, <*netdb.h*> defines the constants

```
#define NI_MAXHOST      1025
#define NI_MAXSERV      32
```

Since glibc 2.8, these definitions are exposed only if suitable feature test macros are defined, namely: **_GNU_SOURCE**, **_DEFAULT_SOURCE** (since glibc 2.19), or (in glibc versions up to and including 2.19) **_BSD_SOURCE** or **_SVID_SOURCE**.

The former is the constant **MAXDNAME** in recent versions of BIND's <*arpa/nameser.h*> header file. The latter is a guess based on the services listed in the current Assigned Numbers RFC.

EXAMPLES

The following code tries to get the numeric hostname and service name, for a given socket address. Note that there is no hardcoded reference to a particular address family.

```
struct sockaddr *addr;     /* input */
socklen_t addrlen;         /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(addr, addrlen, hbuf, sizeof(hbuf), sbuf,
            sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    printf("host=%s, serv=%s\n", hbuf, sbuf);
```

The following version checks if the socket address has a reverse address mapping.

```
struct sockaddr *addr;     /* input */
socklen_t addrlen;         /* input */
char hbuf[NI_MAXHOST];

if (getnameinfo(addr, addrlen, hbuf, sizeof(hbuf),
            NULL, 0, NI_NAMEREQD))
    printf("could not resolve hostname");
else
    printf("host=%s\n", hbuf);
```

An example program using **getnameinfo**() can be found in *getaddrinfo*(3).

SEE ALSO

accept(2), *getpeername*(2), *getsockname*(2), *recvfrom*(2), *socket*(2), *getaddrinfo*(3), *gethostbyaddr*(3), *getservbyname*(3), *getservbyport*(3), *inet_ntop*(3), *hosts*(5), *services*(5), *hostname*(7), *named*(8)

R. Gilligan, S. Thomson, J. Bound and W. Stevens, *Basic Socket Interface Extensions for IPv6*, RFC 2553, March 1999.

Tatsuya Jinmei and Atsushi Onoe, *An Extension of Format for IPv6 Scoped Addresses*, internet draft, work in progress .

Craig Metz, *Protocol Independence Using the Sockets API*, Proceedings of the freenix track: 2000 USENIX annual technical conference, June 2000 .

NAME

getnetent, getnetbyname, getnetbyaddr, setnetent, endnetent − get network entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <netdb.h>

struct netent *getnetent(void);

struct netent *getnetbyname(const char *_name_);
struct netent *getnetbyaddr(uint32_t _net_, int _type_);

void setnetent(int _stayopen_);
void endnetent(void);

DESCRIPTION

The **getnetent**() function reads the next entry from the networks database and returns a *netent* structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

The **getnetbyname**() function returns a *netent* structure for the entry from the database that matches the network *name*.

The **getnetbyaddr**() function returns a *netent* structure for the entry from the database that matches the network number *net* of type *type*. The *net* argument must be in host byte order.

The **setnetent**() function opens a connection to the database, and sets the next entry to the first entry. If *stayopen* is nonzero, then the connection to the database will not be closed between calls to one of the **getnet***() functions.

The **endnetent**() function closes the connection to the database.

The *netent* structure is defined in <*netdb.h*> as follows:

```
struct netent {
    char      *n_name;     /* official network name */
    char     **n_aliases;  /* alias list */
    int        n_addrtype; /* net address type */
    uint32_t   n_net;      /* network number */
}
```

The members of the *netent* structure are:

n_name
: The official name of the network.

n_aliases
: A NULL-terminated list of alternative names for the network.

n_addrtype
: The type of the network number; always **AF_INET**.

n_net
: The network number in host byte order.

RETURN VALUE

The **getnetent**(), **getnetbyname**(), and **getnetbyaddr**() functions return a pointer to a statically allocated *netent* structure, or a null pointer if an error occurs or the end of the file is reached.

FILES

/etc/networks
: networks database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getnetent()	Thread safety	MT-Unsafe race:netent race:netentbuf env locale
getnetbyname()	Thread safety	MT-Unsafe race:netbyname env locale
getnetbyaddr()	Thread safety	MT-Unsafe race:netbyaddr locale
setnetent(), **endnetent**()	Thread safety	MT-Unsafe race:netent env locale

In the above table, *netent* in *race:netent* signifies that if any of the functions **setnetent**(), **getnetent**(), or **endnetent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

Before glibc 2.2, the *net* argument of **getnetbyaddr**() was of type *long*.

SEE ALSO

getnetent_r(3), *getprotoent*(3), *getservent*(3)
RFC 1101

NAME

getnetent_r, getnetbyname_r, getnetbyaddr_r − get network entry (reentrant)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

int getnetent_r(struct netent *restrict result_buf,
                char buf[restrict .buflen], size_t buflen,
                struct netent **restrict result,
                int *restrict h_errnop);
int getnetbyname_r(const char *restrict name,
                struct netent *restrict result_buf,
                char buf[restrict .buflen], size_t buflen,
                struct netent **restrict result,
                int *restrict h_errnop);
int getnetbyaddr_r(uint32_t net, int type,
                struct netent *restrict result_buf,
                char buf[restrict .buflen], size_t buflen,
                struct netent **restrict result,
                int *restrict h_errnop);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getnetent_r(), **getnetbyname_r**(), **getnetbyaddr_r**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getnetent_r**(), **getnetbyname_r**(), and **getnetbyaddr_r**() functions are the reentrant equivalents of, respectively, *getnetent*(3), *getnetbyname*(3), and *getnetbynumber*(3)They differ in the way that the *netent* structure is returned, and in the function calling signature and return value. This manual page describes just the differences from the nonreentrant functions.

Instead of returning a pointer to a statically allocated *netent* structure as the function result, these functions copy the structure into the location pointed to by *result_buf*.

The *buf* array is used to store the string fields pointed to by the returned *netent* structure. (The nonreentrant functions allocate these strings in static storage.) The size of this array is specified in *buflen*. If *buf* is too small, the call fails with the error **ERANGE**, and the caller must try again with a larger buffer. (A buffer of length 1024 bytes should be sufficient for most applications.)

If the function call successfully obtains a network record, then **result* is set pointing to *result_buf*; otherwise, **result* is set to NULL.

The buffer pointed to by *h_errnop* is used to return the value that would be stored in the global variable *h_errno* by the nonreentrant versions of these functions.

RETURN VALUE

On success, these functions return 0. On error, they return one of the positive error numbers listed in ERRORS.

On error, record not found (**getnetbyname_r**(), *getnetbyaddr_r*()), or end of input (**getnetent_r**()) *result* is set to NULL.

ERRORS

ENOENT
(**getnetent_r**()) No more records in database.

ERANGE
buf is too small. Try again with a larger buffer (and increased *buflen*).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getnetent_r(), **getnetbyname_r**(), **getnetbyaddr_r**()	Thread safety	MT-Safe locale

VERSIONS

Functions with similar names exist on some other systems, though typically with different calling signatures.

STANDARDS

GNU.

SEE ALSO

getnetent(3), *networks*(5)

NAME

getopt, getopt_long, getopt_long_only, optarg, optind, opterr, optopt − Parse command-line options

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int getopt(int argc, char *argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

#include <getopt.h>

int getopt_long(int argc, char *argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
int getopt_long_only(int argc, char *argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getopt():
_POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE

getopt_long(), **getopt_long_only**():
_GNU_SOURCE

DESCRIPTION

The **getopt**() function parses the command-line arguments. Its arguments *argc* and *argv* are the argument count and array as passed to the *main*() function on program invocation. An element of *argv* that starts with '−' (and is not exactly "−" or "−−") is an option element. The characters of this element (aside from the initial '−') are option characters. If **getopt**() is called repeatedly, it returns successively each of the option characters from each of the option elements.

The variable *optind* is the index of the next element to be processed in *argv*. The system initializes this value to 1. The caller can reset it to 1 to restart scanning of the same *argv*, or when scanning a new argument vector.

If **getopt**() finds another option character, it returns that character, updating the external variable *optind* and a static variable *nextchar* so that the next call to **getopt**() can resume the scan with the following option character or *argv*-element.

If there are no more option characters, **getopt**() returns −1. Then *optind* is the index in *argv* of the first *argv*-element that is not an option.

optstring is a string containing the legitimate option characters. A legitimate option character is any visible one byte *ascii*(7) character (for which *isgraph*(3) would return nonzero) that is not '−', ':', or ';'. If such a character is followed by a colon, the option requires an argument, so **getopt**() places a pointer to the following text in the same *argv*-element, or the text of the following *argv*-element, in *optarg*. Two colons mean an option takes an optional arg; if there is text in the current *argv*-element (i.e., in the same word as the option name itself, for example, "−oarg"), then it is returned in *optarg*, otherwise *optarg* is set to zero. This is a GNU extension. If *optstring* contains **W** followed by a semicolon, then **−W foo** is treated as the long option **--foo**. (The **−W** option is reserved by POSIX.2 for implementation extensions.) This behavior is a GNU extension, not available with libraries before glibc 2.

By default, **getopt**() permutes the contents of *argv* as it scans, so that eventually all the nonoptions are at the end. Two other scanning modes are also implemented. If the first character of *optstring* is '+' or the environment variable **POSIXLY_CORRECT** is set, then option processing stops as soon as a nonoption argument is encountered. If '+' is not the first character of *optstring*, it is treated as a normal option. If **POSIXLY_CORRECT** behaviour is required in this case *optstring* will contain two '+' symbols. If the first character of *optstring* is '−', then each nonoption *argv*-element is handled as if it were the argument of an option with character code 1. (This is used by programs that were written to expect options and other *argv*-elements in any order and that care about the ordering of the two.) The

special argument "−−" forces an end of option-scanning regardless of the scanning mode.

While processing the option list, **getopt**() can detect two kinds of errors: (1) an option character that was not specified in *optstring* and (2) a missing option argument (i.e., an option at the end of the command line without an expected argument). Such errors are handled and reported as follows:

- By default, **getopt**() prints an error message on standard error, places the erroneous option character in *optopt*, and returns '?' as the function result.
- If the caller has set the global variable *opterr* to zero, then **getopt**() does not print an error message. The caller can determine that there was an error by testing whether the function return value is '?'. (By default, *opterr* has a nonzero value.)
- If the first character (following any optional '+' or '−' described above) of *optstring* is a colon (':'), then **getopt**() likewise does not print an error message. In addition, it returns ':' instead of '?' to indicate a missing option argument. This allows the caller to distinguish the two different types of errors.

getopt_long() and getopt_long_only()

The **getopt_long**() function works like **getopt**() except that it also accepts long options, started with two dashes. (If the program accepts only long options, then *optstring* should be specified as an empty string (""), not NULL.) Long option names may be abbreviated if the abbreviation is unique or is an exact match for some defined option. A long option may take a parameter, of the form **−−arg=param** or **--arg param**.

longopts is a pointer to the first element of an array of *struct option* declared in <*getopt.h*> as

```
struct option {
    const char *name;
    int         has_arg;
    int        *flag;
    int         val;
};
```

The meanings of the different fields are:

name is the name of the long option.

has_arg
: is: **no_argument** (or 0) if the option does not take an argument; **required_argument** (or 1) if the option requires an argument; or **optional_argument** (or 2) if the option takes an optional argument.

flag specifies how results are returned for a long option. If *flag* is NULL, then **getopt_long**() returns *val*. (For example, the calling program may set *val* to the equivalent short option character.) Otherwise, **getopt_long**() returns 0, and *flag* points to a variable which is set to *val* if the option is found, but left unchanged if the option is not found.

val is the value to return, or to load into the variable pointed to by *flag*.

The last element of the array has to be filled with zeros.

If *longindex* is not NULL, it points to a variable which is set to the index of the long option relative to *longopts*.

getopt_long_only() is like **getopt_long**(), but '−' as well as "−−" can indicate a long option. If an option that starts with '−' (not "−−") doesn't match a long option, but does match a short option, it is parsed as a short option instead.

RETURN VALUE

If an option was successfully found, then **getopt**() returns the option character. If all command-line options have been parsed, then **getopt**() returns −1. If **getopt**() encounters an option character that was not in *optstring*, then '?' is returned. If **getopt**() encounters an option with a missing argument, then the return value depends on the first character in *optstring*: if it is ':', then ':' is returned; otherwise '?' is returned.

getopt_long() and **getopt_long_only**() also return the option character when a short option is recognized. For a long option, they return *val* if *flag* is NULL, and 0 otherwise. Error and −1 returns are the same as for **getopt**(), plus '?' for an ambiguous match or an extraneous parameter.

ENVIRONMENT

POSIXLY_CORRECT
If this is set, then option processing stops as soon as a nonoption argument is encountered.

_<PID>_GNU_nonoption_argv_flags_
This variable was used by *bash*(1) 2.0 to communicate to glibc which arguments are the results of wildcard expansion and so should not be considered as options. This behavior was removed in *bash*(1) 2.01, but the support remains in glibc.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getopt(), **getopt_long**(), **getopt_long_only**()	Thread safety	MT-Unsafe race:getopt env

VERSIONS

POSIX specifies that the *argv* array argument should be *const*, but these functions permute its elements unless the environment variable **POSIXLY_CORRECT** is set. *const* is used in the actual prototype to be compatible with other systems; however, this page doesn't show the qualifier, to avoid confusing readers.

STANDARDS

getopt()
POSIX.1-2008.

getopt_long()
getopt_long_only()
GNU.

The use of '+' and '−' in *optstring* is a GNU extension.

HISTORY

getopt()
POSIX.1-2001, and POSIX.2.

On some older implementations, **getopt**() was declared in <*stdio.h*>. SUSv1 permitted the declaration to appear in either <*unistd.h*> or <*stdio.h*>. POSIX.1-1996 marked the use of <*stdio.h*> for this purpose as LEGACY. POSIX.1-2001 does not require the declaration to appear in <*stdio.h*>.

NOTES

A program that scans multiple argument vectors, or rescans the same vector more than once, and wants to make use of GNU extensions such as '+' and '−' at the start of *optstring*, or changes the value of **POSIXLY_CORRECT** between scans, must reinitialize **getopt**() by resetting *optind* to 0, rather than the traditional value of 1. (Resetting to 0 forces the invocation of an internal initialization routine that rechecks **POSIXLY_CORRECT** and checks for GNU extensions in *optstring*.)

Command-line arguments are parsed in strict order meaning that an option requiring an argument will consume the next argument, regardless of whether that argument is the correctly specified option argument or simply the next option (in the scenario the user mis-specifies the command line). For example, if *optstring* is specified as "1n:" and the user specifies the command line arguments incorrectly as *prog −n −1*, the *−n* option will be given the **optarg** value "−1", and the *−1* option will be considered to have not been specified.

EXAMPLES

getopt()

The following trivial example program uses **getopt**() to handle two program options: *−n*, with no associated value; and *−t val*, which expects an associated value.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int flags, opt;
```

```
        int nsecs, tfnd;

        nsecs = 0;
        tfnd = 0;
        flags = 0;
        while ((opt = getopt(argc, argv, "nt:")) != -1) {
            switch (opt) {
            case 'n':
                flags = 1;
                break;
            case 't':
                nsecs = atoi(optarg);
                tfnd = 1;
                break;
            default: /* '?' */
                fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                        argv[0]);
                exit(EXIT_FAILURE);
            }
        }

        printf("flags=%d; tfnd=%d; nsecs=%d; optind=%d\n",
               flags, tfnd, nsecs, optind);

        if (optind >= argc) {
            fprintf(stderr, "Expected argument after options\n");
            exit(EXIT_FAILURE);
        }

        printf("name argument = %s\n", argv[optind]);

        /* Other code omitted */

        exit(EXIT_SUCCESS);
    }
```

getopt_long()

The following example program illustrates the use of **getopt_long**() with most of its features.

```
    #include <getopt.h>
    #include <stdio.h>     /* for printf */
    #include <stdlib.h>    /* for exit */

    int
    main(int argc, char *argv[])
    {
        int c;
        int digit_optind = 0;

        while (1) {
            int this_option_optind = optind ? optind : 1;
            int option_index = 0;
            static struct option long_options[] = {
                {"add",     required_argument, 0,  0 },
                {"append",  no_argument,       0,  0 },
                {"delete",  required_argument, 0,  0 },
                {"verbose", no_argument,       0,  0 },
                {"create",  required_argument, 0, 'c'},
                {"file",    required_argument, 0,  0 },
                {0,         0,                 0,  0 }
```

```
            };

            c = getopt_long(argc, argv, "abc:d:012",
                            long_options, &option_index);
            if (c == -1)
                break;

            switch (c) {
            case 0:
                printf("option %s", long_options[option_index].name);
                if (optarg)
                    printf(" with arg %s", optarg);
                printf("\n");
                break;

            case '0':
            case '1':
            case '2':
                if (digit_optind != 0 && digit_optind != this_option_optind)
                  printf("digits occur in two different argv-elements.\n");
                digit_optind = this_option_optind;
                printf("option %c\n", c);
                break;

            case 'a':
                printf("option a\n");
                break;

            case 'b':
                printf("option b\n");
                break;

            case 'c':
                printf("option c with value '%s'\n", optarg);
                break;

            case 'd':
                printf("option d with value '%s'\n", optarg);
                break;

            case '?':
                break;

            default:
                printf("?? getopt returned character code 0%o ??\n", c);
            }
        }

        if (optind < argc) {
            printf("non-option ARGV-elements: ");
            while (optind < argc)
                printf("%s ", argv[optind++]);
            printf("\n");
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getopt(1), *getsubopt*(3)

NAME

getpass − get a password

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

[[deprecated]] char *getpass(const char * *prompt***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpass():
Since glibc 2.2.2:
_XOPEN_SOURCE && ! (_POSIX_C_SOURCE >= 200112L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE
Before glibc 2.2.2:
none

DESCRIPTION

This function is obsolete. Do not use it. See NOTES. If you want to read input without terminal echoing enabled, see the description of the *ECHO* flag in *termios*(3).

The **getpass**() function opens */dev/tty* (the controlling terminal of the process), outputs the string *prompt*, turns off echoing, reads one line (the "password"), restores the terminal state and closes */dev/tty* again.

RETURN VALUE

The function **getpass**() returns a pointer to a static buffer containing (the first **PASS_MAX** bytes of) the password without the trailing newline, terminated by a null byte ('\0'). This buffer may be overwritten by a following call. On error, the terminal state is restored, *errno* is set to indicate the error, and NULL is returned.

ERRORS

ENXIO
The process does not have a controlling terminal.

FILES

/dev/tty

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpass()	Thread safety	MT-Unsafe term

STANDARDS

None.

HISTORY

Version 7 AT&T UNIX. Present in SUSv2, but marked LEGACY. Removed in POSIX.1-2001.

NOTES

You should use instead *readpassphrase*(3bsd), provided by *libbsd*.

In the GNU C library implementation, if */dev/tty* cannot be opened, the prompt is written to *stderr* and the password is read from *stdin*. There is no limit on the length of the password. Line editing is not disabled.

According to SUSv2, the value of **PASS_MAX** must be defined in <*limits.h*> in case it is smaller than 8, and can in any case be obtained using *sysconf(_SC_PASS_MAX)*. However, POSIX.2 withdraws the constants **PASS_MAX** and **_SC_PASS_MAX**, and the function **getpass**(). The glibc version accepts **_SC_PASS_MAX** and returns **BUFSIZ** (e.g., 8192).

BUGS

The calling process should zero the password as soon as possible to avoid leaving the cleartext password visible in the process's address space.

SEE ALSO

crypt(3)

NAME

getprotoent, getprotobyname, getprotobynumber, setprotoent, endprotoent − get protocol entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent(void);

struct protoent *getprotobyname(const char *_name_);
struct protoent *getprotobynumber(int *proto***);**

void setprotoent(int *stayopen***);**
void endprotoent(void);

DESCRIPTION

The **getprotoent**() function reads the next entry from the protocols database (see *protocols*(5)) and returns a *protoent* structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

The **getprotobyname**() function returns a *protoent* structure for the entry from the database that matches the protocol name *name*. A connection is opened to the database if necessary.

The **getprotobynumber**() function returns a *protoent* structure for the entry from the database that matches the protocol number *number*. A connection is opened to the database if necessary.

The **setprotoent**() function opens a connection to the database, and sets the next entry to the first entry. If *stayopen* is nonzero, then the connection to the database will not be closed between calls to one of the **getproto***() functions.

The **endprotoent**() function closes the connection to the database.

The *protoent* structure is defined in <*netdb.h*> as follows:

```
struct protoent {
    char  *p_name;       /* official protocol name */
    char **p_aliases;    /* alias list */
    int    p_proto;      /* protocol number */
}
```

The members of the *protoent* structure are:

p_name
: The official name of the protocol.

p_aliases
: A NULL-terminated list of alternative names for the protocol.

p_proto
: The protocol number.

RETURN VALUE

The **getprotoent**(), **getprotobyname**(), and **getprotobynumber**() functions return a pointer to a statically allocated *protoent* structure, or a null pointer if an error occurs or the end of the file is reached.

FILES

/etc/protocols
: protocol database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getprotoent()	Thread safety	MT-Unsafe race:protoent race:protoentbuf locale
getprotobyname()	Thread safety	MT-Unsafe race:protobyname locale
getprotobynumber()	Thread safety	MT-Unsafe race:protobynumber locale
setprotoent(), **endprotoent**()	Thread safety	MT-Unsafe race:protoent locale

In the above table, *protoent* in *race:protoent* signifies that if any of the functions **setprotoent**(), **getprotoent**(), or **endprotoent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

SEE ALSO

getnetent(3), *getprotoent_r*(3), *getservent*(3), *protocols*(5)

NAME

getprotoent_r, getprotobyname_r, getprotobynumber_r − get protocol entry (reentrant)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

int getprotoent_r(struct protoent *restrict result_buf,
                  char buf[restrict .buflen], size_t buflen,
                  struct protoent **restrict result);
int getprotobyname_r(const char *restrict name,
                  struct protoent *restrict result_buf,
                  char buf[restrict .buflen], size_t buflen,
                  struct protoent **restrict result);
int getprotobynumber_r(int proto,
                  struct protoent *restrict result_buf,
                  char buf[restrict .buflen], size_t buflen,
                  struct protoent **restrict result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getprotoent_r(), **getprotobyname_r**(), **getprotobynumber_r**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getprotoent_r**(), **getprotobyname_r**(), and **getprotobynumber_r**() functions are the reentrant equivalents of, respectively, *getprotoent*(3), *getprotobyname*(3), and *getprotobynumber*(3). They differ in the way that the *protoent* structure is returned, and in the function calling signature and return value. This manual page describes just the differences from the nonreentrant functions.

Instead of returning a pointer to a statically allocated *protoent* structure as the function result, these functions copy the structure into the location pointed to by *result_buf*.

The *buf* array is used to store the string fields pointed to by the returned *protoent* structure. (The nonreentrant functions allocate these strings in static storage.) The size of this array is specified in *buflen*. If *buf* is too small, the call fails with the error **ERANGE**, and the caller must try again with a larger buffer. (A buffer of length 1024 bytes should be sufficient for most applications.)

If the function call successfully obtains a protocol record, then **result* is set pointing to *result_buf*; otherwise, **result* is set to NULL.

RETURN VALUE

On success, these functions return 0. On error, they return one of the positive error numbers listed in ERRORS.

On error, record not found (**getprotobyname_r**(), *getprotobynumber_r*()), or end of input (**getprotoent_r**()) *result* is set to NULL.

ERRORS

ENOENT
(**getprotoent_r**()) No more records in database.

ERANGE
buf is too small. Try again with a larger buffer (and increased *buflen*).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getprotoent_r(), **getprotobyname_r**(), **getprotobynumber_r**()	Thread safety	MT-Safe locale

VERSIONS

Functions with similar names exist on some other systems, though typically with different calling signatures.

STANDARDS

GNU.

EXAMPLES

The program below uses **getprotobyname_r**() to retrieve the protocol record for the protocol named in its first command-line argument. If a second (integer) command-line argument is supplied, it is used as the initial value for *buflen*; if **getprotobyname_r**() fails with the error **ERANGE**, the program retries with larger buffer sizes. The following shell session shows a couple of sample runs:

```
$ ./a.out tcp 1
ERANGE! Retrying with larger buffer
getprotobyname_r() returned: 0 (success)  (buflen=78)
p_name=tcp; p_proto=6; aliases=TCP
$ ./a.out xxx 1
ERANGE! Retrying with larger buffer
getprotobyname_r() returned: 0 (success)  (buflen=100)
Call failed/record not found
```

Program source

```
#define _GNU_SOURCE
#include <ctype.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BUF 10000

int
main(int argc, char *argv[])
{
    int buflen, erange_cnt, s;
    struct protoent result_buf;
    struct protoent *result;
    char buf[MAX_BUF];

    if (argc < 2) {
        printf("Usage: %s proto-name [buflen]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    buflen = 1024;
    if (argc > 2)
        buflen = atoi(argv[2]);

    if (buflen > MAX_BUF) {
        printf("Exceeded buffer limit (%d)\n", MAX_BUF);
        exit(EXIT_FAILURE);
    }

    erange_cnt = 0;
    do {
        s = getprotobyname_r(argv[1], &result_buf,
                             buf, buflen, &result);
        if (s == ERANGE) {
```

```
                if (erange_cnt == 0)
                    printf("ERANGE! Retrying with larger buffer\n");
                erange_cnt++;

                /* Increment a byte at a time so we can see exactly
                   what size buffer was required. */

                buflen++;

                if (buflen > MAX_BUF) {
                    printf("Exceeded buffer limit (%d)\n", MAX_BUF);
                    exit(EXIT_FAILURE);
                }
            }
        } while (s == ERANGE);

        printf("getprotobyname_r() returned: %s  (buflen=%d)\n",
               (s == 0) ? "0 (success)" : (s == ENOENT) ? "ENOENT" :
               strerror(s), buflen);

        if (s != 0 || result == NULL) {
            printf("Call failed/record not found\n");
            exit(EXIT_FAILURE);
        }

        printf("p_name=%s; p_proto=%d; aliases=",
               result_buf.p_name, result_buf.p_proto);
        for (char **p = result_buf.p_aliases; *p != NULL; p++)
            printf("%s ", *p);
        printf("\n");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getprotoent(3), *protocols*(5)

NAME

getpt − open a new pseudoterminal master

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <stdlib.h>

int getpt(void);
```

DESCRIPTION

getpt() opens a new pseudoterminal device and returns a file descriptor that refers to that device. It is equivalent to opening the pseudoterminal multiplexor device

```
open("/dev/ptmx", O_RDWR);
```

on Linux systems, though the pseudoterminal multiplexor device is located elsewhere on some systems that use the GNU C library.

RETURN VALUE

getpt() returns an open file descriptor upon successful completion. Otherwise, it returns −1 and sets *errno* to indicate the error.

ERRORS

getpt() can fail with various errors described in *open*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpt()	Thread safety	MT-Safe

VERSIONS

Use *posix_openpt*(3) instead.

STANDARDS

GNU.

HISTORY

glibc 2.1.

SEE ALSO

grantpt(3), *posix_openpt*(3), *ptsname*(3), *unlockpt*(3), *ptmx*(4), *pty*(7)

NAME

getpw − reconstruct password line entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/types.h>
#include <pwd.h>

[[deprecated]] int getpw(uid_t uid, char *buf);
```

DESCRIPTION

The **getpw**() function reconstructs the password line entry for the given user ID *uid* in the buffer *buf*. The returned buffer contains a line of format

```
name:passwd:uid:gid:gecos:dir:shell
```

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char   *pw_name;       /* username */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* user information */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

For more information about the fields of this structure, see *passwd*(5).

RETURN VALUE

The **getpw**() function returns 0 on success; on error, it returns −1, and *errno* is set to indicate the error.

If *uid* is not found in the password database, **getpw**() returns −1, sets *errno* to 0, and leaves *buf* unchanged.

ERRORS

0 or **ENOENT**
: No user corresponding to *uid*.

EINVAL
: *buf* is NULL.

ENOMEM
: Insufficient memory to allocate *passwd* structure.

FILES

/etc/passwd
: password database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpw()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

SVr2.

BUGS

The **getpw**() function is dangerous as it may overflow the provided buffer *buf*. It is obsoleted by *getpwuid*(3).

SEE ALSO

endpwent(3), *fgetpwent*(3), *getpwent*(3), *getpwnam*(3), *getpwuid*(3), *putpwent*(3), *setpwent*(3), *passwd*(5)

NAME

getpwent, setpwent, endpwent − get password file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpwent(), **setpwent**(), **endpwent**():
_XOPEN_SOURCE >= 500
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getpwent**() function returns a pointer to a structure containing the broken-out fields of a record from the password database (e.g., the local password file */etc/passwd*, NIS, and LDAP). The first time **getpwent**() is called, it returns the first entry; thereafter, it returns successive entries.

The **setpwent**() function rewinds to the beginning of the password database.

The **endpwent**() function is used to close the password database after all processing has been performed.

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char   *pw_name;       /* username */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* user information */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

For more information about the fields of this structure, see *passwd*(5).

RETURN VALUE

The **getpwent**() function returns a pointer to a *passwd* structure, or NULL if there are no more entries or an error occurred. If an error occurs, *errno* is set to indicate the error. If one wants to check *errno* after the call, it should be set to zero before the call.

The return value may point to a static area, and may be overwritten by subsequent calls to **getpwent**(), *getpwnam*(3), or *getpwuid*(3). (Do not pass the returned pointer to *free*(3).)

ERRORS

EINTR
A signal was caught; see *signal*(7).

EIO I/O error.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
Insufficient memory to allocate *passwd* structure.

ERANGE
Insufficient buffer space supplied.

FILES

/etc/passwd
local password database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpwent()	Thread safety	MT-Unsafe race:pwent race:pwentbuf locale
setpwent(), **endpwent**()	Thread safety	MT-Unsafe race:pwent locale

In the above table, *pwent* in *race:pwent* signifies that if any of the functions **setpwent**(), **getpwent**(), or **endpwent**() are used in parallel in different threads of a program, then data races could occur.

VERSIONS

The *pw_gecos* field is not specified in POSIX, but is present on most implementations.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

fgetpwent(3), *getpw*(3), *getpwent_r*(3), *getpwnam*(3), *getpwuid*(3), *putpwent*(3), *passwd*(5)

NAME

getpwent_r, fgetpwent_r − get passwd file entry reentrantly

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <pwd.h>

int getpwent_r(struct passwd *restrict pwbuf,
               char buf[restrict .buflen], size_t buflen,
               struct passwd **restrict pwbufp);
int fgetpwent_r(FILE *restrict stream, struct passwd *restrict pwbuf,
               char buf[restrict .buflen], size_t buflen,
               struct passwd **restrict pwbufp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpwent_r(),
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

fgetpwent_r():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

The functions **getpwent_r**() and **fgetpwent_r**() are the reentrant versions of *getpwent*(3) and *fgetpwent*(3). The former reads the next passwd entry from the stream initialized by *setpwent*(3). The latter reads the next passwd entry from *stream*.

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char    *pw_name;       /* username */
    char    *pw_passwd;     /* user password */
    uid_t    pw_uid;        /* user ID */
    gid_t    pw_gid;        /* group ID */
    char    *pw_gecos;      /* user information */
    char    *pw_dir;        /* home directory */
    char    *pw_shell;      /* shell program */
};
```

For more information about the fields of this structure, see *passwd*(5).

The nonreentrant functions return a pointer to static storage, where this static storage contains further pointers to user name, password, gecos field, home directory and shell. The reentrant functions described here return all of that in caller-provided buffers. First of all there is the buffer *pwbuf* that can hold a *struct passwd*. And next the buffer *buf* of size *buflen* that can hold additional strings. The result of these functions, the *struct passwd* read from the stream, is stored in the provided buffer **pwbuf*, and a pointer to this *struct passwd* is returned in **pwbufp*.

RETURN VALUE

On success, these functions return 0 and **pwbufp* is a pointer to the *struct passwd*. On error, these functions return an error value and **pwbufp* is NULL.

ERRORS

ENOENT
No more entries.

ERANGE
Insufficient buffer space supplied. Try again with larger buffer.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpwent_r()	Thread safety	MT-Unsafe race:pwent locale
fgetpwent_r()	Thread safety	MT-Safe

In the above table, *pwent* in *race:pwent* signifies that if any of the functions **setpwent**(), **getpwent**(), **endpwent**(), or **getpwent_r**() are used in parallel in different threads of a program, then data races could occur.

VERSIONS

Other systems use the prototype

```
struct passwd *
getpwent_r(struct passwd *pwd, char *buf, int buflen);
```

or, better,

```
int
getpwent_r(struct passwd *pwd, char *buf, int buflen,
           FILE **pw_fp);
```

STANDARDS

None.

HISTORY

These functions are done in a style resembling the POSIX version of functions like *getpwnam_r*(3).

NOTES

The function **getpwent_r**() is not really reentrant since it shares the reading position in the stream with all other threads.

EXAMPLES

```
#define _GNU_SOURCE
#include <pwd.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFLEN 4096

int
main(void)
{
    struct passwd pw;
    struct passwd *pwp;
    char buf[BUFLEN];
    int i;

    setpwent();
    while (1) {
        i = getpwent_r(&pw, buf, sizeof(buf), &pwp);
        if (i)
            break;
        printf("%s (%jd)\tHOME %s\tSHELL %s\n", pwp->pw_name,
               (intmax_t) pwp->pw_uid, pwp->pw_dir, pwp->pw_shell);
    }
    endpwent();
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fgetpwent(3), *getpw*(3), *getpwent*(3), *getpwnam*(3), *getpwuid*(3), *putpwent*(3), *passwd*(5)

NAME

getpwnam, getpwnam_r, getpwuid, getpwuid_r − get password file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);

int getpwnam_r(const char *restrict name, struct passwd *restrict pwd,
        char buf[restrict .buflen], size_t buflen,
        struct passwd **restrict result);
int getpwuid_r(uid_t uid, struct passwd *restrict pwd,
        char buf[restrict .buflen], size_t buflen,
        struct passwd **restrict result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getpwnam_r(), **getpwuid_r**():
_POSIX_C_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getpwnam**() function returns a pointer to a structure containing the broken-out fields of the record in the password database (e.g., the local password file */etc/passwd*, NIS, and LDAP) that matches the username *name*.

The **getpwuid**() function returns a pointer to a structure containing the broken-out fields of the record in the password database that matches the user ID *uid*.

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char   *pw_name;       /* username */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* user information */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

See *passwd*(5) for more information about these fields.

The **getpwnam_r**() and **getpwuid_r**() functions obtain the same information as **getpwnam**() and **getpwuid**(), but store the retrieved *passwd* structure in the space pointed to by *pwd*. The string fields pointed to by the members of the *passwd* structure are stored in the buffer *buf* of size *buflen*. A pointer to the result (in case of success) or NULL (in case no entry was found or an error occurred) is stored in **result*.

The call

```
sysconf(_SC_GETPW_R_SIZE_MAX)
```

returns either −1, without changing *errno*, or an initial suggested size for *buf*. (If this size is too small, the call fails with **ERANGE**, in which case the caller can retry with a larger buffer.)

RETURN VALUE

The **getpwnam**() and **getpwuid**() functions return a pointer to a *passwd* structure, or NULL if the matching entry is not found or an error occurs. If an error occurs, *errno* is set to indicate the error. If one wants to check *errno* after the call, it should be set to zero before the call.

The return value may point to a static area, and may be overwritten by subsequent calls to *getpwent*(3), **getpwnam**(), or **getpwuid**(). (Do not pass the returned pointer to *free*(3).)

On success, **getpwnam_r**() and **getpwuid_r**() return zero, and set **result* to *pwd*. If no matching password record was found, these functions return 0 and store NULL in **result*. In case of error, an error number is returned, and NULL is stored in **result*.

ERRORS

0 or **ENOENT** or **ESRCH** or **EBADF** or **EPERM** or ...
The given *name* or *uid* was not found.

EINTR
A signal was caught; see *signal*(7).

EIO I/O error.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
Insufficient memory to allocate *passwd* structure.

ERANGE
Insufficient buffer space supplied.

FILES

/etc/passwd
local password database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getpwnam()	Thread safety	MT-Unsafe race:pwnam locale
getpwuid()	Thread safety	MT-Unsafe race:pwuid locale
getpwnam_r(), **getpwuid_r**()	Thread safety	MT-Safe locale

VERSIONS

The *pw_gecos* field is not specified in POSIX, but is present on most implementations.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

The formulation given above under "RETURN VALUE" is from POSIX.1-2001. It does not call "not found" an error, and hence does not specify what value *errno* might have in this situation. But that makes it impossible to recognize errors. One might argue that according to POSIX *errno* should be left unchanged if an entry is not found. Experiments on various UNIX-like systems show that lots of different values occur in this situation: 0, ENOENT, EBADF, ESRCH, EWOULDBLOCK, EPERM, and probably others.

The *pw_dir* field contains the name of the initial working directory of the user. Login programs use the value of this field to initialize the **HOME** environment variable for the login shell. An application that wants to determine its user's home directory should inspect the value of **HOME** (rather than the value *getpwuid(getuid())−>pw_dir*) since this allows the user to modify their notion of "the home directory" during a login session. To determine the (initial) home directory of another user, it is necessary to use *getpwnam("username")−>pw_dir* or similar.

EXAMPLES

The program below demonstrates the use of **getpwnam_r**() to find the full username and user ID for the username supplied as a command-line argument.

```
#include <errno.h>
#include <pwd.h>
#include <stdint.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    struct passwd pwd;
    struct passwd *result;
    char *buf;
    long bufsize;
    int s;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s username\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    if (bufsize == -1)          /* Value was indeterminate */
        bufsize = 16384;        /* Should be more than enough */

    buf = malloc(bufsize);
    if (buf == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    s = getpwnam_r(argv[1], &pwd, buf, bufsize, &result);
    if (result == NULL) {
        if (s == 0)
            printf("Not found\n");
        else {
            errno = s;
            perror("getpwnam_r");
        }
        exit(EXIT_FAILURE);
    }

    printf("Name: %s; UID: %jd\n", pwd.pw_gecos,
           (intmax_t) pwd.pw_uid);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

endpwent(3), *fgetpwent*(3), *getgrnam*(3), *getpw*(3), *getpwent*(3), *getspnam*(3), *putpwent*(3), *setpwent*(3), *passwd*(5)

NAME

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, endrpcent − get RPC entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent(void);

struct rpcent *getrpcbyname(const char *name);
struct rpcent *getrpcbynumber(int number);

void setrpcent(int stayopen);
void endrpcent(void);
```

DESCRIPTION

The **getrpcent**(), **getrpcbyname**(), and **getrpcbynumber**() functions each return a pointer to an object with the following structure containing the broken-out fields of an entry in the RPC program number data base.

```
struct rpcent {
    char  *r_name;     /* name of server for this RPC program */
    char **r_aliases;  /* alias list */
    long   r_number;   /* RPC program number */
};
```

The members of this structure are:

r_name The name of the server for this RPC program.

r_aliases
: A NULL-terminated list of alternate names for the RPC program.

r_number
: The RPC program number for this service.

The **getrpcent**() function reads the next entry from the database. A connection is opened to the database if necessary.

The **setrpcent**() function opens a connection to the database, and sets the next entry to the first entry. If *stayopen* is nonzero, then the connection to the database will not be closed between calls to one of the **getrpc***() functions.

The **endrpcent**() function closes the connection to the database.

The **getrpcbyname**() and **getrpcbynumber**() functions sequentially search from the beginning of the file until a matching RPC program name or program number is found, or until end-of-file is encountered.

RETURN VALUE

On success, **getrpcent**(), **getrpcbyname**(), and **getrpcbynumber**() return a pointer to a statically allocated *rpcent* structure. NULL is returned on EOF or error.

FILES

/etc/rpc
: RPC program number database.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getrpcent(), **getrpcbyname**(), **getrpcbynumber**()	Thread safety	MT-Unsafe
setrpcent(), **endrpcent**()	Thread safety	MT-Safe locale

STANDARDS

BSD.

HISTORY

BSD, Solaris.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

SEE ALSO

getrpcent_r(3), *rpc*(5), *rpcinfo*(8), *ypserv*(8)

NAME

getrpcent_r, getrpcbyname_r, getrpcbynumber_r − get RPC entry (reentrant)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

int getrpcent_r(struct rpcent *result_buf, char buf[.buflen],
                size_t buflen, struct rpcent **result);
int getrpcbyname_r(const char *name,
                struct rpcent *result_buf, char buf[.buflen],
                size_t buflen, struct rpcent **result);
int getrpcbynumber_r(int number,
                struct rpcent *result_buf, char buf[.buflen],
                size_t buflen, struct rpcent **result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getrpcent_r(), **getrpcbyname_r**(), **getrpcbynumber_r**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getrpcent_r**(), **getrpcbyname_r**(), and **getrpcbynumber_r**() functions are the reentrant equivalents of, respectively, *getrpcent*(3), *getrpcbyname*(3), and *getrpcbynumber*(3). They differ in the way that the *rpcent* structure is returned, and in the function calling signature and return value. This manual page describes just the differences from the nonreentrant functions.

Instead of returning a pointer to a statically allocated *rpcent* structure as the function result, these functions copy the structure into the location pointed to by *result_buf*.

The *buf* array is used to store the string fields pointed to by the returned *rpcent* structure. (The nonreentrant functions allocate these strings in static storage.) The size of this array is specified in *buflen*. If *buf* is too small, the call fails with the error **ERANGE**, and the caller must try again with a larger buffer. (A buffer of length 1024 bytes should be sufficient for most applications.)

If the function call successfully obtains an RPC record, then **result* is set pointing to *result_buf*; otherwise, **result* is set to NULL.

RETURN VALUE

On success, these functions return 0. On error, they return one of the positive error numbers listed in ERRORS.

On error, record not found (**getrpcbyname_r**(), *getrpcbynumber_r*()), or end of input (**getrpcent_r**()) *result* is set to NULL.

ERRORS

ENOENT
(**getrpcent_r**()) No more records in database.

ERANGE
buf is too small. Try again with a larger buffer (and increased *buflen*).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getrpcent_r(), **getrpcbyname_r**(), **getrpcbynumber_r**()	Thread safety	MT-Safe locale

VERSIONS

Functions with similar names exist on some other systems, though typically with different calling signatures.

STANDARDS

GNU.

SEE ALSO

getrpcent(3), *rpc*(5)

NAME

getrpcport − get RPC port number

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <rpc/rpc.h>

int getrpcport(const char ****host*, unsigned long** *prognum***,**
unsigned long *versnum***, unsigned int** *proto***);**

DESCRIPTION

getrpcport() returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getrpcport()	Thread safety	MT-Safe env locale

STANDARDS

BSD.

HISTORY

BSD, Solaris.

NAME

gets − get a string from standard input (DEPRECATED)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

[[deprecated]] char *gets(char *s);

DESCRIPTION

Never use this function.

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

gets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gets()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

LSB deprecates **gets**(). POSIX.1-2008 marks **gets**() obsolescent. ISO C11 removes the specification of **gets**() from the C language, and since glibc 2.16, glibc header files don't expose the function declaration if the **_ISOC11_SOURCE** feature test macro is defined.

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

For more information, see CWE-242 (aka "Use of Inherently Dangerous Function") at http://cwe.mitre.org/data/definitions/242.html

SEE ALSO

read(2), *write*(2), *ferror*(3), *fgetc*(3), *fgets*(3), *fgetwc*(3), *fgetws*(3), *fopen*(3), *fread*(3), *fseek*(3), *getline*(3), *getwchar*(3), *puts*(3), *scanf*(3), *ungetwc*(3), *unlocked_stdio*(3), *feature_test_macros*(7)

NAME

getservent, getservbyname, getservbyport, setservent, endservent − get service entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(const char * *name***, const char *** *proto***);**
struct servent *getservbyport(int *port***, const char *** *proto***);**

void setservent(int *stayopen***);**
void endservent(void);

DESCRIPTION

The **getservent**() function reads the next entry from the services database (see *services*(5)) and returns a *servent* structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

The **getservbyname**() function returns a *servent* structure for the entry from the database that matches the service *name* using protocol *proto*. If *proto* is NULL, any protocol will be matched. A connection is opened to the database if necessary.

The **getservbyport**() function returns a *servent* structure for the entry from the database that matches the port *port* (given in network byte order) using protocol *proto*. If *proto* is NULL, any protocol will be matched. A connection is opened to the database if necessary.

The **setservent**() function opens a connection to the database, and sets the next entry to the first entry. If *stayopen* is nonzero, then the connection to the database will not be closed between calls to one of the **getserv***() functions.

The **endservent**() function closes the connection to the database.

The *servent* structure is defined in <*netdb.h*> as follows:

```
struct servent {
    char  *s_name;       /* official service name */
    char **s_aliases;    /* alias list */
    int    s_port;       /* port number */
    char  *s_proto;      /* protocol to use */
}
```

The members of the *servent* structure are:

s_name
: The official name of the service.

s_aliases
: A NULL-terminated list of alternative names for the service.

s_port The port number for the service given in network byte order.

s_proto The name of the protocol to use with this service.

RETURN VALUE

The **getservent**(), **getservbyname**(), and **getservbyport**() functions return a pointer to a statically allocated *servent* structure, or NULL if an error occurs or the end of the file is reached.

FILES

/etc/services
: services database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getservent()	Thread safety	MT-Unsafe race:servent race:serventbuf locale
getservbyname()	Thread safety	MT-Unsafe race:servbyname locale
getservbyport()	Thread safety	MT-Unsafe race:servbyport locale
setservent(), **endservent**()	Thread safety	MT-Unsafe race:servent locale

In the above table, *servent* in *race:servent* signifies that if any of the functions **setservent**(), **getservent**(), or **endservent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

SEE ALSO

getnetent(3), *getprotoent*(3), *getservent_r*(3), *services*(5)

NAME

getservent_r, getservbyname_r, getservbyport_r − get service entry (reentrant)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

int getservent_r(struct servent *restrict result_buf,
                 char buf[restrict .buflen], size_t buflen,
                 struct servent **restrict result);
int getservbyname_r(const char *restrict name,
                 const char *restrict proto,
                 struct servent *restrict result_buf,
                 char buf[restrict .buflen], size_t buflen,
                 struct servent **restrict result);
int getservbyport_r(int port,
                 const char *restrict proto,
                 struct servent *restrict result_buf,
                 char buf[restrict .buflen], size_t buflen,
                 struct servent **restrict result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getservent_r(), **getservbyname_r**(), **getservbyport_r**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **getservent_r**(), **getservbyname_r**(), and **getservbyport_r**() functions are the reentrant equivalents of, respectively, *getservent*(3), *getservbyname*(3), and *getservbyport*(3). They differ in the way that the *servent* structure is returned, and in the function calling signature and return value. This manual page describes just the differences from the nonreentrant functions.

Instead of returning a pointer to a statically allocated *servent* structure as the function result, these functions copy the structure into the location pointed to by *result_buf*.

The *buf* array is used to store the string fields pointed to by the returned *servent* structure. (The nonreentrant functions allocate these strings in static storage.) The size of this array is specified in *buflen*. If *buf* is too small, the call fails with the error **ERANGE**, and the caller must try again with a larger buffer. (A buffer of length 1024 bytes should be sufficient for most applications.)

If the function call successfully obtains a service record, then **result* is set pointing to *result_buf*; otherwise, **result* is set to NULL.

RETURN VALUE

On success, these functions return 0. On error, they return one of the positive error numbers listed in errors.

On error, record not found (**getservbyname_r**(), *getservbyport_r*()), or end of input (**getservent_r**()) *result* is set to NULL.

ERRORS

ENOENT
(**getservent_r**()) No more records in database.

ERANGE
buf is too small. Try again with a larger buffer (and increased *buflen*).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getservent_r(), **getservbyname_r**(), **getservbyport_r**()	Thread safety	MT-Safe locale

VERSIONS

Functions with similar names exist on some other systems, though typically with different calling signatures.

STANDARDS

GNU.

EXAMPLES

The program below uses **getservbyport_r**() to retrieve the service record for the port and protocol named in its first command-line argument. If a third (integer) command-line argument is supplied, it is used as the initial value for *buflen*; if **getservbyport_r**() fails with the error **ERANGE**, the program retries with larger buffer sizes. The following shell session shows a couple of sample runs:

```
$ ./a.out 7 tcp 1
ERANGE! Retrying with larger buffer
getservbyport_r() returned: 0 (success)  (buflen=87)
s_name=echo; s_proto=tcp; s_port=7; aliases=
$ ./a.out 77777 tcp
getservbyport_r() returned: 0 (success)  (buflen=1024)
Call failed/record not found
```

Program source

```
#define _GNU_SOURCE
#include <ctype.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BUF 10000

int
main(int argc, char *argv[])
{
    int buflen, erange_cnt, port, s;
    struct servent result_buf;
    struct servent *result;
    char buf[MAX_BUF];
    char *protop;

    if (argc < 3) {
        printf("Usage: %s port-num proto-name [buflen]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    port = htons(atoi(argv[1]));
    protop = (strcmp(argv[2], "null") == 0 ||
              strcmp(argv[2], "NULL") == 0) ?  NULL : argv[2];

    buflen = 1024;
    if (argc > 3)
        buflen = atoi(argv[3]);

    if (buflen > MAX_BUF) {
        printf("Exceeded buffer limit (%d)\n", MAX_BUF);
        exit(EXIT_FAILURE);
    }

    erange_cnt = 0;
```

```
        do {
            s = getservbyport_r(port, protop, &result_buf,
                                buf, buflen, &result);
            if (s == ERANGE) {
                if (erange_cnt == 0)
                    printf("ERANGE! Retrying with larger buffer\n");
                erange_cnt++;

                /* Increment a byte at a time so we can see exactly
                   what size buffer was required. */

                buflen++;

                if (buflen > MAX_BUF) {
                    printf("Exceeded buffer limit (%d)\n", MAX_BUF);
                    exit(EXIT_FAILURE);
                }
            }
        } while (s == ERANGE);

        printf("getservbyport_r() returned: %s  (buflen=%d)\n",
               (s == 0) ? "0 (success)" : (s == ENOENT) ? "ENOENT" :
               strerror(s), buflen);

        if (s != 0 || result == NULL) {
            printf("Call failed/record not found\n");
            exit(EXIT_FAILURE);
        }

        printf("s_name=%s; s_proto=%s; s_port=%d; aliases=",
               result_buf.s_name, result_buf.s_proto,
               ntohs(result_buf.s_port));
        for (char **p = result_buf.s_aliases; *p != NULL; p++)
            printf("%s ", *p);
        printf("\n");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getservent(3), *services*(5)

NAME

getspnam, getspnam_r, getspent, getspent_r, setspent, endspent, fgetspent, fgetspent_r, sgetspent, sgetspent_r, putspent, lckpwdf, ulckpwdf − get shadow password file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
/* General shadow password file API */
#include <shadow.h>

struct spwd *getspnam(const char *name);
struct spwd *getspent(void);

void setspent(void);
void endspent(void);

struct spwd *fgetspent(FILE *stream);
struct spwd *sgetspent(const char *s);

int putspent(const struct spwd *p, FILE *stream);

int lckpwdf(void);
int ulckpwdf(void);

/* GNU extension */
#include <shadow.h>

int getspent_r(struct spwd *spbuf,
        char buf[buflen], size_t buflen, struct spwd **spbufp);
int getspnam_r(const char *name, struct spwd *spbuf,
        char buf[buflen], size_t buflen, struct spwd **spbufp);

int fgetspent_r(FILE *stream, struct spwd *spbuf,
        char buf[buflen], size_t buflen, struct spwd **spbufp);
int sgetspent_r(const char *s, struct spwd *spbuf,
        char buf[buflen], size_t buflen, struct spwd **spbufp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getspent_r(), **getspnam_r**(), **fgetspent_r**(), **sgetspent_r**():
- Since glibc 2.19:
  - _DEFAULT_SOURCE
- glibc 2.19 and earlier:
  - _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Long ago it was considered safe to have encrypted passwords openly visible in the password file. When computers got faster and people got more security-conscious, this was no longer acceptable. Julianne Frances Haugh implemented the shadow password suite that keeps the encrypted passwords in the shadow password database (e.g., the local shadow password file */etc/shadow*, NIS, and LDAP), readable only by root.

The functions described below resemble those for the traditional password database (e.g., see *getpwnam*(3) and *getpwent*(3)).

The **getspnam**() function returns a pointer to a structure containing the broken-out fields of the record in the shadow password database that matches the username *name*.

The **getspent**() function returns a pointer to the next entry in the shadow password database. The position in the input stream is initialized by **setspent**(). When done reading, the program may call **endspent**() so that resources can be deallocated.

The **fgetspent**() function is similar to **getspent**() but uses the supplied stream instead of the one implicitly opened by **setspent**().

The **sgetspent**() function parses the supplied string *s* into a struct *spwd*.

The **putspent**() function writes the contents of the supplied struct *spwd *p* as a text line in the shadow password file format to *stream*. String entries with value NULL and numerical entries with value −1

are written as an empty string.

The **lckpwdf**() function is intended to protect against multiple simultaneous accesses of the shadow password database. It tries to acquire a lock, and returns 0 on success, or −1 on failure (lock not obtained within 15 seconds). The **ulckpwdf**() function releases the lock again. Note that there is no protection against direct access of the shadow password file. Only programs that use **lckpwdf**() will notice the lock.

These were the functions that formed the original shadow API. They are widely available.

Reentrant versions

Analogous to the reentrant functions for the password database, glibc also has reentrant functions for the shadow password database. The **getspnam_r**() function is like **getspnam**() but stores the retrieved shadow password structure in the space pointed to by *spbuf*. This shadow password structure contains pointers to strings, and these strings are stored in the buffer *buf* of size *buflen*. A pointer to the result (in case of success) or NULL (in case no entry was found or an error occurred) is stored in **spbufp*.

The functions **getspent_r**(), **fgetspent_r**(), and **sgetspent_r**() are similarly analogous to their nonreentrant counterparts.

Some non-glibc systems also have functions with these names, often with different prototypes.

Structure

The shadow password structure is defined in <*shadow.h*> as follows:

```
struct spwd {
    char *sp_namp;     /* Login name */
    char *sp_pwdp;     /* Encrypted password */
    long  sp_lstchg;   /* Date of last change
                          (measured in days since
                          1970-01-01 00:00:00 +0000 (UTC)) */
    long  sp_min;      /* Min # of days between changes */
    long  sp_max;      /* Max # of days between changes */
    long  sp_warn;     /* # of days before password expires
                          to warn user to change it */
    long  sp_inact;    /* # of days after password expires
                          until account is disabled */
    long  sp_expire;   /* Date when account expires
                          (measured in days since
                          1970-01-01 00:00:00 +0000 (UTC)) */
    unsigned long sp_flag;  /* Reserved */
};
```

RETURN VALUE

The functions that return a pointer return NULL if no more entries are available or if an error occurs during processing. The functions which have *int* as the return value return 0 for success and −1 for failure, with *errno* set to indicate the error.

For the nonreentrant functions, the return value may point to static area, and may be overwritten by subsequent calls to these functions.

The reentrant functions return zero on success. In case of error, an error number is returned.

ERRORS

EACCES
The caller does not have permission to access the shadow password file.

ERANGE
Supplied buffer is too small.

FILES

/etc/shadow
local shadow password database file

/etc/.pwd.lock
lock file

The include file <*paths.h*> defines the constant **_PATH_SHADOW** to the pathname of the shadow

password file.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getspnam()	Thread safety	MT-Unsafe race:getspnam locale
getspent()	Thread safety	MT-Unsafe race:getspent race:spentbuf locale
setspent(), **endspent**(), **getspent_r**()	Thread safety	MT-Unsafe race:getspent locale
fgetspent()	Thread safety	MT-Unsafe race:fgetspent
sgetspent()	Thread safety	MT-Unsafe race:sgetspent
putspent(), **getspnam_r**(), **sgetspent_r**()	Thread safety	MT-Safe locale
lckpwdf(), **ulckpwdf**(), **fgetspent_r**()	Thread safety	MT-Safe

In the above table, *getspent* in *race:getspent* signifies that if any of the functions **setspent**(), **getspent**(), **getspent_r**(), or **endspent**() are used in parallel in different threads of a program, then data races could occur.

VERSIONS

Many other systems provide a similar API.

STANDARDS

None.

SEE ALSO

getgrnam(3), *getpwnam*(3), *getpwnam_r*(3), *shadow*(5)

NAME

getsubopt − parse suboption arguments from a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int getsubopt(char **restrict optionp, char *const *restrict tokens,
        char **restrict valuep);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getsubopt():
  _XOPEN_SOURCE >= 500
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION

getsubopt() parses the list of comma-separated suboptions provided in *optionp*. (Such a suboption list is typically produced when *getopt*(3) is used to parse a command line; see for example the *−o* option of *mount*(8)Each suboption may include an associated value, which is separated from the suboption name by an equal sign. The following is an example of the kind of string that might be passed in *optionp*:

```
ro,name=xyz
```

The *tokens* argument is a pointer to a NULL-terminated array of pointers to the tokens that **getsubopt**() will look for in *optionp*. The tokens should be distinct, null-terminated strings containing at least one character, with no embedded equal signs or commas.

Each call to **getsubopt**() returns information about the next unprocessed suboption in *optionp*. The first equal sign in a suboption (if any) is interpreted as a separator between the name and the value of that suboption. The value extends to the next comma, or (for the last suboption) to the end of the string. If the name of the suboption matches a known name from *tokens*, and a value string was found, **getsubopt**() sets **valuep* to the address of that string. The first comma in *optionp* is overwritten with a null byte, so **valuep* is precisely the "value string" for that suboption.

If the suboption is recognized, but no value string was found, **valuep* is set to NULL.

When **getsubopt**() returns, *optionp* points to the next suboption, or to the null byte ('\0') at the end of the string if the last suboption was just processed.

RETURN VALUE

If the first suboption in *optionp* is recognized, **getsubopt**() returns the index of the matching suboption element in *tokens*. Otherwise, −1 is returned and **valuep* is the entire *name***[**=*value***]** string.

Since **optionp* is changed, the first suboption before the call to **getsubopt**() is not (necessarily) the same as the first suboption after **getsubopt**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getsubopt()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Since **getsubopt**() overwrites any commas it finds in the string **optionp*, that string must be writable; it cannot be a string constant.

EXAMPLES

The following program expects suboptions following a "−o" option.

```
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
```

```
#include <assert.h>

int
main(int argc, char *argv[])
{
    enum {
        RO_OPT = 0,
        RW_OPT,
        NAME_OPT
    };
    char *const token[] = {
        [RO_OPT]   = "ro",
        [RW_OPT]   = "rw",
        [NAME_OPT] = "name",
        NULL
    };
    char *subopts;
    char *value;
    int opt;

    int readonly = 0;
    int readwrite = 0;
    char *name = NULL;
    int errfnd = 0;

    while ((opt = getopt(argc, argv, "o:")) != -1) {
        switch (opt) {
        case 'o':
            subopts = optarg;
            while (*subopts != '\0' && !errfnd) {

                switch (getsubopt(&subopts, token, &value)) {
                case RO_OPT:
                    readonly = 1;
                    break;

                case RW_OPT:
                    readwrite = 1;
                    break;

                case NAME_OPT:
                    if (value == NULL) {
                        fprintf(stderr,
                                "Missing value for suboption '%s'\n",
                                token[NAME_OPT]);
                        errfnd = 1;
                        continue;
                    }

                    name = value;
                    break;

                default:
                    fprintf(stderr,
                            "No match found for token: /%s/\n", value);
                    errfnd = 1;
                    break;
                }
```

```
                }
                if (readwrite && readonly) {
                    fprintf(stderr,
                            "Only one of '%s' and '%s' can be specified\n",
                            token[RO_OPT], token[RW_OPT]);
                    errfnd = 1;
                }
                break;

            default:
                errfnd = 1;
            }
        }

        if (errfnd || argc == 1) {
            fprintf(stderr, "\nUsage: %s −o <suboptstring>\n", argv[0]);
            fprintf(stderr,
                    "suboptions are 'ro', 'rw', and 'name=<value>'\n");
            exit(EXIT_FAILURE);
        }

        /* Remainder of program... */

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getopt(3)

NAME

getttyent, getttynam, setttyent, endttyent − get ttys file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <ttyent.h>

struct ttyent *getttyent(void);
struct ttyent *getttynam(const char *_name_);

int setttyent(void);
int endttyent(void);

DESCRIPTION

These functions provide an interface to the file **_PATH_TTYS** (e.g., */etc/ttys*).

The function **setttyent**() opens the file or rewinds it if already open.

The function **endttyent**() closes the file.

The function **getttynam**() searches for a given terminal name in the file. It returns a pointer to a *ttyent* structure (description below).

The function **getttyent**() opens the file **_PATH_TTYS** (if necessary) and returns the first entry. If the file is already open, the next entry. The *ttyent* structure has the form:

```
struct ttyent {
    char *ty_name;     /* terminal device name */
    char *ty_getty;    /* command to execute, usually getty */
    char *ty_type;     /* terminal type for termcap */
    int   ty_status;   /* status flags */
    char *ty_window;   /* command to start up window manager */
    char *ty_comment;  /* comment field */
};
```

ty_status can be:

```
#define TTY_ON     0x01  /* enable logins (start ty_getty program) */
#define TTY_SECURE 0x02  /* allow UID 0 to login */
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getttyent(), **setttyent**(), **endttyent**(), **getttynam**()	Thread safety	MT-Unsafe race:ttyent

STANDARDS

BSD.

NOTES

Under Linux, the file */etc/ttys*, and the functions described above, are not used.

SEE ALSO

ttyname(3), *ttyslot*(3)

NAME

getusershell, setusershell, endusershell − get permitted user shells

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

char *getusershell(void);
void setusershell(void);
void endusershell(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getusershell(), **setusershell**(), **endusershell**():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

The **getusershell**() function returns the next line from the file */etc/shells*, opening the file if necessary. The line should contain the pathname of a valid user shell. If */etc/shells* does not exist or is unreadable, **getusershell**() behaves as if */bin/sh* and */bin/csh* were listed in the file.

The **setusershell**() function rewinds */etc/shells*.

The **endusershell**() function closes */etc/shells*.

RETURN VALUE

The **getusershell**() function returns NULL on end-of-file.

FILES

/etc/shells

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getusershell(), **setusershell**(), **endusershell**()	Thread safety	MT-Unsafe

STANDARDS

None.

HISTORY

4.3BSD.

SEE ALSO

shells(5)

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname − access utmp file entries

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent(void);
struct utmp *getutid(const struct utmp *ut);
struct utmp *getutline(const struct utmp *ut);

struct utmp *pututline(const struct utmp *ut);

void setutent(void);
void endutent(void);

int utmpname(const char * file);
```

DESCRIPTION

New applications should use the POSIX.1-specified "utmpx" versions of these functions; see STANDARDS.

utmpname() sets the name of the utmp-format file for the other utmp functions to access. If **utmpname**() is not used to set the filename before the other functions are used, they assume **_PATH_UTMP**, as defined in <*paths.h*>.

setutent() rewinds the file pointer to the beginning of the utmp file. It is generally a good idea to call it before any of the other functions.

endutent() closes the utmp file. It should be called when the user code is done accessing the file with the other functions.

getutent() reads a line from the current file position in the utmp file. It returns a pointer to a structure containing the fields of the line. The definition of this structure is shown in *utmp*(5).

getutid() searches forward from the current file position in the utmp file based upon *ut*. If *ut−>ut_type* is one of **RUN_LVL**, **BOOT_TIME**, **NEW_TIME**, or **OLD_TIME**, **getutid**() will find the first entry whose *ut_type* field matches *ut−>ut_type*. If *ut−>ut_type* is one of **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, **getutid**() will find the first entry whose *ut_id* field matches *ut−>ut_id*.

getutline() searches forward from the current file position in the utmp file. It scans entries whose *ut_type* is **USER_PROCESS** or **LOGIN_PROCESS** and returns the first one whose *ut_line* field matches *ut−>ut_line*.

pututline() writes the *utmp* structure *ut* into the utmp file. It uses **getutid**() to search for the proper place in the file to insert the new entry. If it cannot find an appropriate slot for *ut*, **pututline**() will append the new entry to the end of the file.

RETURN VALUE

getutent(), **getutid**(), and **getutline**() return a pointer to a *struct utmp* on success, and NULL on failure (which includes the "record not found" case). This *struct utmp* is allocated in static storage, and may be overwritten by subsequent calls.

On success **pututline**() returns *ut*; on failure, it returns NULL.

utmpname() returns 0 if the new name was successfully stored, or −1 on failure.

On failure, these functions *errno* set to indicate the error.

ERRORS

ENOMEM
: Out of memory.

ESRCH
: Record not found.

setutent(), **pututline**(), and the **getut***() functions can also fail for the reasons described in *open*(2).

FILES

/var/run/utmp
database of currently logged-in users

/var/log/wtmp
database of past user logins

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getutent()	Thread safety	MT-Unsafe init race:utent race:utentbuf sig:ALRM timer
getutid(), **getutline**()	Thread safety	MT-Unsafe init race:utent sig:ALRM timer
pututline()	Thread safety	MT-Unsafe race:utent sig:ALRM timer
setutent(), **endutent**(), **utmpname**()	Thread safety	MT-Unsafe race:utent

In the above table, *utent* in *race:utent* signifies that if any of the functions **setutent**(), **getutent**(), **getutid**(), **getutline**(), **pututline**(), **utmpname**(), or **endutent**() are used in parallel in different threads of a program, then data races could occur.

STANDARDS

None.

HISTORY

XPG2, SVr4.

In XPG2 and SVID 2 the function **pututline**() is documented to return void, and that is what it does on many systems (AIX, HP-UX). HP-UX introduces a new function **_pututline**() with the prototype given above for **pututline**().

All these functions are obsolete now on non-Linux systems. POSIX.1-2001 and POSIX.1-2008, following SUSv1, does not have any of these functions, but instead uses

```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *);
struct utmpx *getutxline(const struct utmpx *);
struct utmpx *pututxline(const struct utmpx *);
void setutxent(void);
void endutxent(void);
```

These functions are provided by glibc, and perform the same task as their equivalents without the "x", but use *struct utmpx*, defined on Linux to be the same as *struct utmp*. For completeness, glibc also provides **utmpxname**(), although this function is not specified by POSIX.1.

On some other systems, the *utmpx* structure is a superset of the *utmp* structure, with additional fields, and larger versions of the existing fields, and parallel files are maintained, often */var/*/utmpx* and */var/*/wtmpx*.

Linux glibc on the other hand does not use a parallel *utmpx* file since its *utmp* structure is already large enough. The "x" functions listed above are just aliases for their counterparts without the "x" (e.g., **getutxent**() is an alias for *getutent*())

NOTES

glibc notes

The above functions are not thread-safe. glibc adds reentrant versions

#include <utmp.h>

int getutent_r(struct utmp ****ubuf*, struct utmp *****ubufp***);**
int getutid_r(struct utmp ****ut*,**
struct utmp ****ubuf*, struct utmp *****ubufp***);**
int getutline_r(struct utmp ****ut*,**
struct utmp ****ubuf*, struct utmp *****ubufp***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getutent_r(), **getutid_r**(), **getutline_r**():
_GNU_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

These functions are GNU extensions, analogs of the functions of the same name without the _r suffix. The *ubuf* argument gives these functions a place to store their result. On success, they return 0, and a pointer to the result is written in **ubufp*. On error, these functions return −1. There are no utmpx equivalents of the above functions. (POSIX.1 does not specify such functions.)

EXAMPLES

The following example adds and removes a utmp record, assuming it is run from within a pseudo terminal. For usage in a real application, you should check the return values of *getpwuid*(3) and *ttyname*(3).

```
#include <pwd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <utmp.h>

int
main(void)
{
    struct utmp entry;

    system("echo before adding entry:;who");

    entry.ut_type = USER_PROCESS;
    entry.ut_pid = getpid();
    strcpy(entry.ut_line, ttyname(STDIN_FILENO) + strlen("/dev/"));
    /* only correct for ptys named /dev/tty[pqr][0-9a-z] */
    strcpy(entry.ut_id, ttyname(STDIN_FILENO) + strlen("/dev/tty"));
    entry.ut_time = time(NULL);
    strcpy(entry.ut_user, getpwuid(getuid())->pw_name);
    memset(entry.ut_host, 0, UT_HOSTSIZE);
    entry.ut_addr = 0;
    setutent();
    pututline(&entry);

    system("echo after adding entry:;who");

    entry.ut_type = DEAD_PROCESS;
    memset(entry.ut_line, 0, UT_LINESIZE);
    entry.ut_time = 0;
    memset(entry.ut_user, 0, UT_NAMESIZE);
    setutent();
    pututline(&entry);

    system("echo after removing entry:;who");

    endutent();
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getutmp(3), *utmp*(5)

NAME

getutmp, getutmpx − copy utmp structure to utmpx, and vice versa

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <utmpx.h>

void getutmp(const struct utmpx *ux, struct utmp *u);
void getutmpx(const struct utmp *u, struct utmpx *ux);
```

DESCRIPTION

The **getutmp**() function copies the fields of the *utmpx* structure pointed to by *ux* to the corresponding fields of the *utmp* structure pointed to by *u*. The **getutmpx**() function performs the converse operation.

RETURN VALUE

These functions do not return a value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getutmp(), **getutmpx**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

glibc 2.1.1. Solaris, NetBSD.

NOTES

These functions exist primarily for compatibility with other systems where the *utmp* and *utmpx* structures contain different fields, or the size of corresponding fields differs. On Linux, the two structures contain the same fields, and the fields have the same sizes.

SEE ALSO

utmpdump(1), *getutent*(3), *utmp*(5)

NAME

getw, putw − input and output of words (ints)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int getw(FILE *stream);
int putw(int w, FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getw(), **putw**():
Since glibc 2.3.3:
_XOPEN_SOURCE && ! (_POSIX_C_SOURCE >= 200112L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
Before glibc 2.3.3:
_SVID_SOURCE || _BSD_SOURCE || _XOPEN_SOURCE

DESCRIPTION

getw() reads a word (that is, an *int*) from *stream*. It's provided for compatibility with SVr4. We recommend you use *fread*(3) instead.

putw() writes the word *w* (that is, an *int*) to *stream*. It is provided for compatibility with SVr4, but we recommend you use *fwrite*(3) instead.

RETURN VALUE

Normally, **getw**() returns the word read, and **putw**() returns 0. On error, they return **EOF**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getw(), **putw**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

SVr4, SUSv2.

BUGS

The value returned on error is also a legitimate data value. *ferror*(3) can be used to distinguish between the two cases.

SEE ALSO

ferror(3), *fread*(3), *fwrite*(3), *getc*(3), *putc*(3)

NAME

getwchar − read a wide character from standard input

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wint_t getwchar(void);

DESCRIPTION

The **getwchar**() function is the wide-character equivalent of the *getchar*(3) function. It reads a wide character from *stdin* and returns it. If the end of stream is reached, or if *ferror(stdin)* becomes true, it returns **WEOF**. If a wide-character conversion error occurs, it sets *errno* to **EILSEQ** and returns **WEOF**.

For a nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

The **getwchar**() function returns the next wide-character from standard input, or **WEOF**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getwchar()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The behavior of **getwchar**() depends on the **LC_CTYPE** category of the current locale.

It is reasonable to expect that **getwchar**() will actually read a multibyte sequence from standard input and then convert it to a wide character.

SEE ALSO

fgetwc(3), *unlocked_stdio*(3)

NAME

glob, globfree − find pathnames matching a pattern, free memory from glob()

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <glob.h>

int glob(const char *restrict pattern, int flags,
    int (*errfunc)(const char *epath, int eerrno),
    glob_t *restrict pglob);
void globfree(glob_t *pglob);
```

DESCRIPTION

The **glob**() function searches for all the pathnames matching *pattern* according to the rules used by the shell (see *glob*(7)). No tilde expansion or parameter substitution is done; if you want these, use *wordexp*(3).

The **globfree**() function frees the dynamically allocated storage from an earlier call to **glob**().

The results of a **glob**() call are stored in the structure pointed to by *pglob*. This structure is of type *glob_t* (declared in <*glob.h*>) and includes the following elements defined by POSIX.2 (more may be present as an extension):

```
typedef struct {
    size_t   gl_pathc;    /* Count of paths matched so far  */
    char   **gl_pathv;    /* List of matched pathnames.  */
    size_t   gl_offs;     /* Slots to reserve in gl_pathv.  */
} glob_t;
```

Results are stored in dynamically allocated storage.

The argument *flags* is made up of the bitwise OR of zero or more the following symbolic constants, which modify the behavior of **glob**():

GLOB_ERR
: Return upon a read error (because a directory does not have read permission, for example). By default, **glob**() attempts carry on despite errors, reading all of the directories that it can.

GLOB_MARK
: Append a slash to each path which corresponds to a directory.

GLOB_NOSORT
: Don’t sort the returned pathnames. The only reason to do this is to save processing time. By default, the returned pathnames are sorted.

GLOB_DOOFFS
: Reserve *pglob−>gl_offs* slots at the beginning of the list of strings in *pglob−>pathv*. The reserved slots contain null pointers.

GLOB_NOCHECK
: If no pattern matches, return the original pattern. By default, **glob**() returns **GLOB_NOMATCH** if there are no matches.

GLOB_APPEND
: Append the results of this call to the vector of results returned by a previous call to **glob**(). Do not set this flag on the first invocation of **glob**().

GLOB_NOESCAPE
: Don’t allow backslash ('\') to be used as an escape character. Normally, a backslash can be used to quote the following character, providing a mechanism to turn off the special meaning metacharacters.

flags may also include any of the following, which are GNU extensions and not defined by POSIX.2:

GLOB_PERIOD
: Allow a leading period to be matched by metacharacters. By default, metacharacters can’t match a leading period.

GLOB_ALTDIRFUNC
Use alternative functions *pglob->gl_closedir*, *pglob->gl_readdir*, *pglob->gl_opendir*, *pglob->gl_lstat*, and *pglob->gl_stat* for filesystem access instead of the normal library functions.

GLOB_BRACE
Expand *csh*(1) style brace expressions of the form **{a,b}**. Brace expressions can be nested. Thus, for example, specifying the pattern "{foo/{,cat,dog},bar}" would return the same results as four separate **glob**() calls using the strings: "foo/", "foo/cat", "foo/dog", and "bar".

GLOB_NOMAGIC
If the pattern contains no metacharacters, then it should be returned as the sole matching word, even if there is no file with that name.

GLOB_TILDE
Carry out tilde expansion. If a tilde ('~') is the only character in the pattern, or an initial tilde is followed immediately by a slash ('/'), then the home directory of the caller is substituted for the tilde. If an initial tilde is followed by a username (e.g., "~andrea/bin"), then the tilde and username are substituted by the home directory of that user. If the username is invalid, or the home directory cannot be determined, then no substitution is performed.

GLOB_TILDE_CHECK
This provides behavior similar to that of **GLOB_TILDE**. The difference is that if the username is invalid, or the home directory cannot be determined, then instead of using the pattern itself as the name, **glob**() returns **GLOB_NOMATCH** to indicate an error.

GLOB_ONLYDIR
This is a *hint* to **glob**() that the caller is interested only in directories that match the pattern. If the implementation can easily determine file-type information, then nondirectory files are not returned to the caller. However, the caller must still check that returned files are directories. (The purpose of this flag is merely to optimize performance when the caller is interested only in directories.)

If *errfunc* is not NULL, it will be called in case of an error with the arguments *epath*, a pointer to the path which failed, and *eerrno*, the value of *errno* as returned from one of the calls to *opendir*(3), *readdir*(3), or *stat*(2). If *errfunc* returns nonzero, or if **GLOB_ERR** is set, **glob**() will terminate after the call to *errfunc*.

Upon successful return, *pglob->gl_pathc* contains the number of matched pathnames and *pglob->gl_pathv* contains a pointer to the list of pointers to matched pathnames. The list of pointers is terminated by a null pointer.

It is possible to call **glob**() several times. In that case, the **GLOB_APPEND** flag has to be set in *flags* on the second and later invocations.

As a GNU extension, *pglob->gl_flags* is set to the flags specified, **or**ed with **GLOB_MAGCHAR** if any metacharacters were found.

RETURN VALUE

On successful completion, **glob**() returns zero. Other possible returns are:

GLOB_NOSPACE
for running out of memory,

GLOB_ABORTED
for a read error, and

GLOB_NOMATCH
for no found matches.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
glob()	Thread safety	MT-Unsafe race:utent env sig:ALRM timer locale
globfree()	Thread safety	MT-Safe

In the above table, *utent* in *race:utent* signifies that if any of the functions *setutent*(3), *getutent*(3), or

endutent(3) are used in parallel in different threads of a program, then data races could occur. **glob**() calls those functions, so we use race:utent to remind users.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, POSIX.2.

NOTES

The structure elements *gl_pathc* and *gl_offs* are declared as *size_t* in glibc 2.1, as they should be according to POSIX.2, but are declared as *int* in glibc 2.0.

BUGS

The **glob**() function may fail due to failure of underlying function calls, such as *malloc*(3) or *opendir*(3). These will store their error code in *errno*.

EXAMPLES

One example of use is the following code, which simulates typing

```
ls -l *.c ../*.c
```

in the shell:

```
glob_t globbuf;

globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob("../*.c", GLOB_DOOFFS | GLOB_APPEND, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp("ls", &globbuf.gl_pathv[0]);
```

SEE ALSO

ls(1), *sh*(1), *stat*(2), *exec*(3), *fnmatch*(3), *malloc*(3), *opendir*(3), *readdir*(3), *wordexp*(3), *glob*(7)

NAME

gnu_get_libc_version, gnu_get_libc_release − get glibc version and release

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <gnu/libc-version.h>

const char *gnu_get_libc_version(void);
const char *gnu_get_libc_release(void);
```

DESCRIPTION

The function **gnu_get_libc_version**() returns a string that identifies the glibc version available on the system.

The function **gnu_get_libc_release**() returns a string indicates the release status of the glibc version available on the system. This will be a string such as *stable*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gnu_get_libc_version(), **gnu_get_libc_release**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

EXAMPLES

When run, the program below will produce output such as the following:

```
$ ./a.out
GNU libc version: 2.8
GNU libc release: stable
```

Program source

```
#include <stdio.h>
#include <stdlib.h>

#include <gnu/libc-version.h>

int
main(void)
{
    printf("GNU libc version: %s\n", gnu_get_libc_version());
    printf("GNU libc release: %s\n", gnu_get_libc_release());
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

confstr(3)

NAME

grantpt − grant access to the slave pseudoterminal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int grantpt(int *fd***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

grantpt():
Since glibc 2.24:
_XOPEN_SOURCE >= 500
glibc 2.23 and earlier:
_XOPEN_SOURCE

DESCRIPTION

The **grantpt**() function changes the mode and owner of the slave pseudoterminal device corresponding to the master pseudoterminal referred to by the file descriptor *fd*. The user ID of the slave is set to the real UID of the calling process. The group ID is set to an unspecified value (e.g., *tty*). The mode of the slave is set to 0620 (crw−−w−−−−).

The behavior of **grantpt**() is unspecified if a signal handler is installed to catch **SIGCHLD** signals.

RETURN VALUE

When successful, **grantpt**() returns 0. Otherwise, it returns −1 and sets *errno* to indicate the error.

ERRORS

EACCES
The corresponding slave pseudoterminal could not be accessed.

EBADF
The *fd* argument is not a valid open file descriptor.

EINVAL
The *fd* argument is valid but not associated with a master pseudoterminal.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
grantpt()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

This is part of the UNIX 98 pseudoterminal support, see *pts*(4).

Historical systems implemented this function via a set-user-ID helper binary called "pt_chown". glibc on Linux before glibc 2.33 could do so as well, in order to support configurations with only BSD pseudoterminals; this support has been removed. On modern systems this is either a no-op —with permissions configured on pty allocation, as is the case on Linux— or an *ioctl*(2).

SEE ALSO

open(2), *posix_openpt*(3), *ptsname*(3), *unlockpt*(3), *pts*(4), *pty*(7)

NAME

group_member − test whether a process is in a group

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int group_member(gid_t *gid***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

group_member():
_GNU_SOURCE

DESCRIPTION

The **group_member**() function tests whether any of the caller's supplementary group IDs (as returned by *getgroups*(2)) matches *gid*.

RETURN VALUE

The **group_member**() function returns nonzero if any of the caller's supplementary group IDs matches *gid*, and zero otherwise.

STANDARDS

GNU.

SEE ALSO

getgid(2), *getgroups*(2), *getgrouplist*(3), *group*(5)

NAME

gsignal, ssignal − software signal facility

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

typedef void (*sighandler_t)(int);

[[deprecated]] int gsignal(int signum);

[[deprecated]] sighandler_t ssignal(int signum, sighandler_t action);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

gsignal(), **ssignal**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

Don't use these functions under Linux. Due to a historical mistake, under Linux these functions are aliases for *raise*(3) and *signal*(2), respectively.

Elsewhere, on System V-like systems, these functions implement software signaling, entirely independent of the classical *signal*(2) and *kill*(2) functions. The function **ssignal**() defines the action to take when the software signal with number *signum* is raised using the function **gsignal**(), and returns the previous such action or **SIG_DFL**. The function **gsignal**() does the following: if no action (or the action **SIG_DFL**) was specified for *signum*, then it does nothing and returns 0. If the action **SIG_IGN** was specified for *signum*, then it does nothing and returns 1. Otherwise, it resets the action to **SIG_DFL** and calls the action function with argument *signum*, and returns the value returned by that function. The range of possible values *signum* varies (often 1–15 or 1–17).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
gsignal()	Thread safety	MT-Safe
ssignal()	Thread safety	MT-Safe sigintr

STANDARDS

None.

HISTORY

AIX, DG/UX, HP-UX, SCO, Solaris, Tru64. They are called obsolete under most of these systems, and are broken under glibc. Some systems also have **gsignal_r**() and **ssignal_r**().

SEE ALSO

kill(2), *signal*(2), *raise*(3)

NAME

hash − hash database access method

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <db.h>

DESCRIPTION

Note well: This page documents interfaces provided up until glibc 2.1. Since glibc 2.2, glibc no longer provides these interfaces. Probably, you are looking for the APIs provided by the *libdb* library instead.

The routine *dbopen*(3) is the library interface to database files. One of the supported file formats is hash files. The general description of the database access methods is in *dbopen*(3), this manual page describes only the hash-specific information.

The hash data structure is an extensible, dynamic hashing scheme.

The access-method-specific data structure provided to *dbopen*(3) is defined in the <*db.h*> include file as follows:

```
typedef struct {
    unsigned int       bsize;
    unsigned int       ffactor;
    unsigned int       nelem;
    unsigned int       cachesize;
    uint32_t         (*hash)(const void *, size_t);
    int         lorder;
} HASHINFO;
```

The elements of this structure are as follows:

bsize defines the hash table bucket size, and is, by default, 256 bytes. It may be preferable to increase the page size for disk-resident tables and tables with large data items.

ffactor indicates a desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. The default value is 8.

nelem is an estimate of the final size of the hash table. If not set or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. The default value is 1.

cachesize is the suggested maximum size, in bytes, of the memory cache. This value is *only advisory*, and the access method will allocate more memory rather than fail.

hash is a user-defined hash function. Since no hash function performs equally well on all possible data, the user may find that the built-in hash function does poorly on a particular data set. A user-specified hash functions must take two arguments (a pointer to a byte string and a length) and return a 32-bit quantity to be used as the hash value.

lorder is the byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified), the current host order is used. If the file already exists, the specified value is ignored and the value specified when the tree was created is used.

If the file already exists (and the **O_TRUNC** flag is not specified), the values specified for *bsize*, *ffactor*, *lorder*, and *nelem* are ignored and the values specified when the tree was created are used.

If a hash function is specified, *hash_open* attempts to determine if the hash function specified is the same as the one with which the database was created, and fails if it is not.

Backward-compatible interfaces to the routines described in *dbm*(3), and *ndbm*(3) are provided, however these interfaces are not compatible with previous file formats.

ERRORS

The *hash* access method routines may fail and set *errno* for any of the errors specified for the library routine *dbopen*(3).

BUGS

Only big and little endian byte order are supported.

SEE ALSO

btree(3), *dbopen*(3), *mpool*(3), *recno*(3)

Dynamic Hash Tables, Per-Ake Larson, Communications of the ACM, April 1988.

A New Hash Package for UNIX, Margo Seltzer, USENIX Proceedings, Winter 1991.

NAME

hcreate, hdestroy, hsearch, hcreate_r, hdestroy_r, hsearch_r − hash table management

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);

ENTRY *hsearch(ENTRY item, ACTION action);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <search.h>

int hcreate_r(size_t nel, struct hsearch_data *htab);
void hdestroy_r(struct hsearch_data *htab);

int hsearch_r(ENTRY item, ACTION action, ENTRY **retval,
        struct hsearch_data *htab);
```

DESCRIPTION

The three functions **hcreate**(), **hsearch**(), and **hdestroy**() allow the caller to create and manage a hash search table containing entries consisting of a key (a string) and associated data. Using these functions, only one hash table can be used at a time.

The three functions **hcreate_r**(), **hsearch_r**(), **hdestroy_r**() are reentrant versions that allow a program to use more than one hash search table at the same time. The last argument, *htab*, points to a structure that describes the table on which the function is to operate. The programmer should treat this structure as opaque (i.e., do not attempt to directly access or modify the fields in this structure).

First a hash table must be created using **hcreate**(). The argument *nel* specifies the maximum number of entries in the table. (This maximum cannot be changed later, so choose it wisely.) The implementation may adjust this value upward to improve the performance of the resulting hash table.

The **hcreate_r**() function performs the same task as **hcreate**(), but for the table described by the structure **htab*. The structure pointed to by *htab* must be zeroed before the first call to **hcreate_r**().

The function **hdestroy**() frees the memory occupied by the hash table that was created by **hcreate**(). After calling **hdestroy**(), a new hash table can be created using **hcreate**(). The **hdestroy_r**() function performs the analogous task for a hash table described by **htab*, which was previously created using **hcreate_r**().

The **hsearch**() function searches the hash table for an item with the same key as *item* (where "the same" is determined using *strcmp*(3)), and if successful returns a pointer to it.

The argument *item* is of type *ENTRY*, which is defined in <*search.h*> as follows:

```
typedef struct entry {
    char *key;
    void *data;
} ENTRY;
```

The field *key* points to a null-terminated string which is the search key. The field *data* points to data that is associated with that key.

The argument *action* determines what **hsearch**() does after an unsuccessful search. This argument must either have the value **ENTER**, meaning insert a copy of *item* (and return a pointer to the new hash table entry as the function result), or the value **FIND**, meaning that NULL should be returned. (If *action* is **FIND**, then *data* is ignored.)

The **hsearch_r**() function is like **hsearch**() but operates on the hash table described by **htab*. The **hsearch_r**() function differs from **hsearch**() in that a pointer to the found item is returned in **retval*, rather than as the function result.

RETURN VALUE

hcreate() and **hcreate_r**() return nonzero on success. They return 0 on error, with *errno* set to indicate the error.

On success, **hsearch**() returns a pointer to an entry in the hash table. **hsearch**() returns NULL on error, that is, if *action* is **ENTER** and the hash table is full, or *action* is **FIND** and *item* cannot be found in the hash table. **hsearch_r**() returns nonzero on success, and 0 on error. In the event of an error, these two functions set *errno* to indicate the error.

ERRORS

hcreate_r() and **hdestroy_r**() can fail for the following reasons:

EINVAL
htab is NULL.

hsearch() and **hsearch_r**() can fail for the following reasons:

ENOMEM
action was **ENTER**, *key* was not found in the table, and there was no room in the table to add a new entry.

ESRCH
action was **FIND**, and *key* was not found in the table.

POSIX.1 specifies only the **ENOMEM** error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
hcreate(), **hsearch**(), **hdestroy**()	Thread safety	MT-Unsafe race:hsearch
hcreate_r(), **hsearch_r**(), **hdestroy_r**()	Thread safety	MT-Safe race:htab

STANDARDS

hcreate()
hsearch()
hdestroy()
POSIX.1-2008.

hcreate_r()
hsearch_r()
hdestroy_r()
GNU.

HISTORY

hcreate()
hsearch()
hdestroy()
SVr4, POSIX.1-2001.

hcreate_r()
hsearch_r()
hdestroy_r()
GNU.

NOTES

Hash table implementations are usually more efficient when the table contains enough free space to minimize collisions. Typically, this means that *nel* should be at least 25% larger than the maximum number of elements that the caller expects to store in the table.

The **hdestroy**() and **hdestroy_r**() functions do not free the buffers pointed to by the *key* and *data* elements of the hash table entries. (It can't do this because it doesn't know whether these buffers were allocated dynamically.) If these buffers need to be freed (perhaps because the program is repeatedly creating and destroying hash tables, rather than creating a single table whose lifetime matches that of the program), then the program must maintain bookkeeping data structures that allow it to free them.

BUGS

SVr4 and POSIX.1-2001 specify that *action* is significant only for unsuccessful searches, so that an **ENTER** should not do anything for a successful search. In libc and glibc (before glibc 2.3), the implementation violates the specification, updating the *data* for the given *key* in this case.

Individual hash table entries can be added, but not deleted.

EXAMPLES

The following program inserts 24 items into a hash table, then prints some of them.

```
#include <search.h>
#include <stdio.h>
#include <stdlib.h>

static char *data[] = { "alpha", "bravo", "charlie", "delta",
     "echo", "foxtrot", "golf", "hotel", "india", "juliet",
     "kilo", "lima", "mike", "november", "oscar", "papa",
     "quebec", "romeo", "sierra", "tango", "uniform",
     "victor", "whisky", "x-ray", "yankee", "zulu"
};

int
main(void)
{
    ENTRY e;
    ENTRY *ep;

    hcreate(30);

    for (size_t i = 0; i < 24; i++) {
        e.key = data[i];
        /* data is just an integer, instead of a
           pointer to something */
        e.data = (void *) i;
        ep = hsearch(e, ENTER);
        /* there should be no failures */
        if (ep == NULL) {
            fprintf(stderr, "entry failed\n");
            exit(EXIT_FAILURE);
        }
    }

    for (size_t i = 22; i < 26; i++) {
        /* print two entries from the table, and
           show that two are not in the table */
        e.key = data[i];
        ep = hsearch(e, FIND);
        printf("%9.9s -> %9.9s:%d\n", e.key,
               ep ? ep->key : "NULL", ep ? (int)(ep->data) : 0);
    }
    hdestroy();
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

bsearch(3), *lsearch*(3), *malloc*(3), *tsearch*(3)

NAME

hypot, hypotf, hypotl − Euclidean distance function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

hypot():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

hypotf(), **hypotl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return sqrt(*x***x*+*y***y*). This is the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*, or the distance of the point (*x*,*y*) from the origin.

The calculation is performed without undue overflow or underflow during the intermediate steps of the calculation.

RETURN VALUE

On success, these functions return the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*.

If *x* or *y* is an infinity, positive infinity is returned.

If *x* or *y* is a NaN, and the other argument is not an infinity, a NaN is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively.

If both arguments are subnormal, and the result is subnormal, a range error occurs, and the correct result is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error: result underflow
: An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

  These functions do not set *errno* for this case.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
hypot(), **hypotf**(), **hypotl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

cabs(3), *sqrt*(3)

NAME

iconv − perform character set conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <iconv.h>

size_t iconv(iconv_t cd,
             char **restrict inbuf, size_t *restrict inbytesleft,
             char **restrict outbuf, size_t *restrict outbytesleft);
```

DESCRIPTION

The **iconv**() function converts a sequence of characters in one character encoding to a sequence of characters in another character encoding. The *cd* argument is a conversion descriptor, previously created by a call to *iconv_open*(3); the conversion descriptor defines the character encodings that **iconv**() uses for the conversion. The *inbuf* argument is the address of a variable that points to the first character of the input sequence; *inbytesleft* indicates the number of bytes in that buffer. The *outbuf* argument is the address of a variable that points to the first byte available in the output buffer; *outbytesleft* indicates the number of bytes available in the output buffer.

The main case is when *inbuf* is not NULL and **inbuf* is not NULL. In this case, the **iconv**() function converts the multibyte sequence starting at **inbuf* to a multibyte sequence starting at **outbuf*. At most **inbytesleft* bytes, starting at **inbuf*, will be read. At most **outbytesleft* bytes, starting at **outbuf*, will be written.

The **iconv**() function converts one multibyte character at a time, and for each character conversion it increments **inbuf* and decrements **inbytesleft* by the number of converted input bytes, it increments **outbuf* and decrements **outbytesleft* by the number of converted output bytes, and it updates the conversion state contained in *cd*. If the character encoding of the input is stateful, the **iconv**() function can also convert a sequence of input bytes to an update to the conversion state without producing any output bytes; such input is called a *shift sequence*. The conversion can stop for five reasons:

- An invalid multibyte sequence is encountered in the input. In this case, it sets *errno* to **EILSEQ** and returns *(size_t) −1*. **inbuf* is left pointing to the beginning of the invalid multibyte sequence.
- A multibyte sequence is encountered that is valid but that cannot be translated to the character encoding of the output. This condition depends on the implementation and on the conversion descriptor. In the GNU C library and GNU libiconv, if *cd* was created without the suffix **//TRANSLIT** or **//IGNORE**, the conversion is strict: lossy conversions produce this condition. If the suffix **//TRANSLIT** was specified, transliteration can avoid this condition in some cases. In the musl C library, this condition cannot occur because a conversion to **'*'** is used as a fallback. In the FreeBSD, NetBSD, and Solaris implementations of **iconv**(), this condition cannot occur either, because a conversion to **'?'** is used as a fallback. When this condition is met, **iconv**() sets *errno* to **EILSEQ** and returns *(size_t) −1*. **inbuf* is left pointing to the beginning of the unconvertible multibyte sequence.
- The input byte sequence has been entirely converted, that is, **inbytesleft* has gone down to 0. In this case, **iconv**() returns the number of nonreversible conversions performed during this call.
- An incomplete multibyte sequence is encountered in the input, and the input byte sequence terminates after it. In this case, it sets *errno* to **EINVAL** and returns *(size_t) −1*. **inbuf* is left pointing to the beginning of the incomplete multibyte sequence.
- The output buffer has no more room for the next converted character. In this case, it sets *errno* to **E2BIG** and returns *(size_t) −1*.

A different case is when *inbuf* is NULL or **inbuf* is NULL, but *outbuf* is not NULL and **outbuf* is not NULL. In this case, the **iconv**() function attempts to set *cd*'s conversion state to the initial state and store a corresponding shift sequence at **outbuf*. At most **outbytesleft* bytes, starting at **outbuf*, will be written. If the output buffer has no more room for this reset sequence, it sets *errno* to **E2BIG** and returns *(size_t) −1*. Otherwise, it increments **outbuf* and decrements **outbytesleft* by the number of bytes written.

A third case is when *inbuf* is NULL or **inbuf* is NULL, and *outbuf* is NULL or **outbuf* is NULL. In

this case, the **iconv**() function sets *cd*'s conversion state to the initial state.

RETURN VALUE

The **iconv**() function returns the number of characters converted in a nonreversible way during this call; reversible conversions are not counted. In case of error, **iconv**() returns *(size_t) −1* and sets *errno* to indicate the error.

ERRORS

The following errors can occur, among others:

E2BIG There is not sufficient room at **outbuf*.

EILSEQ
An invalid multibyte sequence has been encountered in the input.

EINVAL
An incomplete multibyte sequence has been encountered in the input.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iconv()	Thread safety	MT-Safe race:cd

The **iconv**() function is MT-Safe, as long as callers arrange for mutual exclusion on the *cd* argument.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

In each series of calls to **iconv**(), the last should be one with *inbuf* or **inbuf* equal to NULL, in order to flush out any partially converted input.

Although *inbuf* and *outbuf* are typed as *char* **, this does not mean that the objects they point can be interpreted as C strings or as arrays of characters: the interpretation of character byte sequences is handled internally by the conversion functions. In some encodings, a zero byte may be a valid part of a multibyte character.

The caller of **iconv**() must ensure that the pointers passed to the function are suitable for accessing characters in the appropriate character set. This includes ensuring correct alignment on platforms that have tight restrictions on alignment.

SEE ALSO

iconv_close(3), *iconv_open*(3), *iconvconfig*(8)

NAME

iconv_close − deallocate descriptor for character set conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <iconv.h>

int iconv_close(iconv_t *cd***);**

DESCRIPTION

The **iconv_close**() function deallocates a conversion descriptor *cd* previously allocated using *iconv_open*(3).

RETURN VALUE

On success, **iconv_close**() returns 0; otherwise, it returns −1 and sets *errno* to indicate the error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iconv_close()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

SEE ALSO

iconv(3), *iconv_open*(3)

NAME

iconv_open − allocate descriptor for character set conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <iconv.h>

iconv_t iconv_open(const char * *tocode***, const char *** *fromcode***);**

DESCRIPTION

The **iconv_open**() function allocates a conversion descriptor suitable for converting byte sequences from character encoding *fromcode* to character encoding *tocode*.

The values permitted for *fromcode* and *tocode* and the supported combinations are system-dependent. For the GNU C library, the permitted values are listed by the *iconv −−list* command, and all combinations of the listed values are supported. Furthermore the GNU C library and the GNU libiconv library support the following two suffixes:

//TRANSLIT
: When the string "//TRANSLIT" is appended to *tocode*, transliteration is activated. This means that when a character cannot be represented in the target character set, it can be approximated through one or several similarly looking characters.

//IGNORE
: When the string "//IGNORE" is appended to *tocode*, characters that cannot be represented in the target character set will be silently discarded.

The resulting conversion descriptor can be used with *iconv*(3) any number of times. It remains valid until deallocated using *iconv_close*(3).

A conversion descriptor contains a conversion state. After creation using **iconv_open**(), the state is in the initial state. Using *iconv*(3) modifies the descriptor's conversion state. To bring the state back to the initial state, use *iconv*(3) with NULL as *inbuf* argument.

RETURN VALUE

On success, **iconv_open**() returns a freshly allocated conversion descriptor. On failure, it returns *(iconv_t) −1* and sets *errno* to indicate the error.

ERRORS

The following error can occur, among others:

EINVAL
: The conversion from *fromcode* to *tocode* is not supported by the implementation.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iconv_open()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001, SUSv2.

SEE ALSO

iconv(1), *iconv*(3), *iconv_close*(3)

NAME

if_nameindex, if_freenameindex − get network interface names and indexes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <net/if.h>

struct if_nameindex *if_nameindex(*void***);**
void if_freenameindex(struct if_nameindex ** *ptr*);**

DESCRIPTION

The **if_nameindex**() function returns an array of *if_nameindex* structures, each containing information about one of the network interfaces on the local system. The *if_nameindex* structure contains at least the following entries:

```
unsigned int if_index; /* Index of interface (1, 2, ...) */
char        *if_name;  /* Null-terminated name ("eth0", etc.) */
```

The *if_index* field contains the interface index. The *if_name* field points to the null-terminated interface name. The end of the array is indicated by entry with *if_index* set to zero and *if_name* set to NULL.

The data structure returned by **if_nameindex**() is dynamically allocated and should be freed using **if_freenameindex**() when no longer needed.

RETURN VALUE

On success, **if_nameindex**() returns pointer to the array; on error, NULL is returned, and *errno* is set to indicate the error.

ERRORS

if_nameindex() may fail and set *errno* if:

ENOBUFS
Insufficient resources available.

if_nameindex() may also fail for any of the errors specified for *socket*(2), *bind*(2), *ioctl*(2), *getsockname*(2), *recvmsg*(2), *sendto*(2), or *malloc*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
if_nameindex(), **if_freenameindex**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008, RFC 3493.

HISTORY

glibc 2.1. POSIX.1-2001. BSDi.

Before glibc 2.3.4, the implementation supported only interfaces with IPv4 addresses. Support of interfaces that don't have IPv4 addresses is available only on kernels that support netlink.

EXAMPLES

The program below demonstrates the use of the functions described on this page. An example of the output this program might produce is the following:

```
$ ./a.out
1: lo
2: wlan0
3: em1
```

Program source

```
#include <net/if.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
```

```
main(void)
{
    struct if_nameindex *if_ni, *i;

    if_ni = if_nameindex();
    if (if_ni == NULL) {
        perror("if_nameindex");
        exit(EXIT_FAILURE);
    }

    for (i = if_ni; !(i->if_index == 0 && i->if_name == NULL); i++)
        printf("%u: %s\n", i->if_index, i->if_name);

    if_freenameindex(if_ni);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getsockopt(2), *setsockopt*(2), *getifaddrs*(3), *if_indextoname*(3), *if_nametoindex*(3), *ifconfig*(8)

NAME

if_nametoindex, if_indextoname − mappings between network interface names and indexes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
char *if_indextoname(unsigned int ifindex, char *ifname);
```

DESCRIPTION

The **if_nametoindex**() function returns the index of the network interface corresponding to the name *ifname*.

The **if_indextoname**() function returns the name of the network interface corresponding to the interface index *ifindex*. The name is placed in the buffer pointed to by *ifname*. The buffer must allow for the storage of at least **IF_NAMESIZE** bytes.

RETURN VALUE

On success, **if_nametoindex**() returns the index number of the network interface; on error, 0 is returned and *errno* is set to indicate the error.

On success, **if_indextoname**() returns *ifname*; on error, NULL is returned and *errno* is set to indicate the error.

ERRORS

if_nametoindex() may fail and set *errno* if:

ENODEV
No interface found with given name.

if_indextoname() may fail and set *errno* if:

ENXIO
No interface found for the index.

if_nametoindex() and **if_indextoname**() may also fail for any of the errors specified for *socket*(2) or *ioctl*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
if_nametoindex(), **if_indextoname**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008, RFC 3493.

HISTORY

POSIX.1-2001. BSDi.

SEE ALSO

getifaddrs(3), *if_nameindex*(3), *ifconfig*(8)

NAME

ilogb, ilogbf, ilogbl − get integer exponent of a floating-point value

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ilogb():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

ilogbf(), **ilogbl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the exponent part of their argument as a signed integer. When no error occurs, these functions are equivalent to the corresponding *logb*(3) functions, cast to *int*.

RETURN VALUE

On success, these functions return the exponent of *x*, as a signed integer.

If *x* is zero, then a domain error occurs, and the functions return **FP_ILOGB0**.

If *x* is a NaN, then a domain error occurs, and the functions return **FP_ILOGBNAN**.

If *x* is negative infinity or positive infinity, then a domain error occurs, and the functions return **INT_MAX**.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is 0 or a NaN
: An invalid floating-point exception (**FE_INVALID**) is raised, and *errno* is set to **EDOM** (but see BUGS).

Domain error: *x* is an infinity
: An invalid floating-point exception (**FE_INVALID**) is raised, and *errno* is set to **EDOM** (but see BUGS).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ilogb(), **ilogbf**(), **ilogbl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

BUGS

Before glibc 2.16, the following bugs existed in the glibc implementation of these functions:

- The domain error case where *x* is 0 or a NaN did not cause *errno* to be set or (on some architectures) raise a floating-point exception.
- The domain error case where *x* is an infinity did not cause *errno* to be set or raise a floating-point exception.

SEE ALSO

log(3), *logb*(3), *significand*(3)

NAME

index, rindex − locate character in string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <strings.h>

[[deprecated]] char *index(const char *s, int c);
[[deprecated]] char *rindex(const char *s, int c);
```

DESCRIPTION

index() is identical to *strchr*(3).

rindex() is identical to *strrchr*(3).

Use *strchr*(3) and *strrchr*(3) instead of these functions.

STANDARDS

None.

HISTORY

4.3BSD; marked as LEGACY in POSIX.1-2001. Removed in POSIX.1-2008, recommending *strchr*(3) and *strrchr*(3) instead.

SEE ALSO

strchr(3), *strrchr*(3)

NAME

inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof − Internet address manipulation routines

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);

in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);

[[deprecated]] char *inet_ntoa(struct in_addr in);

[[deprecated]] struct in_addr inet_makeaddr(in_addr_t net,
                                            in_addr_t host);

[[deprecated]] in_addr_t inet_lnaof(struct in_addr in);
[[deprecated]] in_addr_t inet_netof(struct in_addr in);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

inet_aton(), **inet_ntoa**():
Since glibc 2.19:
_DEFAULT_SOURCE
In glibc up to and including 2.19:
_BSD_SOURCE || _BSD_SOURCE

DESCRIPTION

inet_aton() converts the Internet host address *cp* from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that *inp* points to. **inet_aton**() returns nonzero if the address is valid, zero if not. The address supplied in *cp* can have one of the following forms:

a.b.c.d Each of the four numeric parts specifies a byte of the address; the bytes are assigned in left-to-right order to produce the binary address.

a.b.c Parts *a* and *b* specify the first two bytes of the binary address. Part *c* is interpreted as a 16-bit value that defines the rightmost two bytes of the binary address. This notation is suitable for specifying (outmoded) Class B network addresses.

a.b Part *a* specifies the first byte of the binary address. Part *b* is interpreted as a 24-bit value that defines the rightmost three bytes of the binary address. This notation is suitable for specifying (outmoded) Class A network addresses.

a The value *a* is interpreted as a 32-bit value that is stored directly into the binary address without any byte rearrangement.

In all of the above forms, components of the dotted address can be specified in decimal, octal (with a leading *0*), or hexadecimal, with a leading *0X*). Addresses in any of these forms are collectively termed *IPV4 numbers-and-dots notation*. The form that uses exactly four decimal numbers is referred to as *IPv4 dotted-decimal notation* (or sometimes: *IPv4 dotted-quad notation*).

inet_aton() returns 1 if the supplied string was successfully interpreted, or 0 if the string is invalid (**errno** is *not* set on error).

The **inet_addr**() function converts the Internet host address *cp* from IPv4 numbers-and-dots notation into binary data in network byte order. If the input is invalid, **INADDR_NONE** (usually −1) is returned. Use of this function is problematic because −1 is a valid address (255.255.255.255). Avoid its use in favor of **inet_aton**(), *inet_pton*(3), or *getaddrinfo*(3), which provide a cleaner way to indicate error return.

The **inet_network**() function converts *cp*, a string in IPv4 numbers-and-dots notation, into a number in host byte order suitable for use as an Internet network address. On success, the converted address is

returned. If the input is invalid, −1 is returned.

The **inet_ntoa**() function converts the Internet host address *in*, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

The **inet_lnaof**() function returns the local network address part of the Internet address *in*. The returned value is in host byte order.

The **inet_netof**() function returns the network number part of the Internet address *in*. The returned value is in host byte order.

The **inet_makeaddr**() function is the converse of **inet_netof**() and **inet_lnaof**(). It returns an Internet host address in network byte order, created by combining the network number *net* with the local address *host*, both in host byte order.

The structure *in_addr* as used in **inet_ntoa**(), **inet_makeaddr**(), **inet_lnaof**(), and **inet_netof**() is defined in <*netinet/in.h*> as:

```
typedef uint32_t in_addr_t;

struct in_addr {
    in_addr_t s_addr;
};
```

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
inet_aton(), **inet_addr**(), **inet_network**(), **inet_ntoa**()	Thread safety	MT-Safe locale
inet_makeaddr(), **inet_lnaof**(), **inet_netof**()	Thread safety	MT-Safe

STANDARDS

inet_addr()
inet_ntoa()
: POSIX.1-2008.

inet_aton()
: None.

STANDARDS

inet_addr()
inet_ntoa()
: POSIX.1-2001, 4.3BSD.

inet_lnaof(), **inet_netof**(), and **inet_makeaddr**() are legacy functions that assume they are dealing with *classful network addresses*. Classful networking divides IPv4 network addresses into host and network components at byte boundaries, as follows:

Class A
: This address type is indicated by the value 0 in the most significant bit of the (network byte ordered) address. The network address is contained in the most significant byte, and the host address occupies the remaining three bytes.

Class B
: This address type is indicated by the binary value 10 in the most significant two bits of the address. The network address is contained in the two most significant bytes, and the host address occupies the remaining two bytes.

Class C
: This address type is indicated by the binary value 110 in the most significant three bits of the address. The network address is contained in the three most significant bytes, and the host address occupies the remaining byte.

Classful network addresses are now obsolete, having been superseded by Classless Inter-Domain Routing (CIDR), which divides addresses into network and host components at arbitrary bit (rather than byte) boundaries.

NOTES

On x86 architectures, the host byte order is Least Significant Byte first (little endian), whereas the network byte order, as used on the Internet, is Most Significant Byte first (big endian).

EXAMPLES

An example of the use of **inet_aton**() and **inet_ntoa**() is shown below. Here are some example runs:

```
$ ./a.out 226.000.000.037      # Last byte is in octal
226.0.0.31
$ ./a.out 0x7f.1               # First byte is in hex
127.0.0.1
```

Program source

```
#define _DEFAULT_SOURCE
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    struct in_addr addr;

    if (argc != 2) {
        fprintf(stderr, "%s <dotted-address>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (inet_aton(argv[1], &addr) == 0) {
        fprintf(stderr, "Invalid address\n");
        exit(EXIT_FAILURE);
    }

    printf("%s\n", inet_ntoa(addr));
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

byteorder(3), *getaddrinfo*(3), *gethostbyname*(3), *getnameinfo*(3), *getnetent*(3), *inet_net_pton*(3), *inet_ntop*(3), *inet_pton*(3), *hosts*(5), *networks*(5)

NAME

inet_net_pton, inet_net_ntop − Internet network number conversion

LIBRARY

Resolver library (*libresolv*, *−lresolv*)

SYNOPSIS

```
#include <arpa/inet.h>

int inet_net_pton(int af, const char *pres,
                  void netp[.nsize], size_t nsize);
char *inet_net_ntop(int af,
                  const void netp[(.bits - CHAR_BIT + 1) / CHAR_BIT],
                  int bits,
                  char  pres[.psize], size_t  psize);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

inet_net_pton(), **inet_net_ntop**():
Since glibc 2.20:
_DEFAULT_SOURCE
Before glibc 2.20:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions convert network numbers between presentation (i.e., printable) format and network (i.e., binary) format.

For both functions, *af* specifies the address family for the conversion; the only supported value is **AF_INET**.

inet_net_pton()

The **inet_net_pton**() function converts *pres*, a null-terminated string containing an Internet network number in presentation format to network format. The result of the conversion, which is in network byte order, is placed in the buffer pointed to by *netp*. (The *netp* argument typically points to an *in_addr* structure.) The *nsize* argument specifies the number of bytes available in *netp*.

On success, **inet_net_pton**() returns the number of bits in the network number field of the result placed in *netp*. For a discussion of the input presentation format and the return value, see NOTES.

Note: the buffer pointed to by *netp* should be zeroed out before calling **inet_net_pton**(), since the call writes only as many bytes as are required for the network number (or as are explicitly specified by *pres*), which may be less than the number of bytes in a complete network address.

inet_net_ntop()

The **inet_net_ntop**() function converts the network number in the buffer pointed to by *netp* to presentation format; **netp* is interpreted as a value in network byte order. The *bits* argument specifies the number of bits in the network number in **netp*.

The null-terminated presentation-format string is placed in the buffer pointed to by *pres*. The *psize* argument specifies the number of bytes available in *pres*. The presentation string is in CIDR format: a dotted-decimal number representing the network address, followed by a slash, and the size of the network number in bits.

RETURN VALUE

On success, **inet_net_pton**() returns the number of bits in the network number. On error, it returns −1, and *errno* is set to indicate the error.

On success, **inet_net_ntop**() returns *pres*. On error, it returns NULL, and *errno* is set to indicate the error.

ERRORS

EAFNOSUPPORT
af specified a value other than **AF_INET**.

EMSGSIZE
The size of the output buffer was insufficient.

ENOENT
(**inet_net_pton**()) *pres* was not in correct presentation format.

STANDARDS

None.

NOTES

Input presentation format for inet_net_pton()

The network number may be specified either as a hexadecimal value or in dotted-decimal notation.

Hexadecimal values are indicated by an initial "0x" or "0X". The hexadecimal digits populate the nibbles (half octets) of the network number from left to right in network byte order.

In dotted-decimal notation, up to four octets are specified, as decimal numbers separated by dots. Thus, any of the following forms are accepted:

```
a.b.c.d
a.b.c
a.b
a
```

Each part is a number in the range 0 to 255 that populates one byte of the resulting network number, going from left to right, in network-byte (big endian) order. Where a part is omitted, the resulting byte in the network number is zero.

For either hexadecimal or dotted-decimal format, the network number can optionally be followed by a slash and a number in the range 0 to 32, which specifies the size of the network number in bits.

Return value of inet_net_pton()

The return value of **inet_net_pton**() is the number of bits in the network number field. If the input presentation string terminates with a slash and an explicit size value, then that size becomes the return value of **inet_net_pton**(). Otherwise, the return value, *bits*, is inferred as follows:

- If the most significant byte of the network number is greater than or equal to 240, then *bits* is 32.
- Otherwise, if the most significant byte of the network number is greater than or equal to 224, then *bits* is 4.
- Otherwise, if the most significant byte of the network number is greater than or equal to 192, then *bits* is 24.
- Otherwise, if the most significant byte of the network number is greater than or equal to 128, then *bits* is 16.
- Otherwise, *bits* is 8.

If the resulting *bits* value from the above steps is greater than or equal to 8, but the number of octets specified in the network number exceed *bits/8*, then *bits* is set to 8 times the number of octets actually specified.

EXAMPLES

The program below demonstrates the use of **inet_net_pton**() and **inet_net_ntop**(). It uses **inet_net_pton**() to convert the presentation format network address provided in its first command-line argument to binary form, displays the return value from **inet_net_pton**(). It then uses **inet_net_ntop**() to convert the binary form back to presentation format, and displays the resulting string.

In order to demonstrate that **inet_net_pton**() may not write to all bytes of its *netp* argument, the program allows an optional second command-line argument, a number used to initialize the buffer before **inet_net_pton**() is called. As its final line of output, the program displays all of the bytes of the buffer returned by **inet_net_pton**() allowing the user to see which bytes have not been touched by **inet_net_pton**().

An example run, showing that **inet_net_pton**() infers the number of bits in the network number:

```
$ ./a.out 193.168
inet_net_pton() returned: 24
inet_net_ntop() yielded:  193.168.0/24
Raw address:              c1a80000
```

Demonstrate that **inet_net_pton**() does not zero out unused bytes in its result buffer:

```
$ ./a.out 193.168 0xffffffff
inet_net_pton() returned: 24
inet_net_ntop() yielded:  193.168.0/24
Raw address:              c1a800ff
```

Demonstrate that **inet_net_pton**() will widen the inferred size of the network number, if the supplied number of bytes in the presentation string exceeds the inferred value:

```
$ ./a.out 193.168.1.128
inet_net_pton() returned: 32
inet_net_ntop() yielded:  193.168.1.128/32
Raw address:              c1a80180
```

Explicitly specifying the size of the network number overrides any inference about its size (but any extra bytes that are explicitly specified will still be used by **inet_net_pton**(): to populate the result buffer):

```
$ ./a.out 193.168.1.128/24
inet_net_pton() returned: 24
inet_net_ntop() yielded:  193.168.1/24
Raw address:              c1a80180
```

Program source

```
/* Link with "-lresolv" */

#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    char buf[100];
    struct in_addr addr;
    int bits;

    if (argc < 2) {
        fprintf(stderr,
                "Usage: %s presentation-form [addr-init-value]\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    /* If argv[2] is supplied (a numeric value), use it to initialize
       the output buffer given to inet_net_pton(), so that we can see
       that inet_net_pton() initializes only those bytes needed for
       the network number. If argv[2] is not supplied, then initialize
       the buffer to zero (as is recommended practice). */

    addr.s_addr = (argc > 2) ? strtod(argv[2], NULL) : 0;

    /* Convert presentation network number in argv[1] to binary. */

    bits = inet_net_pton(AF_INET, argv[1], &addr, sizeof(addr));
    if (bits == -1)
        errExit("inet_net_ntop");

    printf("inet_net_pton() returned: %d\n", bits);
```

```
        /* Convert binary format back to presentation, using 'bits'
           returned by inet_net_pton(). */

        if (inet_net_ntop(AF_INET, &addr, bits, buf, sizeof(buf)) == NULL)
            errExit("inet_net_ntop");

        printf("inet_net_ntop() yielded:  %s\n", buf);

        /* Display 'addr' in raw form (in network byte order), so we can
           see bytes not displayed by inet_net_ntop(); some of those bytes
           may not have been touched by inet_net_ntop(), and so will still
           have any initial value that was specified in argv[2]. */

        printf("Raw address:              %x\n", htonl(addr.s_addr));

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

inet(3), *networks*(5)

NAME

inet_ntop − convert IPv4 and IPv6 addresses from binary to text form

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *restrict src,
                      char dst[restrict .size], socklen_t size);
```

DESCRIPTION

This function converts the network address structure *src* in the *af* address family into a character string. The resulting string is copied to the buffer pointed to by *dst*, which must be a non-null pointer. The caller specifies the number of bytes available in this buffer in the argument *size*.

inet_ntop() extends the *inet_ntoa*(3) function to support multiple address families, *inet_ntoa*(3) is now considered to be deprecated in favor of **inet_ntop**(). The following address families are currently supported:

AF_INET
: *src* points to a *struct in_addr* (in network byte order) which is converted to an IPv4 network address in the dotted-decimal format, "*ddd.ddd.ddd.ddd*". The buffer *dst* must be at least **INET_ADDRSTRLEN** bytes long.

AF_INET6
: *src* points to a *struct in6_addr* (in network byte order) which is converted to a representation of this address in the most appropriate IPv6 network address format for this address. The buffer *dst* must be at least **INET6_ADDRSTRLEN** bytes long.

RETURN VALUE

On success, **inet_ntop**() returns a non-null pointer to *dst*. NULL is returned if there was an error, with *errno* set to indicate the error.

ERRORS

EAFNOSUPPORT
: *af* was not a valid address family.

ENOSPC
: The converted address string would exceed the size given by *size*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
inet_ntop()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Note that RFC 2553 defines a prototype where the last argument *size* is of type *size_t*. Many systems follow RFC 2553. glibc 2.0 and 2.1 have *size_t*, but 2.2 and later have *socklen_t*.

BUGS

AF_INET6 converts IPv4-mapped IPv6 addresses into an IPv6 format.

EXAMPLES

See *inet_pton*(3).

SEE ALSO

getnameinfo(3), *inet*(3), *inet_pton*(3)

NAME

inet_pton − convert IPv4 and IPv6 addresses from text to binary form

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <arpa/inet.h>

int inet_pton(int *af***, const char *restrict** *src***, void *restrict** *dst***);**

DESCRIPTION

This function converts the character string *src* into a network address structure in the *af* address family, then copies the network address structure to *dst*. The *af* argument must be either **AF_INET** or **AF_INET6**. *dst* is written in network byte order.

The following address families are currently supported:

AF_INET

src points to a character string containing an IPv4 network address in dotted-decimal format, "*ddd.ddd.ddd.ddd*", where *ddd* is a decimal number of up to three digits in the range 0 to 255. The address is converted to a *struct in_addr* and copied to *dst*, which must be *sizeof(struct in_addr)* (4) bytes (32 bits) long.

AF_INET6

src points to a character string containing an IPv6 network address. The address is converted to a *struct in6_addr* and copied to *dst*, which must be *sizeof(struct in6_addr)* (16) bytes (128 bits) long. The allowed formats for IPv6 addresses follow these rules:

- The preferred format is *x:x:x:x:x:x:x:x*. This form consists of eight hexadecimal numbers, each of which expresses a 16-bit value (i.e., each *x* can be up to 4 hex digits).
- A series of contiguous zero values in the preferred format can be abbreviated to *::*. Only one instance of *::* can occur in an address. For example, the loopback address *0:0:0:0:0:0:0:1* can be abbreviated as *::1*. The wildcard address, consisting of all zeros, can be written as *::*.
- An alternate format is useful for expressing IPv4-mapped IPv6 addresses. This form is written as *x:x:x:x:x:x:d.d.d.d*, where the six leading *x*s are hexadecimal values that define the six most-significant 16-bit pieces of the address (i.e., 96 bits), and the *d*s express a value in dotted-decimal notation that defines the least significant 32 bits of the address. An example of such an address is *::FFFF:204.152.189.116*.

See RFC 2373 for further details on the representation of IPv6 addresses.

RETURN VALUE

inet_pton() returns 1 on success (network address was successfully converted). 0 is returned if *src* does not contain a character string representing a valid network address in the specified address family. If *af* does not contain a valid address family, −1 is returned and *errno* is set to **EAFNOSUPPORT**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
inet_pton()	Thread safety	MT-Safe locale

VERSIONS

Unlike *inet_aton*(3) and *inet_addr*(3), **inet_pton**() supports IPv6 addresses. On the other hand, **inet_pton**() accepts only IPv4 addresses in dotted-decimal notation, whereas *inet_aton*(3) and *inet_addr*(3) allow the more general numbers-and-dots notation (hexadecimal and octal number formats, and formats that don't require all four bytes to be explicitly written). For an interface that handles both IPv6 addresses, and IPv4 addresses in numbers-and-dots notation, see *getaddrinfo*(3).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

BUGS

AF_INET6 does not recognize IPv4 addresses. An explicit IPv4-mapped IPv6 address must be supplied in *src* instead.

EXAMPLES

The program below demonstrates the use of **inet_pton**() and *inet_ntop*(3). Here are some example runs:

```
$ ./a.out i6 0:0:0:0:0:0:0:0
::
$ ./a.out i6 1:0:0:0:0:0:0:8
1::8
$ ./a.out i6 0:0:0:0:0:FFFF:204.152.189.116
::ffff:204.152.189.116
```

Program source

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    unsigned char buf[sizeof(struct in6_addr)];
    int domain, s;
    char str[INET6_ADDRSTRLEN];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s {i4|i6|<num>} string\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    domain = (strcmp(argv[1], "i4") == 0) ? AF_INET :
             (strcmp(argv[1], "i6") == 0) ? AF_INET6 : atoi(argv[1]);

    s = inet_pton(domain, argv[2], buf);
    if (s <= 0) {
        if (s == 0)
            fprintf(stderr, "Not in presentation format");
        else
            perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(domain, buf, str, INET6_ADDRSTRLEN) == NULL) {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    printf("%s\n", str);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getaddrinfo(3), *inet*(3), *inet_ntop*(3)

NAME

INFINITY, NAN, HUGE_VAL, HUGE_VALF, HUGE_VALL − floating-point constants

LIBRARY

Math library (*libm*)

SYNOPSIS

```
#define _ISOC99_SOURCE      /* See feature_test_macros(7) */
#include <math.h>

INFINITY

NAN

HUGE_VAL
HUGE_VALF
HUGE_VALL
```

DESCRIPTION

The macro **INFINITY** expands to a *float* constant representing positive infinity.

The macro **NAN** expands to a *float* constant representing a quiet NaN (when supported). A *quiet* NaN is a NaN ("not-a-number") that does not raise exceptions when it is used in arithmetic. The opposite is a *signaling* NaN. See IEC 60559:1989.

The macros **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL** expand to constants of types *double*, *float*, and *long double*, respectively, that represent a large positive value, possibly positive infinity.

STANDARDS

C11.

HISTORY

C99.

On a glibc system, the macro **HUGE_VAL** is always available. Availability of the **NAN** macro can be tested using **#ifdef NAN**, and similarly for **INFINITY**, **HUGE_VALF**, **HUGE_VALL**. They will be defined by <*math.h*> if **_ISOC99_SOURCE** or **_GNU_SOURCE** is defined, or **__STDC_VERSION__** is defined and has a value not less than 199901L.

SEE ALSO

fpclassify(3), *math_error*(7)

NAME

initgroups − initialize the supplementary group access list

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

int initgroups(const char *user, gid_t group);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

initgroups():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **initgroups**() function initializes the group access list by reading the group database */etc/group* and using all groups of which *user* is a member. The additional group *group* is also added to the list.

The *user* argument must be non-NULL.

RETURN VALUE

The **initgroups**() function returns 0 on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

ENOMEM
Insufficient memory to allocate group information structure.

EPERM
The calling process has insufficient privilege. See the underlying system call *setgroups*(2).

FILES

/etc/group
group database file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
initgroups()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

SVr4, 4.3BSD.

SEE ALSO

getgroups(2), *setgroups*(2), *credentials*(7)

NAME

insque, remque − insert/remove an item from a queue

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <search.h>

void insque(void ****elem*, void ****prev***);**
void remque(void ****elem*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

insque(), **remque**():
_XOPEN_SOURCE >= 500
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

DESCRIPTION

The **insque**() and **remque**() functions manipulate doubly linked lists. Each element in the list is a structure of which the first two elements are a forward and a backward pointer. The linked list may be linear (i.e., NULL forward pointer at the end of the list and NULL backward pointer at the start of the list) or circular.

The **insque**() function inserts the element pointed to by *elem* immediately after the element pointed to by *prev*.

If the list is linear, then the call *insque(elem, NULL)* can be used to insert the initial list element, and the call sets the forward and backward pointers of *elem* to NULL.

If the list is circular, the caller should ensure that the forward and backward pointers of the first element are initialized to point to that element, and the *prev* argument of the **insque**() call should also point to the element.

The **remque**() function removes the element pointed to by *elem* from the doubly linked list.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
insque(), **remque**()	Thread safety	MT-Safe

VERSIONS

On ancient systems, the arguments of these functions were of type *struct qelem* *, defined as:

```
struct qelem {
    struct qelem *q_forw;
    struct qelem *q_back;
    char          q_data[1];
};
```

This is still what you will get if **_GNU_SOURCE** is defined before including <*search.h*>.

The location of the prototypes for these functions differs among several versions of UNIX. The above is the POSIX version. Some systems place them in <*string.h*>.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

BUGS

In glibc 2.4 and earlier, it was not possible to specify *prev* as NULL. Consequently, to build a linear list, the caller had to build a list using an initial call that contained the first two elements of the list, with the forward and backward pointers in each element suitably initialized.

EXAMPLES

The program below demonstrates the use of **insque**(). Here is an example run of the program:

```
$ ./a.out -c a b c
Traversing completed list:
    a
    b
    c
That was a circular list
```

Program source

```
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

struct element {
    struct element *forward;
    struct element *backward;
    char *name;
};

static struct element *
new_element(void)
{
    struct element *e;

    e = malloc(sizeof(*e));
    if (e == NULL) {
        fprintf(stderr, "malloc() failed\n");
        exit(EXIT_FAILURE);
    }

    return e;
}

int
main(int argc, char *argv[])
{
    struct element *first, *elem, *prev;
    int circular, opt, errfnd;

    /* The "-c" command-line option can be used to specify that the
       list is circular. */

    errfnd = 0;
    circular = 0;
    while ((opt = getopt(argc, argv, "c")) != -1) {
        switch (opt) {
        case 'c':
            circular = 1;
            break;
        default:
            errfnd = 1;
            break;
        }
    }

    if (errfnd || optind >= argc) {
        fprintf(stderr,  "Usage: %s [-c] string...\n", argv[0]);
        exit(EXIT_FAILURE);
```

```
        }

        /* Create first element and place it in the linked list. */

        elem = new_element();
        first = elem;

        elem->name = argv[optind];

        if (circular) {
            elem->forward = elem;
            elem->backward = elem;
            insque(elem, elem);
        } else {
            insque(elem, NULL);
        }

        /* Add remaining command-line arguments as list elements. */

        while (++optind < argc) {
            prev = elem;

            elem = new_element();
            elem->name = argv[optind];
            insque(elem, prev);
        }

        /* Traverse the list from the start, printing element names. */

        printf("Traversing completed list:\n");
        elem = first;
        do {
            printf("    %s\n", elem->name);
            elem = elem->forward;
        } while (elem != NULL && elem != first);

        if (elem == first)
            printf("That was a circular list\n");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

queue(7)

NAME

isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isalnum_l, isalpha_l, isascii_l, isblank_l, iscntrl_l, isdigit_l, isgraph_l, islower_l, isprint_l, ispunct_l, isspace_l, isupper_l, isxdigit_l − character classification functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);

int isascii(int c);
int isblank(int c);

int isalnum_l(int c, locale_t locale);
int isalpha_l(int c, locale_t locale);
int isblank_l(int c, locale_t locale);
int iscntrl_l(int c, locale_t locale);
int isdigit_l(int c, locale_t locale);
int isgraph_l(int c, locale_t locale);
int islower_l(int c, locale_t locale);
int isprint_l(int c, locale_t locale);
int ispunct_l(int c, locale_t locale);
int isspace_l(int c, locale_t locale);
int isupper_l(int c, locale_t locale);
int isxdigit_l(int c, locale_t locale);

int isascii_l(int c, locale_t locale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

isascii():
  _XOPEN_SOURCE
    || /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE

isblank():
  _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

salnum_l(), **salpha_l**(), **sblank_l**(), **scntrl_l**(), **sdigit_l**(), **sgraph_l**(), **slower_l**(), **sprint_l**(), **spunct_l**(), **sspace_l**(), **supper_l**(), **sxdigit_l**():
  Since glibc 2.10:
    _XOPEN_SOURCE >= 700
  Before glibc 2.10:
    _GNU_SOURCE

isascii_l():
  Since glibc 2.10:
    _XOPEN_SOURCE >= 700 && (_SVID_SOURCE || _BSD_SOURCE)
  Before glibc 2.10:
    _GNU_SOURCE

DESCRIPTION

These functions check whether *c*, which must have the value of an *unsigned char* or **EOF**, falls into a certain character class according to the specified locale. The functions without the "_l" suffix perform the check based on the current locale.

The functions with the "_l" suffix perform the check based on the locale specified by the locale object *locale*. The behavior of these functions is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)) or is not a valid locale object handle.

The list below explains the operation of the functions without the "_l" suffix; the functions with the "_l" suffix differ only in using the locale object *locale* instead of the current locale.

isalnum()
: checks for an alphanumeric character; it is equivalent to **(isalpha(***c***) || isdigit(***c***))**.

isalpha()
: checks for an alphabetic character; in the standard **"C"** locale, it is equivalent to **(isupper(***c***) || islower(***c***))**. In some locales, there may be additional characters for which **isalpha**() is true—letters which are neither uppercase nor lowercase.

isascii()
: checks whether *c* is a 7-bit *unsigned char* value that fits into the ASCII character set.

isblank()
: checks for a blank character; that is, a space or a tab.

iscntrl()
: checks for a control character.

isdigit()
: checks for a digit (0 through 9).

isgraph()
: checks for any printable character except space.

islower()
: checks for a lowercase character.

isprint()
: checks for any printable character including space.

ispunct()
: checks for any printable character which is not a space or an alphanumeric character.

isspace()
: checks for white-space characters. In the **"C"** and **"POSIX"** locales, these are: space, form-feed (**'\f'**), newline (**'\n'**), carriage return (**'\r'**), horizontal tab (**'\t'**), and vertical tab (**'\v'**).

isupper()
: checks for an uppercase letter.

isxdigit()
: checks for hexadecimal digits, that is, one of
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

RETURN VALUE

The values returned are nonzero if the character *c* falls into the tested class, and zero if not.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
isalnum(), **isalpha**(), **isascii**(), **isblank**(), **iscntrl**(), **isdigit**(), **isgraph**(), **islower**(), **isprint**(), **ispunct**(), **isspace**(), **isupper**(), **isxdigit**()	Thread safety	MT-Safe

STANDARDS

isalnum()

isalpha()
iscntrl()
isdigit()
isgraph()
islower()
isprint()
ispunct()
isspace()
isupper()
isxdigit()
isblank()
C11, POSIX.1-2008.

isascii()
isalnum_l()
isalpha_l()
isblank_l()
iscntrl_l()
isdigit_l()
isgraph_l()
islower_l()
isprint_l()
ispunct_l()
isspace_l()
isupper_l()
isxdigit_l()
POSIX.1-2008.

isascii_l()
GNU.

HISTORY

isalnum()
isalpha()
iscntrl()
isdigit()
isgraph()
islower()
isprint()
ispunct()
isspace()
isupper()
isxdigit()
C89, POSIX.1-2001.

isblank()
C99, POSIX.1-2001.

isascii()
POSIX.1-2001 (XSI).

POSIX.1-2008 marks it as obsolete, noting that it cannot be used portably in a localized application.

isalnum_l()
isalpha_l()
isblank_l()
iscntrl_l()
isdigit_l()
isgraph_l()
islower_l()

isprint_l()
ispunct_l()
isspace_l()
isupper_l()
isxdigit_l()
glibc 2.3. POSIX.1-2008.

isascii_l()
glibc 2.3.

CAVEATS

The standards require that the argument *c* for these functions is either **EOF** or a value that is representable in the type *unsigned char*; otherwise, the behavior is undefined. If the argument *c* is of type *char*, it must be cast to *unsigned char*, as in the following example:

```
char c;
...
res = toupper((unsigned char) c);
```

This is necessary because *char* may be the equivalent of *signed char*, in which case a byte where the top bit is set would be sign extended when converting to *int*, yielding a value that is outside the range of *unsigned char*.

The details of what characters belong to which class depend on the locale. For example, **isupper**() will not recognize an A-umlaut (Ä) as an uppercase letter in the default **C** locale.

SEE ALSO

iswalnum(3), *iswalpha*(3), *iswblank*(3), *iswcntrl*(3), *iswdigit*(3), *iswgraph*(3), *iswlower*(3), *iswprint*(3), *iswpunct*(3), *iswspace*(3), *iswupper*(3), *iswxdigit*(3), *newlocale*(3), *setlocale*(3), *toascii*(3), *tolower*(3), *toupper*(3), *uselocale*(3), *ascii*(7), *locale*(7)

NAME

isatty − test whether a file descriptor refers to a terminal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int isatty(int *fd***);**

DESCRIPTION

The **isatty**() function tests whether *fd* is an open file descriptor referring to a terminal.

RETURN VALUE

isatty() returns 1 if *fd* is an open file descriptor referring to a terminal; otherwise 0 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF

fd is not a valid file descriptor.

ENOTTY

fd refers to a file other than a terminal. On some older kernels, some types of files resulted in the error **EINVAL** in this case (which is a violation of POSIX, which specifies the error **ENOTTY**).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
isatty()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

fstat(2), *ttyname*(3)

NAME

isfdtype − test file type of a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/stat.h>
#include <sys/socket.h>

int isfdtype(int *fd***, int** *fdtype***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

isfdtype():
Since glibc 2.20:
_DEFAULT_SOURCE
Before glibc 2.20:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **isfdtype**() function tests whether the file descriptor *fd* refers to a file of type *fdtype*. The *fdtype* argument specifies one of the **S_IF*** constants defined in <*sys/stat.h*> and documented in *stat*(2) (e.g., **S_IFREG**).

RETURN VALUE

The **isfdtype**() function returns 1 if the file descriptor *fd* is of type *fdtype* and 0 if it is not. On failure, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **isfdtype**() function can fail with any of the same errors as *fstat*(2).

VERSIONS

Portable applications should use *fstat*(2) instead.

STANDARDS

None.

HISTORY

It appeared in the draft POSIX.1g standard. It is present on OpenBSD and Tru64 UNIX (where the required header file in both cases is just <*sys/stat.h*>, as shown in the POSIX.1g draft).

SEE ALSO

fstat(2)

NAME

isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered − floating-point relational tests without exception for NaN

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

int isgreater(x, y);
int isgreaterequal(x, y);
int isless(x, y);
int islessequal(x, y);
int islessgreater(x, y);
int isunordered(x, y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions described here:
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The normal relational operations (like <, "less than") fail if one of the operands is NaN. This will cause an exception. To avoid this, C99 defines the macros listed below.

These macros are guaranteed to evaluate their arguments only once. The arguments must be of real floating-point type (note: do not pass integer values as arguments to these macros, since the arguments will *not* be promoted to real-floating types).

isgreater()
: determines *(x) > (y)* without an exception if *x* or *y* is NaN.

isgreaterequal()
: determines *(x) >= (y)* without an exception if *x* or *y* is NaN.

isless() determines *(x) < (y)* without an exception if *x* or *y* is NaN.

islessequal()
: determines *(x) <= (y)* without an exception if *x* or *y* is NaN.

islessgreater()
: determines *(x) < (y) || (x) > (y)* without an exception if *x* or *y* is NaN. This macro is not equivalent to *x != y* because that expression is true if *x* or *y* is NaN.

isunordered()
: determines whether its arguments are unordered, that is, whether at least one of the arguments is a NaN.

RETURN VALUE

The macros other than **isunordered**() return the result of the relational comparison; these macros return 0 if either argument is a NaN.

isunordered() returns 1 if *x* or *y* is NaN and 0 otherwise.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
isgreater(), **isgreaterequal**(), **isless**(), **islessequal**(), **islessgreater**(), **isunordered**()	Thread safety	MT-Safe

VERSIONS

Not all hardware supports these functions, and where hardware support isn't provided, they will be emulated by macros. This will result in a performance penalty. Don't use these functions if NaN is of no concern for you.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

fpclassify(3), *isnan*(3)

NAME

iswalnum − test for alphanumeric wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswalnum(wint_t *wc***);**

DESCRIPTION

The **iswalnum**() function is the wide-character equivalent of the *isalnum*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "alnum".

The wide-character class "alnum" is a subclass of the wide-character class "graph", and therefore also a subclass of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "alnum" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "alnum" is disjoint from the wide-character class "space" and its subclass "blank".

The wide-character class "alnum" is disjoint from the wide-character class "punct".

The wide-character class "alnum" is the union of the wide-character classes "alpha" and "digit". As such, it also contains the wide-character class "xdigit".

The wide-character class "alnum" always contains at least the letters 'A' to 'Z', 'a' to 'z', and the digits '0' to '9'.

RETURN VALUE

The **iswalnum**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "alnum". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswalnum()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswalnum**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isalnum(3), *iswctype*(3)

NAME

iswalpha − test for alphabetic wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswalpha(wint_t *wc***);**

DESCRIPTION

The **iswalpha**() function is the wide-character equivalent of the *isalpha*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "alpha".

The wide-character class "alpha" is a subclass of the wide-character class "alnum", and therefore also a subclass of the wide-character class "graph" and of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "alpha" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "alpha" is disjoint from the wide-character class "space" and its subclass "blank".

Being a subclass of the wide-character class "alnum", the wide-character class "alpha" is disjoint from the wide-character class "punct".

The wide-character class "alpha" is disjoint from the wide-character class "digit".

The wide-character class "alpha" contains the wide-character classes "upper" and "lower".

The wide-character class "alpha" always contains at least the letters 'A' to 'Z' and 'a' to 'z'.

RETURN VALUE

The **iswalpha**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "alpha". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswalpha()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswalpha**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isalpha(3), *iswctype*(3)

NAME

iswblank − test for whitespace wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswblank(wint_t *wc***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

iswblank():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **iswblank**() function is the wide-character equivalent of the *isblank*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "blank".

The wide-character class "blank" is a subclass of the wide-character class "space".

Being a subclass of the wide-character class "space", the wide-character class "blank" is disjoint from the wide-character class "graph" and therefore also disjoint from its subclasses "alnum", "alpha", "upper", "lower", "digit", "xdigit", "punct".

The wide-character class "blank" always contains at least the space character and the control character '\t'.

RETURN VALUE

The **iswblank**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "blank". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswblank()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The behavior of **iswblank**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isblank(3), *iswctype*(3)

NAME

iswcntrl − test for control wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswcntrl(wint_t *wc***);**

DESCRIPTION

The **iswcntrl**() function is the wide-character equivalent of the *iscntrl*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "cntrl".

The wide-character class "cntrl" is disjoint from the wide-character class "print" and therefore also disjoint from its subclasses "graph", "alpha", "upper", "lower", "digit", "xdigit", "punct".

For an unsigned char *c*, *iscntrl(c)* implies *iswcntrl(btowc(c))*, but not vice versa.

RETURN VALUE

The **iswcntrl**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "cntrl". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswcntrl()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswcntrl**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iscntrl(3), *iswctype*(3)

NAME

iswctype − wide-character classification

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswctype(wint_t *wc***, wctype_t** *desc***);**

DESCRIPTION

If *wc* is a wide character having the character property designated by *desc* (or in other words: belongs to the character class designated by *desc*), then the **iswctype**() function returns nonzero. Otherwise, it returns zero. If *wc* is **WEOF**, zero is returned.

desc must be a character property descriptor returned by the *wctype*(3) function.

RETURN VALUE

The **iswctype**() function returns nonzero if the *wc* has the designated property. Otherwise, it returns 0.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswctype()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswctype**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iswalnum(3), *iswalpha*(3), *iswblank*(3), *iswcntrl*(3), *iswdigit*(3), *iswgraph*(3), *iswlower*(3), *iswprint*(3), *iswpunct*(3), *iswspace*(3), *iswupper*(3), *iswxdigit*(3), *wctype*(3)

NAME

iswdigit − test for decimal digit wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswdigit(wint_t *wc***);**

DESCRIPTION

The **iswdigit**() function is the wide-character equivalent of the *isdigit*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "digit".

The wide-character class "digit" is a subclass of the wide-character class "xdigit", and therefore also a subclass of the wide-character class "alnum", of the wide-character class "graph" and of the wide-character class "print".

Being a subclass of the wide character class "print", the wide-character class "digit" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "digit" is disjoint from the wide-character class "space" and its subclass "blank".

Being a subclass of the wide-character class "alnum", the wide-character class "digit" is disjoint from the wide-character class "punct".

The wide-character class "digit" is disjoint from the wide-character class "alpha" and therefore also disjoint from its subclasses "lower", "upper".

The wide-character class "digit" always contains exactly the digits '0' to '9'.

RETURN VALUE

The **iswdigit**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "digit". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswdigit()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswdigit**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isdigit(3), *iswctype*(3)

NAME

iswgraph − test for graphic wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswgraph(wint_t *wc***);**

DESCRIPTION

The **iswgraph**() function is the wide-character equivalent of the *isgraph*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "graph".

The wide-character class "graph" is a subclass of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "graph" is disjoint from the wide-character class "cntrl".

The wide-character class "graph" is disjoint from the wide-character class "space" and therefore also disjoint from its subclass "blank".

The wide-character class "graph" contains all the wide characters from the wide-character class "print" except the space character. It therefore contains the wide-character classes "alnum" and "punct".

RETURN VALUE

The **iswgraph**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "graph". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswgraph()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswgraph**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isgraph(3), *iswctype*(3)

NAME

iswlower − test for lowercase wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswlower(wint_t *wc***);**

DESCRIPTION

The **iswlower**() function is the wide-character equivalent of the *islower*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "lower".

The wide-character class "lower" is a subclass of the wide-character class "alpha", and therefore also a subclass of the wide-character class "alnum", of the wide-character class "graph" and of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "lower" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "lower" is disjoint from the wide-character class "space" and its subclass "blank".

Being a subclass of the wide-character class "alnum", the wide-character class "lower" is disjoint from the wide-character class "punct".

Being a subclass of the wide-character class "alpha", the wide-character class "lower" is disjoint from the wide-character class "digit".

The wide-character class "lower" contains at least those characters *wc* which are equal to *towlower(wc)* and different from *towupper(wc)*.

The wide-character class "lower" always contains at least the letters 'a' to 'z'.

RETURN VALUE

The **iswlower**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "lower". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswlower()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswlower**() depends on the **LC_CTYPE** category of the current locale.

This function is not very appropriate for dealing with Unicode characters, because Unicode knows about three cases: upper, lower, and title case.

SEE ALSO

islower(3), *iswctype*(3), *towlower*(3)

NAME

iswprint − test for printing wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswprint(wint_t *wc***);**

DESCRIPTION

The **iswprint**() function is the wide-character equivalent of the *isprint*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "print".

The wide-character class "print" is disjoint from the wide-character class "cntrl".

The wide-character class "print" contains the wide-character class "graph".

RETURN VALUE

The **iswprint**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "print". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswprint()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswprint**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isprint(3), *iswctype*(3)

NAME

iswpunct − test for punctuation or symbolic wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswpunct(wint_t *wc***);**

DESCRIPTION

The **iswpunct**() function is the wide-character equivalent of the *ispunct*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "punct".

The wide-character class "punct" is a subclass of the wide-character class "graph", and therefore also a subclass of the wide-character class "print".

The wide-character class "punct" is disjoint from the wide-character class "alnum" and therefore also disjoint from its subclasses "alpha", "upper", "lower", "digit", "xdigit".

Being a subclass of the wide-character class "print", the wide-character class "punct" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "punct" is disjoint from the wide-character class "space" and its subclass "blank".

RETURN VALUE

The **iswpunct**() function returns nonzero if *wc* is a wide-character belonging to the wide-character class "punct". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswpunct()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswpunct**() depends on the **LC_CTYPE** category of the current locale.

This function’s name is a misnomer when dealing with Unicode characters, because the wide-character class "punct" contains both punctuation characters and symbol (math, currency, etc.) characters.

SEE ALSO

ispunct(3), *iswctype*(3)

NAME

iswspace − test for whitespace wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswspace(wint_t *wc***);**

DESCRIPTION

The **iswspace**() function is the wide-character equivalent of the *isspace*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "space".

The wide-character class "space" is disjoint from the wide-character class "graph" and therefore also disjoint from its subclasses "alnum", "alpha", "upper", "lower", "digit", "xdigit", "punct".

The wide-character class "space" contains the wide-character class "blank".

The wide-character class "space" always contains at least the space character and the control characters '\f', '\n', '\r', '\t', and '\v'.

RETURN VALUE

The **iswspace**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "space". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswspace()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswspace**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

isspace(3), *iswctype*(3)

NAME

iswupper − test for uppercase wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswupper(wint_t *wc***);**

DESCRIPTION

The **iswupper**() function is the wide-character equivalent of the *isupper*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "upper".

The wide-character class "upper" is a subclass of the wide-character class "alpha", and therefore also a subclass of the wide-character class "alnum", of the wide-character class "graph" and of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "upper" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "upper" is disjoint from the wide-character class "space" and its subclass "blank".

Being a subclass of the wide-character class "alnum", the wide-character class "upper" is disjoint from the wide-character class "punct".

Being a subclass of the wide-character class "alpha", the wide-character class "upper" is disjoint from the wide-character class "digit".

The wide-character class "upper" contains at least those characters *wc* which are equal to *towupper(wc)* and different from *towlower(wc)*.

The wide-character class "upper" always contains at least the letters 'A' to 'Z'.

RETURN VALUE

The **iswupper**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "upper". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswupper()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswupper**() depends on the **LC_CTYPE** category of the current locale.

This function is not very appropriate for dealing with Unicode characters, because Unicode knows about three cases: upper, lower, and title case.

SEE ALSO

isupper(3), *iswctype*(3), *towupper*(3)

NAME

iswxdigit − test for hexadecimal digit wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

int iswxdigit(wint_t *wc***);**

DESCRIPTION

The **iswxdigit**() function is the wide-character equivalent of the *isxdigit*(3) function. It tests whether *wc* is a wide character belonging to the wide-character class "xdigit".

The wide-character class "xdigit" is a subclass of the wide-character class "alnum", and therefore also a subclass of the wide-character class "graph" and of the wide-character class "print".

Being a subclass of the wide-character class "print", the wide-character class "xdigit" is disjoint from the wide-character class "cntrl".

Being a subclass of the wide-character class "graph", the wide-character class "xdigit" is disjoint from the wide-character class "space" and its subclass "blank".

Being a subclass of the wide-character class "alnum", the wide-character class "xdigit" is disjoint from the wide-character class "punct".

The wide-character class "xdigit" always contains at least the letters 'A' to 'F', 'a' to 'f' and the digits '0' to '9'.

RETURN VALUE

The **iswxdigit**() function returns nonzero if *wc* is a wide character belonging to the wide-character class "xdigit". Otherwise, it returns zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
iswxdigit()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **iswxdigit**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iswctype(3), *isxdigit*(3)

NAME

j0, j0f, j0l, j1, j1f, j1l, jn, jnf, jnl − Bessel functions of the first kind

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double j0(double x);
double j1(double x);
double jn(int n, double x);

float j0f(float x);
float j1f(float x);
float jnf(int n, float x);

long double j0l(long double x);
long double j1l(long double x);
long double jnl(int n, long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

j0(), **j1**(), **jn**():
_XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

j0f(), **j0l**(), **j1f**(), **j1l**(), **jnf**(), **jnl**():
_XOPEN_SOURCE >= 600
|| (_ISOC99_SOURCE && _XOPEN_SOURCE)
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

The **j0**() and **j1**() functions return Bessel functions of *x* of the first kind of orders 0 and 1, respectively. The **jn**() function returns the Bessel function of *x* of the first kind of order *n*.

The **j0f**(), **j1f**(), and **jnf**(), functions are versions that take and return *float* values. The **j0l**(), **j1l**(), and **jnl**() functions are versions that take and return *long double* values.

RETURN VALUE

On success, these functions return the appropriate Bessel value of the first kind for *x*.

If *x* is a NaN, a NaN is returned.

If *x* is too large in magnitude, or the result underflows, a range error occurs, and the return value is 0.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result underflow, or *x* is too large in magnitude
errno is set to **ERANGE**.

These functions do not raise exceptions for *fetestexcept*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
j0(), **j0f**(), **j0l**()	Thread safety	MT-Safe
j1(), **j1f**(), **j1l**()	Thread safety	MT-Safe
jn(), **jnf**(), **jnl**()	Thread safety	MT-Safe

STANDARDS

j0()
j1()
jn() POSIX.1-2008.

Others: BSD.

HISTORY

j0()
j1()
jn() SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.

Others: BSD.

BUGS

There are errors of up to 2e−16 in the values returned by **j0**(), **j1**(), and **jn**() for values of *x* between −8 and 8.

SEE ALSO

y0(3)

NAME

key_decryptsession, key_encryptsession, key_setsecret, key_gendes, key_secretkey_is_set − interfaces to rpc keyserver daemon

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <rpc/rpc.h>

int key_decryptsession(char ****remotename*, des_block ****deskey***);**
int key_encryptsession(char ****remotename*, des_block ****deskey***);**

int key_gendes(des_block ****deskey*);**

int key_setsecret(char ****key*);**
int key_secretkey_is_set(void);

DESCRIPTION

The functions here are used within the RPC's secure authentication mechanism (AUTH_DES). There should be no need for user programs to use this functions.

The function **key_decryptsession**() uses the (remote) server netname and takes the DES key for decrypting. It uses the public key of the server and the secret key associated with the effective UID of the calling process.

The function **key_encryptsession**() is the inverse of **key_decryptsession**(). It encrypts the DES keys with the public key of the server and the secret key associated with the effective UID of the calling process.

The function **key_gendes**() is used to ask the keyserver for a secure conversation key.

The function **key_setsecret**() is used to set the key for the effective UID of the calling process.

The function **key_secretkey_is_set**() can be used to determine whether a key has been set for the effective UID of the calling process.

RETURN VALUE

These functions return 1 on success and 0 on failure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
key_decryptsession(), **key_encryptsession**(), **key_gendes**(), **key_setsecret**(), **key_secretkey_is_set**()	Thread safety	MT-Safe

NOTES

Note that we talk about two types of encryption here. One is asymmetric using a public and secret key. The other is symmetric, the 64-bit DES.

These routines were part of the Linux/Doors-project, abandoned by now.

SEE ALSO

crypt(3)

NAME

killpg − send signal to a process group

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int killpg(int pgrp, int sig);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

killpg():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

killpg() sends the signal *sig* to the process group *pgrp*. See *signal*(7) for a list of signals.

If *pgrp* is 0, **killpg**() sends the signal to the calling process's process group. (POSIX says: if *pgrp* is less than or equal to 1, the behavior is undefined.)

For the permissions required to send a signal to another process, see *kill*(2).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
sig is not a valid signal number.

EPERM
The process does not have permission to send the signal to any of the target processes. For the required permissions, see *kill*(2).

ESRCH
No process can be found in the process group specified by *pgrp*.

ESRCH
The process group was given as 0 but the sending process does not have a process group.

VERSIONS

There are various differences between the permission checking in BSD-type systems and System V-type systems. See the POSIX rationale for *kill*(3p)A difference not mentioned by POSIX concerns the return value **EPERM**: BSD documents that no signal is sent and **EPERM** returned when the permission check failed for at least one target process, while POSIX documents **EPERM** only when the permission check failed for all target processes.

C library/kernel differences

On Linux, **killpg**() is implemented as a library function that makes the call *kill(−pgrp, sig)*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.4BSD (first appeared in 4BSD).

SEE ALSO

getpgrp(2), *kill*(2), *signal*(2), *capabilities*(7), *credentials*(7)

NAME

ldexp, ldexpf, ldexpl − multiply floating-point number by integral power of 2

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ldexpf(), **ldexpl**():

```
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

These functions return the result of multiplying the floating-point number *x* by 2 raised to the power *exp*.

RETURN VALUE

On success, these functions return *x * (2^exp)*.

If *exp* is zero, then *x* is returned.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity (negative infinity), positive infinity (negative infinity) is returned.

If the result underflows, a range error occurs, and zero is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with a sign the same as *x*.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error, overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error, underflow
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ldexp(), **ldexpf**(), **ldexpl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

frexp(3), *modf*(3), *scalbln*(3)

NAME

lgamma, lgammaf, lgammal, lgamma_r, lgammaf_r, lgammal_r, signgam − log gamma function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);

double lgamma_r(double x, int *signp);
float lgammaf_r(float x, int *signp);
long double lgammal_r(long double x, int *signp);

extern int signgam;
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

lgamma():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

lgammaf(), **lgammal**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

lgamma_r(), **lgammaf_r**(), **lgammal_r**():
/* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

signgam:
_XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

For the definition of the Gamma function, see *tgamma*(3).

The **lgamma**(), **lgammaf**(), and **lgammal**() functions return the natural logarithm of the absolute value of the Gamma function. The sign of the Gamma function is returned in the external integer *signgam* declared in <*math.h*>. It is 1 when the Gamma function is positive or zero, −1 when it is negative.

Since using a constant location *signgam* is not thread-safe, the functions **lgamma_r**(), **lgammaf_r**(), and **lgammal_r**() have been introduced; they return the sign via the argument *signp*.

RETURN VALUE

On success, these functions return the natural logarithm of Gamma(x).

If *x* is a NaN, a NaN is returned.

If *x* is 1 or 2, +0 is returned.

If *x* is positive infinity or negative infinity, positive infinity is returned.

If *x* is a nonpositive integer, a pole error occurs, and the functions return +**HUGE_VAL**, +**HUGE_VALF**, or +**HUGE_VALL**, respectively.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the correct mathematical sign.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Pole error: *x* is a nonpositive integer
: *errno* is set to **ERANGE** (but see BUGS). A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

STANDARDS

lgamma()
lgammaf()
lgammal()
: C11, POSIX.1-2008.

signgam
: POSIX.1-2008.

lgamma_r()
lgammaf_r()
lgammal_r()
: None.

HISTORY

lgamma()
lgammaf()
lgammal()
: C99, POSIX.1-2001.

signgam
: POSIX.1-2001.

lgamma_r()
lgammaf_r()
lgammal_r()
: None.

BUGS

In glibc 2.9 and earlier, when a pole error occurs, *errno* is set to **EDOM**; instead of the POSIX-mandated **ERANGE**. Since glibc 2.10, glibc does the right thing.

SEE ALSO

tgamma(3)

NAME

lio_listio − initiate a list of I/O requests

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <aio.h>

int lio_listio(int mode,
               struct aiocb *restrict const aiocb_list[restrict],
               int nitems, struct sigevent *restrict sevp);
```

DESCRIPTION

The **lio_listio**() function initiates the list of I/O operations described by the array *aiocb_list*.

The *mode* operation has one of the following values:

LIO_WAIT
: The call blocks until all operations are complete. The *sevp* argument is ignored.

LIO_NOWAIT
: The I/O operations are queued for processing and the call returns immediately. When all of the I/O operations complete, asynchronous notification occurs, as specified by the *sevp* argument; see *sigevent*(3type) for details. If *sevp* is NULL, no asynchronous notification occurs.

The *aiocb_list* argument is an array of pointers to *aiocb* structures that describe I/O operations. These operations are executed in an unspecified order. The *nitems* argument specifies the size of the array *aiocb_list*. Null pointers in *aiocb_list* are ignored.

In each control block in *aiocb_list*, the *aio_lio_opcode* field specifies the I/O operation to be initiated, as follows:

LIO_READ
: Initiate a read operation. The operation is queued as for a call to *aio_read*(3) specifying this control block.

LIO_WRITE
: Initiate a write operation. The operation is queued as for a call to *aio_write*(3) specifying this control block.

LIO_NOP
: Ignore this control block.

The remaining fields in each control block have the same meanings as for *aio_read*(3) and *aio_write*(3). The *aio_sigevent* fields of each control block can be used to specify notifications for the individual I/O operations (see *sigevent*(7)).

RETURN VALUE

If *mode* is **LIO_NOWAIT**, **lio_listio**() returns 0 if all I/O operations are successfully queued. Otherwise, −1 is returned, and *errno* is set to indicate the error.

If *mode* is **LIO_WAIT**, **lio_listio**() returns 0 when all of the I/O operations have completed successfully. Otherwise, −1 is returned, and *errno* is set to indicate the error.

The return status from **lio_listio**() provides information only about the call itself, not about the individual I/O operations. One or more of the I/O operations may fail, but this does not prevent other operations completing. The status of individual I/O operations in *aiocb_list* can be determined using *aio_error*(3). When an operation has completed, its return status can be obtained using *aio_return*(3). Individual I/O operations can fail for the reasons described in *aio_read*(3) and *aio_write*(3).

ERRORS

The **lio_listio**() function may fail for the following reasons:

EAGAIN
: Out of resources.

EAGAIN
: The number of I/O operations specified by *nitems* would cause the limit **AIO_MAX** to be exceeded.

EINTR
mode was **LIO_WAIT** and a signal was caught before all I/O operations completed; see *signal*(7). (This may even be one of the signals used for asynchronous I/O completion notification.)

EINVAL
mode is invalid, or *nitems* exceeds the limit **AIO_LISTIO_MAX**.

EIO One of more of the operations specified by *aiocb_list* failed. The application can check the status of each operation using *aio_return*(3).

If **lio_listio**() fails with the error **EAGAIN**, **EINTR**, or **EIO**, then some of the operations in *aiocb_list* may have been initiated. If **lio_listio**() fails for any other reason, then none of the I/O operations has been initiated.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lio_listio()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

It is a good idea to zero out the control blocks before use. The control blocks must not be changed while the I/O operations are in progress. The buffer areas being read into or written from must not be accessed during the operations or undefined results may occur. The memory areas involved must remain valid.

Simultaneous I/O operations specifying the same *aiocb* structure produce undefined results.

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *aio*(7)

NAME

LIST_EMPTY, LIST_ENTRY, LIST_FIRST, LIST_FOREACH, LIST_HEAD, LIST_HEAD_INITIALIZER, LIST_INIT, LIST_INSERT_AFTER, LIST_INSERT_BEFORE, LIST_INSERT_HEAD, LIST_NEXT, LIST_REMOVE − implementation of a doubly linked list

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/queue.h>

LIST_ENTRY(TYPE);

LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD LIST_HEAD_INITIALIZER(LIST_HEAD *head***);**
void LIST_INIT(LIST_HEAD * *head***);**

int LIST_EMPTY(LIST_HEAD * *head***);**

void LIST_INSERT_HEAD(LIST_HEAD * *head***,**
struct TYPE * *elm***, LIST_ENTRY** *NAME***);**
void LIST_INSERT_BEFORE(struct TYPE * *listelm***,**
struct TYPE * *elm***, LIST_ENTRY** *NAME***);**
void LIST_INSERT_AFTER(struct TYPE * *listelm***,**
struct TYPE * *elm***, LIST_ENTRY** *NAME***);**

struct TYPE *LIST_FIRST(LIST_HEAD * *head***);**
struct TYPE *LIST_NEXT(struct TYPE * *elm***, LIST_ENTRY** *NAME***);**

LIST_FOREACH(struct TYPE * *var***, LIST_HEAD *** *head***, LIST_ENTRY** *NAME***);**

void LIST_REMOVE(struct TYPE * *elm***, LIST_ENTRY** *NAME***);**

DESCRIPTION

These macros define and operate on doubly linked lists.

In the macro definitions, *TYPE* is the name of a user-defined structure, that must contain a field of type *LIST_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user-defined structure that must be declared using the macro **LIST_HEAD**().

Creation

A list is headed by a structure defined by the **LIST_HEAD**() macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A *LIST_HEAD* structure is declared as follows:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

where *struct HEADNAME* is the structure to be defined, and *struct TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

LIST_ENTRY() declares a structure that connects the elements in the list.

LIST_HEAD_INITIALIZER() evaluates to an initializer for the list *head*.

LIST_INIT() initializes the list referenced by *head*.

LIST_EMPTY() evaluates to true if there are no elements in the list.

Insertion

LIST_INSERT_HEAD() inserts the new element *elm* at the head of the list.

LIST_INSERT_BEFORE() inserts the new element *elm* before the element *listelm*.

LIST_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

Traversal

LIST_FIRST() returns the first element in the list, or NULL if the list is empty.

LIST_NEXT() returns the next element in the list, or NULL if this is the last.

LIST_FOREACH() traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

Removal

LIST_REMOVE() removes the element *elm* from the list.

RETURN VALUE

LIST_EMPTY() returns nonzero if the list is empty, and zero if the list contains at least one entry.

LIST_FIRST(), and **LIST_NEXT**() return a pointer to the first or next *TYPE* structure, respectively.

LIST_HEAD_INITIALIZER() returns an initializer that can be assigned to the list *head*.

STANDARDS

BSD.

HISTORY

4.4BSD.

BUGS

LIST_FOREACH() doesn't allow *var* to be removed or freed within the loop, as it would interfere with the traversal. **LIST_FOREACH_SAFE**(), which is present on the BSDs but is not present in glibc, fixes this limitation by allowing *var* to safely be removed from the list and freed from within the loop without interfering with the traversal.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    LIST_ENTRY(entry) entries;              /* List */
};

LIST_HEAD(listhead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct listhead head;                   /* List head */
    int i;

    LIST_INIT(&head);                       /* Initialize the list */

    n1 = malloc(sizeof(struct entry));      /* Insert at the head */
    LIST_INSERT_HEAD(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));      /* Insert after */
    LIST_INSERT_AFTER(n1, n2, entries);

    n3 = malloc(sizeof(struct entry));      /* Insert before */
    LIST_INSERT_BEFORE(n2, n3, entries);

    i = 0;                                  /* Forward traversal */
    LIST_FOREACH(np, &head, entries)
        np->data = i++;

    LIST_REMOVE(n2, entries);               /* Deletion */
    free(n2);
                                            /* Forward traversal */
```

```
        LIST_FOREACH(np, &head, entries)
            printf("%i\n", np->data);
                                                /* List deletion */
        n1 = LIST_FIRST(&head);
        while (n1 != NULL) {
            n2 = LIST_NEXT(n1, entries);
            free(n1);
            n1 = n2;
        }
        LIST_INIT(&head);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

insque(3), *queue*(7)

NAME

localeconv − get numeric formatting information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <locale.h>

struct lconv *localeconv(void);

DESCRIPTION

The **localeconv**() function returns a pointer to a *struct lconv* for the current locale. This structure is shown in *locale*(7), and contains all values associated with the locale categories **LC_NUMERIC** and **LC_MONETARY**. Programs may also use the functions *printf*(3) and *strfmon*(3), which behave according to the actual locale in use.

RETURN VALUE

The **localeconv**() function returns a pointer to a filled in *struct lconv*. This structure may be (in glibc, *is*) statically allocated, and may be overwritten by subsequent calls. According to POSIX, the caller should not modify the contents of this structure. The **localeconv**() function always succeeds.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
localeconv()	Thread safety	MT-Unsafe race:localeconv locale

STANDARDS

C11.

HISTORY

C89.

BUGS

The *printf*(3) family of functions may or may not honor the current locale.

SEE ALSO

locale(1), *localedef*(1), *isalpha*(3), *nl_langinfo*(3), *setlocale*(3), *strcoll*(3), *strftime*(3), *locale*(7)

NAME

lockf − apply, test or remove a POSIX lock on an open file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

int lockf(int *fd***, int** *op***, off_t** *len***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

lockf():
_XOPEN_SOURCE >= 500
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Apply, test, or remove a POSIX lock on a section of an open file. The file is specified by *fd*, a file descriptor open for writing, the action by *op*, and the section consists of byte positions *pos*..*pos*+*len*−1 if *len* is positive, and *pos*−*len*..*pos*−1 if *len* is negative, where *pos* is the current file position, and if *len* is zero, the section extends from the current file position to infinity, encompassing the present and future end-of-file positions. In all cases, the section may extend past current end-of-file.

On Linux, **lockf**() is just an interface on top of *fcntl*(2) locking. Many other systems implement **lockf**() in this way, but note that POSIX.1 leaves the relationship between **lockf**() and *fcntl*(2) locks unspecified. A portable application should probably avoid mixing calls to these interfaces.

Valid operations are given below:

F_LOCK
: Set an exclusive lock on the specified section of the file. If (part of) this section is already locked, the call blocks until the previous lock is released. If this section overlaps an earlier locked section, both are merged. File locks are released as soon as the process holding the locks closes some file descriptor for the file. A child process does not inherit these locks.

F_TLOCK
: Same as **F_LOCK** but the call never blocks and returns an error instead if the file is already locked.

F_ULOCK
: Unlock the indicated section of the file. This may cause a locked section to be split into two locked sections.

F_TEST
: Test the lock: return 0 if the specified section is unlocked or locked by this process; return −1, set *errno* to **EAGAIN** (**EACCES** on some other systems), if another process holds a lock.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES or **EAGAIN**
: The file is locked and **F_TLOCK** or **F_TEST** was specified, or the operation is prohibited because the file has been memory-mapped by another process.

EBADF
: *fd* is not an open file descriptor; or *op* is **F_LOCK** or **F_TLOCK** and *fd* is not a writable file descriptor.

EDEADLK
: *op* was **F_LOCK** and this lock operation would cause a deadlock.

EINTR
: While waiting to acquire a lock, the call was interrupted by delivery of a signal caught by a handler; see *signal*(7).

EINVAL
An invalid operation was specified in *op*.

ENOLCK
Too many segment locks open, lock table is full.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lockf()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4.

SEE ALSO

fcntl(2), *flock*(2)

locks.txt and *mandatory−locking.txt* in the Linux kernel source directory *Documentation/filesystems* (on older kernels, these files are directly under the *Documentation* directory, and *mandatory−locking.txt* is called *mandatory.txt*)

NAME

log, logf, logl − natural logarithmic function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double log(double x);
float logf(float x);
long double logl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

logf(), **logl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the natural logarithm of *x*.

RETURN VALUE

On success, these functions return the natural logarithm of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is 1, the result is +0.

If *x* is positive infinity, positive infinity is returned.

If *x* is zero, then a pole error occurs, and the functions return **−HUGE_VAL**, **−HUGE_VALF**, or **−HUGE_VALL**, respectively.

If *x* is negative (including negative infinity), then a domain error occurs, and a NaN (not a number) is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is negative
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

Pole error: *x* is zero
: *errno* is set to **ERANGE**. A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
log(), **logf**(), **logl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

BUGS

In glibc 2.5 and earlier, taking the **log**() of a NaN produces a bogus invalid floating-point (**FE_INVALID**) exception.

SEE ALSO

cbrt(3), *clog*(3), *log10*(3), *log1p*(3), *log2*(3), *sqrt*(3)

NAME

log2, log2f, log2l − base-2 logarithmic function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

log2(), **log2f**(), **log2l**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return the base-2 logarithm of *x*.

RETURN VALUE

On success, these functions return the base-2 logarithm of *x*.

For special cases, including where *x* is 0, 1, negative, infinity, or NaN, see *log*(3).

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

For a discussion of the errors that can occur for these functions, see *log*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
log2(), **log2f**(), **log2l**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD.

SEE ALSO

cbrt(3), *clog2*(3), *log*(3), *log10*(3), *sqrt*(3)

NAME

log10, log10f, log10l − base-10 logarithmic function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

log10f(), **log10l**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the base-10 logarithm of *x*.

RETURN VALUE

On success, these functions return the base-10 logarithm of *x*.

For special cases, including where *x* is 0, 1, negative, infinity, or NaN, see *log*(3).

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

For a discussion of the errors that can occur for these functions, see *log*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
log10(), **log10f**(), **log10l**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

cbrt(3), *clog10*(3), *exp10*(3), *log*(3), *log2*(3), *sqrt*(3)

NAME

log1p, log1pf, log1pl − logarithm of 1 plus argument

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double log1p(double *x***);**
float log1pf(float *x***);**
long double log1pl(long double *x***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

log1p():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

log1pf(), **log1pl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return a value equivalent to

log (1 + *x*)

The result is computed in a way that is accurate even if the value of *x* is near zero.

RETURN VALUE

On success, these functions return the natural logarithm of *(1 + x)*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is −1, a pole error occurs, and the functions return −**HUGE_VAL**, −**HUGE_VALF**, or −**HUGE_VALL**, respectively.

If *x* is less than −1 (including negative infinity), a domain error occurs, and a NaN (not a number) is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is less than −1
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

Pole error: *x* is −1
: *errno* is set to **ERANGE** (but see BUGS). A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
log1p(), **log1pf**(), **log1pl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

BUGS

Before glibc 2.22, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred.

Before glibc 2.22, the glibc implementation did not set *errno* to **ERANGE** when a range error occurred.

SEE ALSO

exp(3), *expm1*(3), *log*(3)

NAME

logb, logbf, logbl − get exponent of a floating-point value

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

logb():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

logbf(), **logbl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions extract the exponent from the internal floating-point representation of *x* and return it as a floating-point value. The integer constant **FLT_RADIX**, defined in <*float.h*>, indicates the radix used for the system's floating-point representation. If **FLT_RADIX** is 2, **logb(***x***)** is similar to **floor(log2(fabs(***x***)))**, except that the latter may give an incorrect integer due to intermediate rounding.

If *x* is subnormal, **logb**() returns the exponent *x* would have if it were normalized.

RETURN VALUE

On success, these functions return the exponent of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is zero, then a pole error occurs, and the functions return **−HUGE_VAL**, **−HUGE_VALF**, or **−HUGE_VALL**, respectively.

If *x* is negative infinity or positive infinity, then positive infinity is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Pole error: *x* is 0
A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
logb(), **logbf**(), **logbl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

logb() 4.3BSD (see IEEE.3 in the 4.3BSD manual).

SEE ALSO

ilogb(3), *log*(3)

NAME

login, logout − write utmp and wtmp entries

LIBRARY

System utilities library (*libutil*, *−lutil*)

SYNOPSIS

#include <utmp.h>

void login(const struct utmp ****ut*);**
int logout(const char ****ut_line*);**

DESCRIPTION

The utmp file records who is currently using the system. The wtmp file records all logins and logouts. See *utmp*(5).

The function **login**() takes the supplied *struct utmp*, *ut*, and writes it to both the utmp and the wtmp file.

The function **logout**() clears the entry in the utmp file again.

GNU details

More precisely, **login**() takes the argument *ut* struct, fills the field *ut−>ut_type* (if there is such a field) with the value **USER_PROCESS**, and fills the field *ut−>ut_pid* (if there is such a field) with the process ID of the calling process. Then it tries to fill the field *ut−>ut_line*. It takes the first of *stdin*, *stdout*, *stderr* that is a terminal, and stores the corresponding pathname minus a possible leading */dev/* into this field, and then writes the struct to the utmp file. On the other hand, if no terminal name was found, this field is filled with "???" and the struct is not written to the utmp file. After this, the struct is written to the wtmp file.

The **logout**() function searches the utmp file for an entry matching the *ut_line* argument. If a record is found, it is updated by zeroing out the *ut_name* and *ut_host* fields, updating the *ut_tv* timestamp field and setting *ut_type* (if there is such a field) to **DEAD_PROCESS**.

RETURN VALUE

The **logout**() function returns 1 if the entry was successfully written to the database, or 0 if an error occurred.

FILES

/var/run/utmp
: user accounting database, configured through **_PATH_UTMP** in <*paths.h*>

/var/log/wtmp
: user accounting log file, configured through **_PATH_WTMP** in <*paths.h*>

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
login(), **logout**()	Thread safety	MT-Unsafe race:utent sig:ALRM timer

In the above table, *utent* in *race:utent* signifies that if any of the functions *setutent*(3), *getutent*(3), or *endutent*(3) are used in parallel in different threads of a program, then data races could occur. **login**() and **logout**() calls those functions, so we use race:utent to remind users.

VERSIONS

The member *ut_user* of *struct utmp* is called *ut_name* in BSD. Therefore, *ut_name* is defined as an alias for *ut_user* in <*utmp.h*>.

STANDARDS

BSD.

SEE ALSO

getutent(3), *utmp*(5)

NAME

lrint, lrintf, lrintl, llrint, llrintf, llrintl − round to nearest integer

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions round their argument to the nearest integer value, using the current rounding direction (see *fesetround*(3)).

Note that unlike the *rint*(3) family of functions, the return type of these functions differs from that of their arguments.

RETURN VALUE

These functions return the rounded integer value.

If *x* is a NaN or an infinity, or the rounded value is too large to be stored in a *long* (*long long* in the case of the **ll*** functions), then a domain error occurs, and the return value is unspecified.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is a NaN or infinite, or the rounded value is too large
An invalid floating-point exception (**FE_INVALID**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lrint(), **lrintf**(), **lrintl**(), **llrint**(), **llrintf**(), **llrintl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

ceil(3), *floor*(3), *lround*(3), *nearbyint*(3), *rint*(3), *round*(3)

NAME

lround, lroundf, lroundl, llround, llroundf, llroundl − round to nearest integer

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction (see *fenv*(3)).

Note that unlike the *round*(3) and *ceil*(3), functions, the return type of these functions differs from that of their arguments.

RETURN VALUE

These functions return the rounded integer value.

If *x* is a NaN or an infinity, or the rounded value is too large to be stored in a *long* (*long long* in the case of the **ll*** functions), then a domain error occurs, and the return value is unspecified.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is a NaN or infinite, or the rounded value is too large
An invalid floating-point exception (**FE_INVALID**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lround(), **lroundf**(), **lroundl**(), **llround**(), **llroundf**(), **llroundl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

ceil(3), *floor*(3), *lrint*(3), *nearbyint*(3), *rint*(3), *round*(3)

NAME

lfind, lsearch − linear search of an array

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <search.h>

void *lfind(const void key[.size], const void base[.size * .nmemb],
        size_t *nmemb, size_t size,
        int(*compar)(const void [.size], const void [.size]));
void *lsearch(const void key[.size], void base[.size * .nmemb],
        size_t *nmemb, size_t size,
        int(*compar)(const void [.size], const void [.size]));
```

DESCRIPTION

lfind() and **lsearch**() perform a linear search for *key* in the array *base* which has **nmemb* elements of *size* bytes each. The comparison function referenced by *compar* is expected to have two arguments which point to the *key* object and to an array member, in that order, and which returns zero if the *key* object matches the array member, and nonzero otherwise.

If **lsearch**() does not find a matching element, then the *key* object is inserted at the end of the table, and **nmemb* is incremented. In particular, one should know that a matching element exists, or that more room is available.

RETURN VALUE

lfind() returns a pointer to a matching member of the array, or NULL if no match is found. **lsearch**() returns a pointer to a matching member of the array, or to the newly added member if no match is found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lfind(), **lsearch**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD. libc-4.6.27.

BUGS

The naming is unfortunate.

SEE ALSO

bsearch(3), *hsearch*(3), *tsearch*(3)

NAME

lseek64 − reposition 64-bit read/write file offset

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _LARGEFILE64_SOURCE     /* See feature_test_macros(7) */
#include <sys/types.h>
#include <unistd.h>

off64_t lseek64(int fd, off64_t offset, int whence);
```

DESCRIPTION

The **lseek**() family of functions reposition the offset of the open file associated with the file descriptor *fd* to *offset* bytes relative to the start, current position, or end of the file, when *whence* has the value **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, respectively.

For more details, return value, and errors, see *lseek*(2).

Four interfaces are available: **lseek**(), **lseek64**(), **llseek**(), and **_llseek**().

lseek()

Prototype:

```
off_t lseek(int fd, off_t offset, int whence);
```

The C library's **lseek**() wrapper function uses the type *off_t*. This is a 32-bit signed type on 32-bit architectures, unless one compiles with

```
#define _FILE_OFFSET_BITS 64
```

in which case it is a 64-bit signed type.

lseek64()

Prototype:

```
off64_t lseek64(int fd, off64_t offset, int whence);
```

The **lseek64**() library function uses a 64-bit type even when *off_t* is a 32-bit type. Its prototype (and the type *off64_t*) is available only when one compiles with

```
#define _LARGEFILE64_SOURCE
```

The function **lseek64**() is available since glibc 2.1.

llseek()

Prototype:

```
loff_t llseek(int fd, loff_t offset, int whence);
```

The type *loff_t* is a 64-bit signed type. The **llseek**() library function is available in glibc and works without special defines. However, the glibc headers do not provide a prototype. Users should add the above prototype, or something equivalent, to their own source. When users complained about data loss caused by a miscompilation of *e2fsck*(8), glibc 2.1.3 added the link-time warning

"the `llseek´ function may be dangerous; use `lseek64´ instead."

This makes this function unusable if one desires a warning-free compilation.

Since glibc 2.28, this function symbol is no longer available to newly linked applications.

_llseek()

On 32-bit architectures, this is the system call that is used (by the C library wrapper functions) to implement all of the above functions. The prototype is:

```
int _llseek(int fd, off_t offset_hi, off_t offset_lo,
            loff_t *result, int whence);
```

For more details, see *llseek*(2).

64-bit systems don't need an **_llseek**() system call. Instead, they have an *lseek*(2) system call that supports 64-bit file offsets.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
lseek64()	Thread safety	MT-Safe

NOTES

lseek64() is one of the functions that was specified in the Large File Summit (LFS) specification that was completed in 1996. The purpose of the specification was to provide transitional support that allowed applications on 32-bit systems to access files whose size exceeds that which can be represented with a 32-bit *off_t* type. As noted above, this symbol is exposed by header files if the **_LARGEFILE64_SOURCE** feature test macro is defined. ALternatively, on a 32-bit system, the symbol *lseek* is aliased to *lseek64* if the macro **_FILE_OFFSET_BITS** is defined with the value 64.

SEE ALSO

llseek(2), *lseek*(2)

NAME

makecontext, swapcontext − manipulate user context

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <ucontext.h>

void makecontext(ucontext_t ****ucp*, void (*** *func***)(), int** *argc***, ...);**
int swapcontext(ucontext_t *restrict *oucp***,**
const ucontext_t *restrict *ucp***);**

DESCRIPTION

In a System V-like environment, one has the type *ucontext_t* (defined in <*ucontext.h*> and described in *getcontext*(3)) and the four functions *getcontext*(3), *setcontext*(3), **makecontext**(), and **swapcontext**() that allow user-level context switching between multiple threads of control within a process.

The **makecontext**() function modifies the context pointed to by *ucp* (which was obtained from a call to *getcontext*(3)). Before invoking **makecontext**(), the caller must allocate a new stack for this context and assign its address to *ucp−>uc_stack*, and define a successor context and assign its address to *ucp−>uc_link*.

When this context is later activated (using *setcontext*(3) or *swapcontext*()) the function *func* is called, and passed the series of integer (*int*) arguments that follow *argc*; the caller must specify the number of these arguments in *argc*. When this function returns, the successor context is activated. If the successor context pointer is NULL, the thread exits.

The **swapcontext**() function saves the current context in the structure pointed to by *oucp*, and then activates the context pointed to by *ucp*.

RETURN VALUE

When successful, **swapcontext**() does not return. (But we may return later, in case *oucp* is activated, in which case it looks like **swapcontext**() returns 0.) On error, **swapcontext**() returns −1 and sets *errno* to indicate the error.

ERRORS

ENOMEM
Insufficient stack space left.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
makecontext()	Thread safety	MT-Safe race:ucp
swapcontext()	Thread safety	MT-Safe race:oucp race:ucp

STANDARDS

None.

HISTORY

glibc 2.1. SUSv2, POSIX.1-2001. Removed in POSIX.1-2008, citing portability issues, and recommending that applications be rewritten to use POSIX threads instead.

NOTES

The interpretation of *ucp−>uc_stack* is just as in *sigaltstack*(2), namely, this struct contains the start and length of a memory area to be used as the stack, regardless of the direction of growth of the stack. Thus, it is not necessary for the user program to worry about this direction.

On architectures where *int* and pointer types are the same size (e.g., x86-32, where both types are 32 bits), you may be able to get away with passing pointers as arguments to **makecontext**() following *argc*. However, doing this is not guaranteed to be portable, is undefined according to the standards, and won't work on architectures where pointers are larger than *int*s. Nevertheless, starting with glibc 2.8, glibc makes some changes to **makecontext**(), to permit this on some 64-bit architectures (e.g., x86-64).

EXAMPLES

The example program below demonstrates the use of *getcontext*(3), **makecontext**(), and **swapcontext**(). Running the program produces the following output:

```
$ ./a.out
main: swapcontext(&uctx_main, &uctx_func2)
func2: started
func2: swapcontext(&uctx_func2, &uctx_func1)
func1: started
func1: swapcontext(&uctx_func1, &uctx_func2)
func2: returning
func1: returning
main: exiting
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>

static ucontext_t uctx_main, uctx_func1, uctx_func2;

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void
func1(void)
{
    printf("%s: started\n", __func__);
    printf("%s: swapcontext(&uctx_func1, &uctx_func2)\n", __func__);
    if (swapcontext(&uctx_func1, &uctx_func2) == -1)
        handle_error("swapcontext");
    printf("%s: returning\n", __func__);
}

static void
func2(void)
{
    printf("%s: started\n", __func__);
    printf("%s: swapcontext(&uctx_func2, &uctx_func1)\n", __func__);
    if (swapcontext(&uctx_func2, &uctx_func1) == -1)
        handle_error("swapcontext");
    printf("%s: returning\n", __func__);
}

int
main(int argc, char *argv[])
{
    char func1_stack[16384];
    char func2_stack[16384];

    if (getcontext(&uctx_func1) == -1)
        handle_error("getcontext");
    uctx_func1.uc_stack.ss_sp = func1_stack;
    uctx_func1.uc_stack.ss_size = sizeof(func1_stack);
    uctx_func1.uc_link = &uctx_main;
    makecontext(&uctx_func1, func1, 0);

    if (getcontext(&uctx_func2) == -1)
        handle_error("getcontext");
    uctx_func2.uc_stack.ss_sp = func2_stack;
    uctx_func2.uc_stack.ss_size = sizeof(func2_stack);
    /* Successor context is f1(), unless argc > 1 */
```

```
        uctx_func2.uc_link = (argc > 1) ? NULL : &uctx_func1;
        makecontext(&uctx_func2, func2, 0);

        printf("%s: swapcontext(&uctx_main, &uctx_func2)\n", __func__);
        if (swapcontext(&uctx_main, &uctx_func2) == -1)
            handle_error("swapcontext");

        printf("%s: exiting\n", __func__);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

sigaction(2), *sigaltstack*(2), *sigprocmask*(2), *getcontext*(3), *sigsetjmp*(3)

NAME

makedev, major, minor − manage a device number

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/sysmacros.h>

dev_t makedev(unsigned int *maj***, unsigned int** *min***);**

unsigned int major(dev_t *dev***);**
unsigned int minor(dev_t *dev***);**

DESCRIPTION

A device ID consists of two parts: a major ID, identifying the class of the device, and a minor ID, identifying a specific instance of a device in that class. A device ID is represented using the type *dev_t*.

Given major and minor device IDs, **makedev**() combines these to produce a device ID, returned as the function result. This device ID can be given to *mknod*(2), for example.

The **major**() and **minor**() functions perform the converse task: given a device ID, they return, respectively, the major and minor components. These macros can be useful to, for example, decompose the device IDs in the structure returned by *stat*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
makedev(), **major**(), **minor**()	Thread safety	MT-Safe

VERSIONS

The BSDs expose the definitions for these macros via <*sys/types.h*>.

STANDARDS

None.

HISTORY

BSD, HP-UX, Solaris, AIX, Irix.

These interfaces are defined as macros. Since glibc 2.3.3, they have been aliases for three GNU-specific functions: **gnu_dev_makedev**(), **gnu_dev_major**(), and **gnu_dev_minor**(). The latter names are exported, but the traditional names are more portable.

Depending on the version, glibc also exposes definitions for these macros from <*sys/types.h*> if suitable feature test macros are defined. However, this behavior was deprecated in glibc 2.25, and since glibc 2.28, <*sys/types.h*> no longer provides these definitions.

SEE ALSO

mknod(2), *stat*(2)

NAME

mallinfo, mallinfo2 − obtain memory allocation information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

struct mallinfo mallinfo(void);
struct mallinfo2 mallinfo2(void);

DESCRIPTION

These functions return a copy of a structure containing information about memory allocations performed by *malloc*(3) and related functions. The structure returned by each function contains the same fields. However, the older function, **mallinfo**(), is deprecated since the type used for the fields is too small (see BUGS).

Note that not all allocations are visible to these functions; see BUGS and consider using *malloc_info*(3) instead.

The *mallinfo2* structure returned by **mallinfo2**() is defined as follows:

```
struct mallinfo2 {
    size_t arena;     /* Non-mmapped space allocated (bytes) */
    size_t ordblks;   /* Number of free chunks */
    size_t smblks;    /* Number of free fastbin blocks */
    size_t hblks;     /* Number of mmapped regions */
    size_t hblkhd;    /* Space allocated in mmapped regions
                         (bytes) */
    size_t usmblks;   /* See below */
    size_t fsmblks;   /* Space in freed fastbin blocks (bytes) */
    size_t uordblks;  /* Total allocated space (bytes) */
    size_t fordblks;  /* Total free space (bytes) */
    size_t keepcost;  /* Top-most, releasable space (bytes) */
};
```

The *mallinfo* structure returned by the deprecated **mallinfo**() function is exactly the same, except that the fields are typed as *int*.

The structure fields contain the following information:

arena The total amount of memory allocated by means other than *mmap*(2) (i.e., memory allocated on the heap). This figure includes both in-use blocks and blocks on the free list.

ordblks The number of ordinary (i.e., non-fastbin) free blocks.

smblks The number of fastbin free blocks (see *mallopt*(3)).

hblks The number of blocks currently allocated using *mmap*(2). (See the discussion of **M_MMAP_THRESHOLD** in *mallopt*(3).)

hblkhd The number of bytes in blocks currently allocated using *mmap*(2).

usmblks This field is unused, and is always 0. Historically, it was the "highwater mark" for allocated space—that is, the maximum amount of space that was ever allocated (in bytes); this field was maintained only in nonthreading environments.

fsmblks The total number of bytes in fastbin free blocks.

uordblks The total number of bytes used by in-use allocations.

fordblks The total number of bytes in free blocks.

keepcost The total amount of releasable free space at the top of the heap. This is the maximum number of bytes that could ideally (i.e., ignoring page alignment restrictions, and so on) be released by *malloc_trim*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mallinfo(), **mallinfo2**()	Thread safety	MT-Unsafe init const:mallopt

mallinfo()/ **mallinfo2**() would access some global internal objects. If modify them with non-atomically, may get inconsistent results. The identifier *mallopt* in *const:mallopt* mean that **mallopt**() would modify the global internal objects with atomics, that make sure **mallinfo**()/ **mallinfo2**() is safe enough, others modify with non-atomically maybe not.

STANDARDS

None.

HISTORY

mallinfo()

glibc 2.0. SVID.

mallinfo2()

glibc 2.33.

BUGS

Information is returned for only the main memory allocation area. Allocations in other arenas are excluded. See *malloc_stats*(3) and *malloc_info*(3) for alternatives that include information about other arenas.

The fields of the *mallinfo* structure that is returned by the older **mallinfo**() function are typed as *int*. However, because some internal bookkeeping values may be of type *long*, the reported values may wrap around zero and thus be inaccurate.

EXAMPLES

The program below employs **mallinfo2**() to retrieve memory allocation statistics before and after allocating and freeing some blocks of memory. The statistics are displayed on standard output.

The first two command-line arguments specify the number and size of blocks to be allocated with *malloc*(3).

The remaining three arguments specify which of the allocated blocks should be freed with *free*(3). These three arguments are optional, and specify (in order): the step size to be used in the loop that frees blocks (the default is 1, meaning free all blocks in the range); the ordinal position of the first block to be freed (default 0, meaning the first allocated block); and a number one greater than the ordinal position of the last block to be freed (default is one greater than the maximum block number). If these three arguments are omitted, then the defaults cause all allocated blocks to be freed.

In the following example run of the program, 1000 allocations of 100 bytes are performed, and then every second allocated block is freed:

```
$ ./a.out 1000 100 2
============== Before allocating blocks ==============
Total non-mmapped bytes (arena):       0
# of free chunks (ordblks):            1
# of free fastbin blocks (smblks):     0
# of mapped regions (hblks):           0
Bytes in mapped regions (hblkhd):      0
Max. total allocated space (usmblks):  0
Free bytes held in fastbins (fsmblks): 0
Total allocated space (uordblks):      0
Total free space (fordblks):           0
Topmost releasable block (keepcost):   0

============== After allocating blocks ==============
Total non-mmapped bytes (arena):       135168
# of free chunks (ordblks):            1
# of free fastbin blocks (smblks):     0
# of mapped regions (hblks):           0
Bytes in mapped regions (hblkhd):      0
Max. total allocated space (usmblks):  0
Free bytes held in fastbins (fsmblks): 0
```

```
    Total allocated space (uordblks):      104000
    Total free space (fordblks):           31168
    Topmost releasable block (keepcost):   31168

    ============== After freeing blocks ==============
    Total non-mmapped bytes (arena):       135168
    # of free chunks (ordblks):            501
    # of free fastbin blocks (smblks):     0
    # of mapped regions (hblks):           0
    Bytes in mapped regions (hblkhd):      0
    Max. total allocated space (usmblks):  0
    Free bytes held in fastbins (fsmblks): 0
    Total allocated space (uordblks):      52000
    Total free space (fordblks):           83168
    Topmost releasable block (keepcost):   31168
```

Program source

```
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

static void
display_mallinfo2(void)
{
    struct mallinfo2 mi;

    mi = mallinfo2();

    printf("Total non-mmapped bytes (arena):       %zu\n", mi.arena);
    printf("# of free chunks (ordblks):            %zu\n", mi.ordblks);
    printf("# of free fastbin blocks (smblks):     %zu\n", mi.smblks);
    printf("# of mapped regions (hblks):           %zu\n", mi.hblks);
    printf("Bytes in mapped regions (hblkhd):      %zu\n", mi.hblkhd);
    printf("Max. total allocated space (usmblks):  %zu\n", mi.usmblks);
    printf("Free bytes held in fastbins (fsmblks): %zu\n", mi.fsmblks);
    printf("Total allocated space (uordblks):      %zu\n", mi.uordblks);
    printf("Total free space (fordblks):           %zu\n", mi.fordblks);
    printf("Topmost releasable block (keepcost):   %zu\n", mi.keepcost);
}

int
main(int argc, char *argv[])
{
#define MAX_ALLOCS 2000000
    char *alloc[MAX_ALLOCS];
    size_t blockSize, numBlocks, freeBegin, freeEnd, freeStep;

    if (argc < 3 || strcmp(argv[1], "--help") == 0) {
        fprintf(stderr, "%s num-blocks block-size [free-step "
                "[start-free [end-free]]]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    numBlocks = atoi(argv[1]);
    blockSize = atoi(argv[2]);
    freeStep = (argc > 3) ? atoi(argv[3]) : 1;
    freeBegin = (argc > 4) ? atoi(argv[4]) : 0;
    freeEnd = (argc > 5) ? atoi(argv[5]) : numBlocks;
```

```
        printf("============== Before allocating blocks ==============\n");
        display_mallinfo2();

        for (size_t j = 0; j < numBlocks; j++) {
            if (numBlocks >= MAX_ALLOCS) {
                fprintf(stderr, "Too many allocations\n");
                exit(EXIT_FAILURE);
            }

            alloc[j] = malloc(blockSize);
            if (alloc[j] == NULL) {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
        }

        printf("\n============== After allocating blocks ==============\n");
        display_mallinfo2();

        for (size_t j = freeBegin; j < freeEnd; j += freeStep)
            free(alloc[j]);

        printf("\n============== After freeing blocks ==============\n");
        display_mallinfo2();

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

mmap(2), *malloc*(3), *malloc_info*(3), *malloc_stats*(3), *malloc_trim*(3), *mallopt*(3)

NAME

malloc, free, calloc, realloc, reallocarray − allocate and free dynamic memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *_Nullable ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *_Nullable ptr, size_t size);
void *reallocarray(void *_Nullable ptr, size_t nmemb, size_t size);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

reallocarray():
Since glibc 2.29:
_DEFAULT_SOURCE
glibc 2.28 and earlier:
_GNU_SOURCE

DESCRIPTION

malloc()

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns a unique pointer value that can later be successfully passed to **free**(). (See "Nonportable behavior" for portability issues.)

free()

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**() or related functions. Otherwise, or if *ptr* has already been freed, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

calloc()

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns a unique pointer value that can later be successfully passed to **free**().

If the multiplication of *nmemb* and *size* would result in integer overflow, then **calloc**() returns an error. By contrast, an integer overflow would not be detected in the following call to **malloc**(), with the result that an incorrectly sized block of memory would be allocated:

```
malloc(nmemb * size);
```

realloc()

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents of the memory will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized.

If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*.

If *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)* (but see "Nonportable behavior" for portability issues).

Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc** or related functions. If the area pointed to was moved, a *free(ptr)* is done.

reallocarray()

The **reallocarray**() function changes the size of (and possibly moves) the memory block pointed to by *ptr* to be large enough for an array of *nmemb* elements, each of which is *size* bytes. It is equivalent to the call

```
realloc(ptr, nmemb * size);
```

However, unlike that **realloc**() call, **reallocarray**() fails safely in the case where the multiplication would overflow. If such an overflow occurs, **reallocarray**() returns an error.

RETURN VALUE

The **malloc**(), **calloc**(), **realloc**(), and **reallocarray**() functions return a pointer to the allocated memory, which is suitably aligned for any type that fits into the requested size or less. On error, these functions return NULL and set *errno*. Attempting to allocate more than **PTRDIFF_MAX** bytes is considered an error, as an object that large could cause later pointer subtraction to overflow.

The **free**() function returns no value, and preserves *errno*.

The **realloc**() and **reallocarray**() functions return NULL if *ptr* is not NULL and the requested size is zero; this is not considered an error. (See "Nonportable behavior" for portability issues.) Otherwise, the returned pointer may be the same as *ptr* if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from *ptr* if the allocation was moved to a new address. If these functions fail, the original block is left untouched; it is not freed or moved.

ERRORS

calloc(), **malloc**(), **realloc**(), and **reallocarray**() can fail with the following error:

ENOMEM
Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in *getrlimit*(2). Another reason could be that the number of mappings created by the caller process exceeded the limit specified by */proc/sys/vm/max_map_count*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc(), **free**(), **calloc**(), **realloc**()	Thread safety	MT-Safe

STANDARDS

malloc()
free()
calloc()
realloc()
C11, POSIX.1-2008.

reallocarray()
None.

HISTORY

malloc()
free()
calloc()
realloc()
POSIX.1-2001, C89.

reallocarray()
glibc 2.26. OpenBSD 5.6, FreeBSD 11.0.

malloc() and related functions rejected sizes greater than **PTRDIFF_MAX** starting in glibc 2.30.

free() preserved *errno* starting in glibc 2.33.

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of */proc/sys/vm/overcommit_memory* and */proc/sys/vm/oom_adj* in *proc*(5), and the Linux kernel source file *Documentation/vm/overcommit−accounting.rst*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, using *sbrk*(2). When allocating blocks of memory larger than **MMAP_THRESHOLD** bytes, the glibc **malloc**() implementation allocates the memory as a private anonymous mapping using *mmap*(2). **MMAP_THRESHOLD** is 128 kB by default, but is adjustable using *mallopt*(3). Prior to Linux 4.7 allocations performed using *mmap*(2) were unaffected by the **RLIMIT_DATA** resource limit; since Linux 4.7, this limit is also enforced for allocations performed using *mmap*(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which

threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using *brk*(2) or *mmap*(2)), and managed with its own mutexes.

If your program uses a private memory allocator, it should do so by replacing **malloc**(), **free**(), **calloc**(), and **realloc**(). The replacement functions must implement the documented glibc behaviors, including *errno* handling, size-zero allocations, and overflow checking; otherwise, other library routines may crash or operate incorrectly. For example, if the replacement *free*() does not preserve *errno*, then seemingly unrelated library routines may fail without having a valid reason in *errno*. Private memory allocators may also need to replace other glibc functions; see "Replacing malloc" in the glibc manual for details.

Crashes in memory allocators are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see *mallopt*(3) for details.

Nonportable behavior

The behavior of these functions when the requested size is zero is glibc specific; other implementations may return NULL without setting *errno*, and portable POSIX programs should tolerate such behavior. See *realloc*(3p)

POSIX requires memory allocators to set *errno* upon failure. However, the C standard does not require this, and applications portable to non-POSIX platforms should not assume this.

Portable programs should not use private memory allocators, as POSIX and the C standard do not allow replacement of **malloc**(), **free**(), **calloc**(), and **realloc**().

EXAMPLES

```
#include <err.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOCARRAY(n, type)  ((type *) my_mallocarray(n, sizeof(type)))
#define MALLOC(type)          MALLOCARRAY(1, type)

static inline void *my_mallocarray(size_t nmemb, size_t size);

int
main(void)
{
    char  *p;

    p = MALLOCARRAY(32, char);
    if (p == NULL)
        err(EXIT_FAILURE, "malloc");

    strlcpy(p, "foo", 32);
    puts(p);
}

static inline void *
my_mallocarray(size_t nmemb, size_t size)
{
    return reallocarray(NULL, nmemb, size);
}
```

SEE ALSO

valgrind(1), *brk*(2), *mmap*(2), *alloca*(3), *malloc_get_state*(3), *malloc_info*(3), *malloc_trim*(3), *malloc_usable_size*(3), *mallopt*(3), *mcheck*(3), *mtrace*(3), *posix_memalign*(3)

For details of the GNU C library implementation, see .

NAME

malloc_get_state, malloc_set_state − record and restore state of malloc implementation

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

void *malloc_get_state(void);
int malloc_set_state(void *_state_);

DESCRIPTION

Note: these functions are removed in glibc 2.25.

The **malloc_get_state**() function records the current state of all *malloc*(3) internal bookkeeping variables (but not the actual contents of the heap or the state of *malloc_hook*(3) functions pointers). The state is recorded in a system-dependent opaque data structure dynamically allocated via *malloc*(3), and a pointer to that data structure is returned as the function result. (It is the caller's responsibility to *free*(3) this memory.)

The **malloc_set_state**() function restores the state of all *malloc*(3) internal bookkeeping variables to the values recorded in the opaque data structure pointed to by *state*.

RETURN VALUE

On success, **malloc_get_state**() returns a pointer to a newly allocated opaque data structure. On error (for example, memory could not be allocated for the data structure), **malloc_get_state**() returns NULL.

On success, **malloc_set_state**() returns 0. If the implementation detects that *state* does not point to a correctly formed data structure, **malloc_set_state**() returns −1. If the implementation detects that the version of the data structure referred to by *state* is a more recent version than this implementation knows about, **malloc_set_state**() returns −2.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc_get_state(), **malloc_set_state**()	Thread safety	MT-Safe

STANDARDS

GNU.

NOTES

These functions are useful when using this *malloc*(3) implementation as part of a shared library, and the heap contents are saved/restored via some other method. This technique is used by GNU Emacs to implement its "dumping" function.

Hook function pointers are never saved or restored by these functions, with two exceptions: if malloc checking (see *mallopt*(3)) was in use when **malloc_get_state**() was called, then **malloc_set_state**() resets malloc checking hooks if possible; if malloc checking was not in use in the recorded state, but the caller has requested malloc checking, then the hooks are reset to 0.

SEE ALSO

malloc(3), *mallopt*(3)

NAME

__malloc_hook, __malloc_initialize_hook, __memalign_hook, __free_hook, __realloc_hook, __after_morecore_hook − malloc debugging variables (DEPRECATED)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <malloc.h>

void *(*volatile __malloc_hook)(size_t size, const void *caller);

void *(*volatile __realloc_hook)(void *ptr, size_t size,
                                 const void *caller);

void *(*volatile __memalign_hook)(size_t alignment, size_t size,
                                  const void *caller);

void (*volatile __free_hook)(void *ptr, const void *caller);

void (*__malloc_initialize_hook)(void);

void (*volatile __after_morecore_hook)(void);
```

DESCRIPTION

The GNU C library lets you modify the behavior of *malloc*(3), *realloc*(3), and *free*(3) by specifying appropriate hook functions. You can use these hooks to help you debug programs that use dynamic memory allocation, for example.

The variable **__malloc_initialize_hook** points at a function that is called once when the malloc implementation is initialized. This is a weak variable, so it can be overridden in the application with a definition like the following:

```
void (*__malloc_initialize_hook)(void) = my_init_hook;
```

Now the function *my_init_hook*() can do the initialization of all hooks.

The four functions pointed to by **__malloc_hook**, **__realloc_hook**, **__memalign_hook**, **__free_hook** have a prototype like the functions *malloc*(3), *realloc*(3), *memalign*(3), *free*(3), respectively, except that they have a final argument *caller* that gives the address of the caller of *malloc*(3), etc.

The variable **__after_morecore_hook** points at a function that is called each time after *sbrk*(2) was asked for more memory.

STANDARDS

GNU.

NOTES

The use of these hook functions is not safe in multithreaded programs, and they are now deprecated. From glibc 2.24 onwards, the **__malloc_initialize_hook** variable has been removed from the API, and from glibc 2.34 onwards, all the hook variables have been removed from the API. Programmers should instead preempt calls to the relevant functions by defining and exporting **malloc**(), **free**(), **realloc**(), and **calloc**().

EXAMPLES

Here is a short example of how to use these variables.

```
#include <stdio.h>
#include <malloc.h>

/* Prototypes for our hooks */
static void my_init_hook(void);
static void *my_malloc_hook(size_t, const void *);

/* Variables to save original hooks */
static void *(*old_malloc_hook)(size_t, const void *);

/* Override initializing hook from the C library */
void (*__malloc_initialize_hook)(void) = my_init_hook;
```

```
static void
my_init_hook(void)
{
    old_malloc_hook = __malloc_hook;
    __malloc_hook = my_malloc_hook;
}

static void *
my_malloc_hook(size_t size, const void *caller)
{
    void *result;

    /* Restore all old hooks */
    __malloc_hook = old_malloc_hook;

    /* Call recursively */
    result = malloc(size);

    /* Save underlying hooks */
    old_malloc_hook = __malloc_hook;

    /* printf() might call malloc(), so protect it too */
    printf("malloc(%zu) called from %p returns %p\n",
            size, caller, result);

    /* Restore our own hooks */
    __malloc_hook = my_malloc_hook;

    return result;
}
```

SEE ALSO

mallinfo(3), *malloc*(3), *mcheck*(3), *mtrace*(3)

NAME

malloc_info − export malloc state to a stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

int malloc_info(int *options***, FILE ****stream***);**

DESCRIPTION

The **malloc_info**() function exports an XML string that describes the current state of the memory-allocation implementation in the caller. The string is printed on the file stream *stream*. The exported string includes information about all arenas (see *malloc*(3)).

As currently implemented, *options* must be zero.

RETURN VALUE

On success, **malloc_info**() returns 0. On failure, it returns −1, and *errno* is set to indicate the error.

ERRORS

EINVAL

options was nonzero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc_info()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.10.

NOTES

The memory-allocation information is provided as an XML string (rather than a C structure) because the information may change over time (according to changes in the underlying implementation). The output XML string includes a version field.

The *open_memstream*(3) function can be used to send the output of **malloc_info**() directly into a buffer in memory, rather than to a file.

The **malloc_info**() function is designed to address deficiencies in *malloc_stats*(3) and *mallinfo*(3).

EXAMPLES

The program below takes up to four command-line arguments, of which the first three are mandatory. The first argument specifies the number of threads that the program should create. All of the threads, including the main thread, allocate the number of blocks of memory specified by the second argument. The third argument controls the size of the blocks to be allocated. The main thread creates blocks of this size, the second thread created by the program allocates blocks of twice this size, the third thread allocates blocks of three times this size, and so on.

The program calls **malloc_info**() twice to display the memory-allocation state. The first call takes place before any threads are created or memory allocated. The second call is performed after all threads have allocated memory.

In the following example, the command-line arguments specify the creation of one additional thread, and both the main thread and the additional thread allocate 10000 blocks of memory. After the blocks of memory have been allocated, **malloc_info**() shows the state of two allocation arenas.

```
$ getconf GNU_LIBC_VERSION
glibc 2.13
$ ./a.out 1 10000 100
============ Before allocating blocks ============
<malloc version="1">
<heap nr="0">
<sizes>
```

```
        </sizes>
        <total type="fast" count="0" size="0"/>
        <total type="rest" count="0" size="0"/>
        <system type="current" size="135168"/>
        <system type="max" size="135168"/>
        <aspace type="total" size="135168"/>
        <aspace type="mprotect" size="135168"/>
        </heap>
        <total type="fast" count="0" size="0"/>
        <total type="rest" count="0" size="0"/>
        <system type="current" size="135168"/>
        <system type="max" size="135168"/>
        <aspace type="total" size="135168"/>
        <aspace type="mprotect" size="135168"/>
        </malloc>

        ============ After allocating blocks ============
        <malloc version="1">
        <heap nr="0">
        <sizes>
        </sizes>
        <total type="fast" count="0" size="0"/>
        <total type="rest" count="0" size="0"/>
        <system type="current" size="1081344"/>
        <system type="max" size="1081344"/>
        <aspace type="total" size="1081344"/>
        <aspace type="mprotect" size="1081344"/>
        </heap>
        <heap nr="1">
        <sizes>
        </sizes>
        <total type="fast" count="0" size="0"/>
        <total type="rest" count="0" size="0"/>
        <system type="current" size="1032192"/>
        <system type="max" size="1032192"/>
        <aspace type="total" size="1032192"/>
        <aspace type="mprotect" size="1032192"/>
        </heap>
        <total type="fast" count="0" size="0"/>
        <total type="rest" count="0" size="0"/>
        <system type="current" size="2113536"/>
        <system type="max" size="2113536"/>
        <aspace type="total" size="2113536"/>
        <aspace type="mprotect" size="2113536"/>
        </malloc>
```

Program source

```
#include <err.h>
#include <errno.h>
#include <malloc.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

static size_t        blockSize;
static size_t        numThreads;
static unsigned int  numBlocks;

static void *
```

```
thread_func(void *arg)
{
    int tn = (int) arg;

    /* The multiplier '(2 + tn)' ensures that each thread (including
       the main thread) allocates a different amount of memory. */

    for (unsigned int j = 0; j < numBlocks; j++)
        if (malloc(blockSize * (2 + tn)) == NULL)
            err(EXIT_FAILURE, "malloc-thread");

    sleep(100);         /* Sleep until main thread terminates. */
    return NULL;
}

int
main(int argc, char *argv[])
{
    int        sleepTime;
    pthread_t  *thr;

    if (argc < 4) {
        fprintf(stderr,
                "%s num-threads num-blocks block-size [sleep-time]\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    numThreads = atoi(argv[1]);
    numBlocks = atoi(argv[2]);
    blockSize = atoi(argv[3]);
    sleepTime = (argc > 4) ? atoi(argv[4]) : 0;

    thr = calloc(numThreads, sizeof(*thr));
    if (thr == NULL)
        err(EXIT_FAILURE, "calloc");

    printf("============ Before allocating blocks ============\n");
    malloc_info(0, stdout);

    /* Create threads that allocate different amounts of memory. */

    for (size_t tn = 0; tn < numThreads; tn++) {
        errno = pthread_create(&thr[tn], NULL, thread_func,
                               (void *) tn);
        if (errno != 0)
            err(EXIT_FAILURE, "pthread_create");

        /* If we add a sleep interval after the start-up of each
           thread, the threads likely won't contend for malloc
           mutexes, and therefore additional arenas won't be
           allocated (see malloc(3)). */

        if (sleepTime > 0)
            sleep(sleepTime);
    }

    /* The main thread also allocates some memory. */
```

```
        for (unsigned int j = 0; j < numBlocks; j++)
            if (malloc(blockSize) == NULL)
                err(EXIT_FAILURE, "malloc");

        sleep(2);           /* Give all threads a chance to
                               complete allocations. */

        printf("\n============ After allocating blocks ============\n");
        malloc_info(0, stdout);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

mallinfo(3), *malloc*(3), *malloc_stats*(3), *mallopt*(3), *open_memstream*(3)

NAME

malloc_stats − print memory allocation statistics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

void malloc_stats(void);

DESCRIPTION

The **malloc_stats**() function prints (on standard error) statistics about memory allocated by *malloc*(3) and related functions. For each arena (allocation area), this function prints the total amount of memory allocated and the total number of bytes consumed by in-use allocations. (These two values correspond to the *arena* and *uordblks* fields retrieved by *mallinfo*(3).) In addition, the function prints the sum of these two statistics for all arenas, and the maximum number of blocks and bytes that were ever simultaneously allocated using *mmap*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc_stats()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.0.

NOTES

More detailed information about memory allocations in the main arena can be obtained using *mallinfo*(3).

SEE ALSO

mmap(2), *mallinfo*(3), *malloc*(3), *malloc_info*(3), *mallopt*(3)

NAME

malloc_trim − release free memory from the heap

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

int malloc_trim(size_t *pad***);**

DESCRIPTION

The **malloc_trim**() function attempts to release free memory from the heap (by calling *sbrk*(2) or *madvise*(2) with suitable arguments).

The *pad* argument specifies the amount of free space to leave untrimmed at the top of the heap. If this argument is 0, only the minimum amount of memory is maintained at the top of the heap (i.e., one page or less). A nonzero argument can be used to maintain some trailing space at the top of the heap in order to allow future allocations to be made without having to extend the heap with *sbrk*(2).

RETURN VALUE

The **malloc_trim**() function returns 1 if memory was actually released back to the system, or 0 if it was not possible to release any memory.

ERRORS

No errors are defined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc_trim()	Thread safety	MT-Safe

STANDARDS

GNU.

VERSIONS

glibc 2.0.

NOTES

Only the main heap (using *sbrk*(2)) honors the *pad* argument; thread heaps do not.

Since glibc 2.8 this function frees memory in all arenas and in all chunks with whole free pages.

Before glibc 2.8 this function only freed memory at the top of the heap in the main arena.

SEE ALSO

sbrk(2), *malloc*(3), *mallopt*(3)

NAME

malloc_usable_size − obtain size of block of memory allocated from heap

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

size_t malloc_usable_size(void *_Nullable *ptr***);**

DESCRIPTION

This function can be used for diagnostics or statistics about allocations from *malloc*(3) or a related function.

RETURN VALUE

malloc_usable_size() returns a value no less than the size of the block of allocated memory pointed to by *ptr*. If *ptr* is NULL, 0 is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
malloc_usable_size()	Thread safety	MT-Safe

STANDARDS

GNU.

CAVEATS

The value returned by **malloc_usable_size**() may be greater than the requested size of the allocation because of various internal implementation details, none of which the programmer should rely on. This function is intended to only be used for diagnostics and statistics; writing to the excess memory without first calling *realloc*(3) to resize the allocation is not supported. The returned value is only valid at the time of the call.

SEE ALSO

malloc(3)

NAME

mallopt − set memory allocation parameters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <malloc.h>

int mallopt(int *param***, int** *value***);**

DESCRIPTION

The **mallopt**() function adjusts parameters that control the behavior of the memory-allocation functions (see *malloc*(3)). The *param* argument specifies the parameter to be modified, and *value* specifies the new value for that parameter.

The following values can be specified for *param*:

M_ARENA_MAX

If this parameter has a nonzero value, it defines a hard limit on the maximum number of arenas that can be created. An arena represents a pool of memory that can be used by *malloc*(3) (and similar) calls to service allocation requests. Arenas are thread safe and therefore may have multiple concurrent memory requests. The trade-off is between the number of threads and the number of arenas. The more arenas you have, the lower the per-thread contention, but the higher the memory usage.

The default value of this parameter is 0, meaning that the limit on the number of arenas is determined according to the setting of **M_ARENA_TEST**.

This parameter has been available since glibc 2.10 via **--enable-experimental-malloc**, and since glibc 2.15 by default. In some versions of the allocator there was no limit on the number of created arenas (e.g., CentOS 5, RHEL 5).

When employing newer glibc versions, applications may in some cases exhibit high contention when accessing arenas. In these cases, it may be beneficial to increase **M_ARENA_MAX** to match the number of threads. This is similar in behavior to strategies taken by tcmalloc and jemalloc (e.g., per-thread allocation pools).

M_ARENA_TEST

This parameter specifies a value, in number of arenas created, at which point the system configuration will be examined to determine a hard limit on the number of created arenas. (See **M_ARENA_MAX** for the definition of an arena.)

The computation of the arena hard limit is implementation-defined and is usually calculated as a multiple of the number of available CPUs. Once the hard limit is computed, the result is final and constrains the total number of arenas.

The default value for the **M_ARENA_TEST** parameter is 2 on systems where *sizeof(long)* is 4; otherwise the default value is 8.

This parameter has been available since glibc 2.10 via **--enable-experimental-malloc**, and since glibc 2.15 by default.

The value of **M_ARENA_TEST** is not used when **M_ARENA_MAX** has a nonzero value.

M_CHECK_ACTION

Setting this parameter controls how glibc responds when various kinds of programming errors are detected (e.g., freeing the same pointer twice). The 3 least significant bits (2, 1, and 0) of the value assigned to this parameter determine the glibc behavior, as follows:

Bit 0 If this bit is set, then print a one-line message on *stderr* that provides details about the error. The message starts with the string "*** glibc detected ***", followed by the program name, the name of the memory-allocation function in which the error was detected, a brief description of the error, and the memory address where the error was detected.

Bit 1 If this bit is set, then, after printing any error message specified by bit 0, the program is terminated by calling *abort*(3). Since glibc 2.4, if bit 0 is also set, then, between printing the error message and aborting, the program also prints a stack trace in the

manner of *backtrace*(3), and prints the process's memory mapping in the style of */proc/*pid*/maps* (see *proc*(5)).

Bit 2 (since glibc 2.4)
: This bit has an effect only if bit 0 is also set. If this bit is set, then the one-line message describing the error is simplified to contain just the name of the function where the error was detected and the brief description of the error.

The remaining bits in *value* are ignored.

Combining the above details, the following numeric values are meaningful for **M_CHECK_ACTION**:

0 Ignore error conditions; continue execution (with undefined results).

1 Print a detailed error message and continue execution.

2 Abort the program.

3 Print detailed error message, stack trace, and memory mappings, and abort the program.

5 Print a simple error message and continue execution.

7 Print simple error message, stack trace, and memory mappings, and abort the program.

Since glibc 2.3.4, the default value for the **M_CHECK_ACTION** parameter is 3. In glibc 2.3.3 and earlier, the default value is 1.

Using a nonzero **M_CHECK_ACTION** value can be useful because otherwise a crash may happen much later, and the true cause of the problem is then very hard to track down.

M_MMAP_MAX
: This parameter specifies the maximum number of allocation requests that may be simultaneously serviced using *mmap*(2). This parameter exists because some systems have a limited number of internal tables for use by *mmap*(2), and using more than a few of them may degrade performance.

The default value is 65,536, a value which has no special significance and which serves only as a safeguard. Setting this parameter to 0 disables the use of *mmap*(2) for servicing large allocation requests.

M_MMAP_THRESHOLD
: For allocations greater than or equal to the limit specified (in bytes) by **M_MMAP_THRESHOLD** that can't be satisfied from the free list, the memory-allocation functions employ *mmap*(2) instead of increasing the program break using *sbrk*(2).

Allocating memory using *mmap*(2) has the significant advantage that the allocated memory blocks can always be independently released back to the system. (By contrast, the heap can be trimmed only if memory is freed at the top end.) On the other hand, there are some disadvantages to the use of *mmap*(2): deallocated space is not placed on the free list for reuse by later allocations; memory may be wasted because *mmap*(2) allocations must be page-aligned; and the kernel must perform the expensive task of zeroing out memory allocated via *mmap*(2). Balancing these factors leads to a default setting of 128*1024 for the **M_MMAP_THRESHOLD** parameter.

The lower limit for this parameter is 0. The upper limit is **DEFAULT_MMAP_THRESHOLD_MAX**: 512*1024 on 32-bit systems or *4*1024*1024*sizeof(long)* on 64-bit systems.

Note: Nowadays, glibc uses a dynamic mmap threshold by default. The initial value of the threshold is 128*1024, but when blocks larger than the current threshold and less than or equal to **DEFAULT_MMAP_THRESHOLD_MAX** are freed, the threshold is adjusted upward to the size of the freed block. When dynamic mmap thresholding is in effect, the threshold for trimming the heap is also dynamically adjusted to be twice the dynamic mmap threshold. Dynamic adjustment of the mmap threshold is disabled if any of the **M_TRIM_THRESHOLD**, **M_TOP_PAD**, **M_MMAP_THRESHOLD**, or **M_MMAP_MAX** parameters is set.

M_MXFAST (since glibc 2.3)
Set the upper limit for memory allocation requests that are satisfied using "fastbins". (The measurement unit for this parameter is bytes.) Fastbins are storage areas that hold deallocated blocks of memory of the same size without merging adjacent free blocks. Subsequent reallocation of blocks of the same size can be handled very quickly by allocating from the fastbin, although memory fragmentation and the overall memory footprint of the program can increase.

The default value for this parameter is *64*sizeof(size_t)/4* (i.e., 64 on 32-bit architectures). The range for this parameter is 0 to *80*sizeof(size_t)/4*. Setting **M_MXFAST** to 0 disables the use of fastbins.

M_PERTURB (since glibc 2.4)
If this parameter is set to a nonzero value, then bytes of allocated memory (other than allocations via *calloc*(3)) are initialized to the complement of the value in the least significant byte of *value*, and when allocated memory is released using *free*(3), the freed bytes are set to the least significant byte of *value*. This can be useful for detecting errors where programs incorrectly rely on allocated memory being initialized to zero, or reuse values in memory that has already been freed.

The default value for this parameter is 0.

M_TOP_PAD
This parameter defines the amount of padding to employ when calling *sbrk*(2) to modify the program break. (The measurement unit for this parameter is bytes.) This parameter has an effect in the following circumstances:

- When the program break is increased, then **M_TOP_PAD** bytes are added to the *sbrk*(2) request.
- When the heap is trimmed as a consequence of calling *free*(3) (see the discussion of **M_TRIM_THRESHOLD**) this much free space is preserved at the top of the heap.

In either case, the amount of padding is always rounded to a system page boundary.

Modifying **M_TOP_PAD** is a trade-off between increasing the number of system calls (when the parameter is set low) and wasting unused memory at the top of the heap (when the parameter is set high).

The default value for this parameter is 128*1024.

M_TRIM_THRESHOLD
When the amount of contiguous free memory at the top of the heap grows sufficiently large, *free*(3) employs *sbrk*(2) to release this memory back to the system. (This can be useful in programs that continue to execute for a long period after freeing a significant amount of memory.) The **M_TRIM_THRESHOLD** parameter specifies the minimum size (in bytes) that this block of memory must reach before *sbrk*(2) is used to trim the heap.

The default value for this parameter is 128*1024. Setting **M_TRIM_THRESHOLD** to −1 disables trimming completely.

Modifying **M_TRIM_THRESHOLD** is a trade-off between increasing the number of system calls (when the parameter is set low) and wasting unused memory at the top of the heap (when the parameter is set high).

Environment variables

A number of environment variables can be defined to modify some of the same parameters as are controlled by **mallopt**(). Using these variables has the advantage that the source code of the program need not be changed. To be effective, these variables must be defined before the first call to a memory-allocation function. (If the same parameters are adjusted via **mallopt**(), then the **mallopt**() settings take precedence.) For security reasons, these variables are ignored in set-user-ID and set-group-ID programs.

The environment variables are as follows (note the trailing underscore at the end of the name of some variables):

MALLOC_ARENA_MAX
Controls the same parameter as **mallopt**() **M_ARENA_MAX**.

MALLOC_ARENA_TEST
Controls the same parameter as **mallopt**() **M_ARENA_TEST**.

MALLOC_CHECK_
This environment variable controls the same parameter as **mallopt**() **M_CHECK_ACTION**. If this variable is set to a nonzero value, then a special implementation of the memory-allocation functions is used. (This is accomplished using the *malloc_hook*(3) feature.) This implementation performs additional error checking, but is slower than the standard set of memory-allocation functions. (This implementation does not detect all possible errors; memory leaks can still occur.)

The value assigned to this environment variable should be a single digit, whose meaning is as described for **M_CHECK_ACTION**. Any characters beyond the initial digit are ignored.

For security reasons, the effect of **MALLOC_CHECK_** is disabled by default for set-user-ID and set-group-ID programs. However, if the file */etc/suid−debug* exists (the content of the file is irrelevant), then **MALLOC_CHECK_** also has an effect for set-user-ID and set-group-ID programs.

MALLOC_MMAP_MAX_
Controls the same parameter as **mallopt**() **M_MMAP_MAX**.

MALLOC_MMAP_THRESHOLD_
Controls the same parameter as **mallopt**() **M_MMAP_THRESHOLD**.

MALLOC_PERTURB_
Controls the same parameter as **mallopt**() **M_PERTURB**.

MALLOC_TRIM_THRESHOLD_
Controls the same parameter as **mallopt**() **M_TRIM_THRESHOLD**.

MALLOC_TOP_PAD_
Controls the same parameter as **mallopt**() **M_TOP_PAD**.

RETURN VALUE

On success, **mallopt**() returns 1. On error, it returns 0.

ERRORS

On error, *errno* is *not* set.

VERSIONS

A similar function exists on many System V derivatives, but the range of values for *param* varies across systems. The SVID defined options **M_MXFAST**, **M_NLBLKS**, **M_GRAIN**, and **M_KEEP**, but only the first of these is implemented in glibc.

STANDARDS

None.

HISTORY

glibc 2.0.

BUGS

Specifying an invalid value for *param* does not generate an error.

A calculation error within the glibc implementation means that a call of the form:

```
mallopt(M_MXFAST, n)
```

does not result in fastbins being employed for all allocations of size up to *n*. To ensure desired results, *n* should be rounded up to the next multiple greater than or equal to *(2k+1)*sizeof(size_t)*, where *k* is an integer.

If **mallopt**() is used to set **M_PERTURB**, then, as expected, the bytes of allocated memory are initialized to the complement of the byte in *value*, and when that memory is freed, the bytes of the region are initialized to the byte specified in *value*. However, there is an off-by-*sizeof(size_t)* error in the implementation: instead of initializing precisely the block of memory being freed by the call *free(p)*, the block starting at *p+sizeof(size_t)* is initialized.

EXAMPLES

The program below demonstrates the use of **M_CHECK_ACTION**. If the program is supplied with an (integer) command-line argument, then that argument is used to set the **M_CHECK_ACTION** parameter. The program then allocates a block of memory, and frees it twice (an error).

The following shell session shows what happens when we run this program under glibc, with the default value for **M_CHECK_ACTION**:

```
$ ./a.out
main(): returned from first free() call
*** glibc detected *** ./a.out: double free or corruption (top): 0x09d30008 ***
======= Backtrace: =========
/lib/libc.so.6(+0x6c501)[0x523501]
/lib/libc.so.6(+0x6dd70)[0x524d70]
/lib/libc.so.6(cfree+0x6d)[0x527e5d]
./a.out[0x80485db]
/lib/libc.so.6(__libc_start_main+0xe7)[0x4cdce7]
./a.out[0x8048471]
======= Memory map: ========
001e4000-001fe000 r-xp 00000000 08:06 1083555    /lib/libgcc_s.so.1
001fe000-001ff000 r--p 00019000 08:06 1083555    /lib/libgcc_s.so.1
[some lines omitted]
b7814000-b7817000 rw-p 00000000 00:00 0
bff53000-bff74000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

The following runs show the results when employing other values for **M_CHECK_ACTION**:

```
$ ./a.out 1             # Diagnose error and continue
main(): returned from first free() call
*** glibc detected *** ./a.out: double free or corruption (top): 0x09cbe008 ***
main(): returned from second free() call
$ ./a.out 2             # Abort without error message
main(): returned from first free() call
Aborted (core dumped)
$ ./a.out 0             # Ignore error and continue
main(): returned from first free() call
main(): returned from second free() call
```

The next run shows how to set the same parameter using the **MALLOC_CHECK_** environment variable:

```
$ MALLOC_CHECK_=1 ./a.out
main(): returned from first free() call
*** glibc detected *** ./a.out: free(): invalid pointer: 0x092c2008 ***
main(): returned from second free() call
```

Program source

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    char *p;

    if (argc > 1) {
        if (mallopt(M_CHECK_ACTION, atoi(argv[1])) != 1) {
            fprintf(stderr, "mallopt() failed");
            exit(EXIT_FAILURE);
        }
```

```
        }

        p = malloc(1000);
        if (p == NULL) {
            fprintf(stderr, "malloc() failed");
            exit(EXIT_FAILURE);
        }

        free(p);
        printf("%s(): returned from first free() call\n", __func__);

        free(p);
        printf("%s(): returned from second free() call\n", __func__);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

mmap(2), *sbrk*(2), *mallinfo*(3), *malloc*(3), *malloc_hook*(3), *malloc_info*(3), *malloc_stats*(3), *malloc_trim*(3), *mcheck*(3), *mtrace*(3), *posix_memalign*(3)

NAME

matherr − SVID math library exception handling

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

[[deprecated]] int matherr(struct exception *****exc*);**

[[deprecated]] extern _LIB_VERSION_TYPE _LIB_VERSION;

DESCRIPTION

Note: the mechanism described in this page is no longer supported by glibc. Before glibc 2.27, it had been marked as obsolete. Since glibc 2.27, the mechanism has been removed altogether. New applications should use the techniques described in *math_error*(7) and *fenv*(3). This page documents the **matherr**() mechanism as an aid for maintaining and porting older applications.

The System V Interface Definition (SVID) specifies that various math functions should invoke a function called **matherr**() if a math exception is detected. This function is called before the math function returns; after **matherr**() returns, the system then returns to the math function, which in turn returns to the caller.

To employ **matherr**(), the programmer must define the **_SVID_SOURCE** feature test macro (before including *any* header files), and assign the value **_SVID_** to the external variable **_LIB_VERSION**.

The system provides a default version of **matherr**(). This version does nothing, and returns zero (see below for the significance of this). The default **matherr**() can be overridden by a programmer-defined version, which will be invoked when an exception occurs. The function is invoked with one argument, a pointer to an *exception* structure, defined as follows:

```
struct exception {
    int    type;      /* Exception type */
    char  *name;      /* Name of function causing exception */
    double arg1;      /* 1st argument to function */
    double arg2;      /* 2nd argument to function */
    double retval;    /* Function return value */
}
```

The *type* field has one of the following values:

DOMAIN A domain error occurred (the function argument was outside the range for which the function is defined). The return value depends on the function; *errno* is set to **EDOM**.

SING A pole error occurred (the function result is an infinity). The return value in most cases is **HUGE** (the largest single precision floating-point number), appropriately signed. In most cases, *errno* is set to **EDOM**.

OVERFLOW
An overflow occurred. In most cases, the value **HUGE** is returned, and *errno* is set to **ERANGE**.

UNDERFLOW
An underflow occurred. 0.0 is returned, and *errno* is set to **ERANGE**.

TLOSS Total loss of significance. 0.0 is returned, and *errno* is set to **ERANGE**.

PLOSS Partial loss of significance. This value is unused on glibc (and many other systems).

The *arg1* and *arg2* fields are the arguments supplied to the function (*arg2* is undefined for functions that take only one argument).

The *retval* field specifies the return value that the math function will return to its caller. The programmer-defined **matherr**() can modify this field to change the return value of the math function.

If the **matherr**() function returns zero, then the system sets *errno* as described above, and may print an error message on standard error (see below).

If the **matherr**() function returns a nonzero value, then the system does not set *errno*, and doesn't print an error message.

Math functions that employ matherr()

The table below lists the functions and circumstances in which **matherr**() is called. The "Type" column indicates the value assigned to *exc−>type* when calling **matherr**(). The "Result" column is the default return value assigned to *exc−>retval*.

The "Msg?" and "errno" columns describe the default behavior if **matherr**() returns zero. If the "Msg?" columns contains "y", then the system prints an error message on standard error.

The table uses the following notations and abbreviations:

x first argument to function
y second argument to function
fin finite value for argument
neg negative value for argument
int integral value for argument
o/f result overflowed
u/f result underflowed
|x| absolute value of x
X_TLOSS is a constant defined in <*math.h*>

Function	**Type**	**Result**	**Msg?**	**errno**
acos(|x|>1)	DOMAIN	HUGE	y	EDOM
asin(|x|>1)	DOMAIN	HUGE	y	EDOM
atan2(0,0)	DOMAIN	HUGE	y	EDOM
acosh(x<1)	DOMAIN	NAN	y	EDOM
atanh(|x|>1)	DOMAIN	NAN	y	EDOM
atanh(|x|==1)	SING	(x>0.0)? HUGE_VAL : −HUGE_VAL	y	EDOM
cosh(fin) o/f	OVERFLOW	HUGE	n	ERANGE
sinh(fin) o/f	OVERFLOW	(x>0.0) ? HUGE : −HUGE	n	ERANGE
sqrt(x<0)	DOMAIN	0.0	y	EDOM
hypot(fin,fin) o/f	OVERFLOW	HUGE	n	ERANGE
exp(fin) o/f	OVERFLOW	HUGE	n	ERANGE
exp(fin) u/f	UNDERFLOW	0.0	n	ERANGE
exp2(fin) o/f	OVERFLOW	HUGE	n	ERANGE
exp2(fin) u/f	UNDERFLOW	0.0	n	ERANGE
exp10(fin) o/f	OVERFLOW	HUGE	n	ERANGE
exp10(fin) u/f	UNDERFLOW	0.0	n	ERANGE
j0(|x|>X_TLOSS)	TLOSS	0.0	y	ERANGE
j1(|x|>X_TLOSS)	TLOSS	0.0	y	ERANGE
jn(|x|>X_TLOSS)	TLOSS	0.0	y	ERANGE
y0(x>X_TLOSS)	TLOSS	0.0	y	ERANGE
y1(x>X_TLOSS)	TLOSS	0.0	y	ERANGE
yn(x>X_TLOSS)	TLOSS	0.0	y	ERANGE
y0(0)	DOMAIN	−HUGE	y	EDOM
y0(x<0)	DOMAIN	−HUGE	y	EDOM
y1(0)	DOMAIN	−HUGE	y	EDOM
y1(x<0)	DOMAIN	−HUGE	y	EDOM
yn(n,0)	DOMAIN	−HUGE	y	EDOM
yn(x<0)	DOMAIN	−HUGE	y	EDOM
lgamma(fin) o/f	OVERFLOW	HUGE	n	ERANGE
lgamma(−int) or lgamma(0)	SING	HUGE	y	EDOM
tgamma(fin) o/f	OVERFLOW	HUGE_VAL	n	ERANGE
tgamma(−int)	SING	NAN	y	EDOM
tgamma(0)	SING	copysign(HUGE_VAL,x)	y	ERANGE
log(0)	SING	−HUGE	y	EDOM

log(x<0)	DOMAIN	−HUGE	y	EDOM
log2(0)	SING	−HUGE	n	EDOM
log2(x<0)	DOMAIN	−HUGE	n	EDOM
log10(0)	SING	−HUGE	y	EDOM
log10(x<0)	DOMAIN	−HUGE	y	EDOM
pow(0.0,0.0)	DOMAIN	0.0	y	EDOM
pow(x,y) o/f	OVERFLOW	HUGE	n	ERANGE
pow(x,y) u/f	UNDERFLOW	0.0	n	ERANGE
pow(NaN,0.0)	DOMAIN	x	n	EDOM
0**neg	DOMAIN	0.0	y	EDOM
neg**non-int	DOMAIN	0.0	y	EDOM
scalb() o/f	OVERFLOW	(x>0.0) ? HUGE_VAL : −HUGE_VAL	n	ERANGE
scalb() u/f	UNDERFLOW	copysign(0.0,x)	n	ERANGE
fmod(x,0)	DOMAIN	x	y	EDOM
remainder(x,0)	DOMAIN	NAN	y	EDOM

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
matherr()	Thread safety	MT-Safe

EXAMPLES

The example program demonstrates the use of **matherr**() when calling *log*(3). The program takes up to three command-line arguments. The first argument is the floating-point number to be given to *log*(3). If the optional second argument is provided, then **_LIB_VERSION** is set to **_SVID_** so that **matherr**() is called, and the integer supplied in the command-line argument is used as the return value from **matherr**(). If the optional third command-line argument is supplied, then it specifies an alternative return value that **matherr**() should assign as the return value of the math function.

The following example run, where *log*(3) is given an argument of 0.0, does not use **matherr**():

```
$ ./a.out 0.0
errno: Numerical result out of range
x=-inf
```

In the following run, **matherr**() is called, and returns 0:

```
$ ./a.out 0.0 0
matherr SING exception in log() function
        args:   0.000000, 0.000000
        retval: -340282346638528859811704183484516925440.000000
log: SING error
errno: Numerical argument out of domain
x=-340282346638528859811704183484516925440.000000
```

The message "log: SING error" was printed by the C library.

In the following run, **matherr**() is called, and returns a nonzero value:

```
$ ./a.out 0.0 1
matherr SING exception in log() function
        args:   0.000000, 0.000000
        retval: -340282346638528859811704183484516925440.000000
x=-340282346638528859811704183484516925440.000000
```

In this case, the C library did not print a message, and *errno* was not set.

In the following run, **matherr**() is called, changes the return value of the math function, and returns a nonzero value:

```
$ ./a.out 0.0 1 12345.0
matherr SING exception in log() function
        args:   0.000000, 0.000000
```

```
            retval: -340282346638528859811704183484516925440.000000
    x=12345.000000
```

Program source

```
#define _SVID_SOURCE
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static int matherr_ret = 0;     /* Value that matherr()
                                   should return */
static int change_retval = 0;   /* Should matherr() change
                                   function's return value? */
static double new_retval;       /* New function return value */

int
matherr(struct exception *exc)
{
    fprintf(stderr, "matherr %s exception in %s() function\n",
            (exc->type == DOMAIN) ?    "DOMAIN" :
            (exc->type == OVERFLOW) ?  "OVERFLOW" :
            (exc->type == UNDERFLOW) ? "UNDERFLOW" :
            (exc->type == SING) ?      "SING" :
            (exc->type == TLOSS) ?     "TLOSS" :
            (exc->type == PLOSS) ?     "PLOSS" : "???",
            exc->name);
    fprintf(stderr, "        args:   %f, %f\n",
            exc->arg1, exc->arg2);
    fprintf(stderr, "        retval: %f\n", exc->retval);

    if (change_retval)
        exc->retval = new_retval;

    return matherr_ret;
}

int
main(int argc, char *argv[])
{
    double x;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <argval>"
                " [<matherr-ret> [<new-func-retval>]]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (argc > 2) {
        _LIB_VERSION = _SVID_;
        matherr_ret = atoi(argv[2]);
    }

    if (argc > 3) {
        change_retval = 1;
        new_retval = atof(argv[3]);
    }
```

```
        x = log(atof(argv[1]));
        if (errno != 0)
            perror("errno");

        printf("x=%f\n", x);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fenv(3), *math_error*(7), *standards*(7)

NAME

MAX, MIN − maximum or minimum of two values

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/param.h>

MAX(*a***,** *b***);**
MIN(*a***,** *b***);**

DESCRIPTION

These macros return the maximum or minimum of *a* and *b*.

RETURN VALUE

These macros return the value of one of their arguments, possibly converted to a different type (see BUGS).

ERRORS

These macros may raise the "invalid" floating-point exception when any of the arguments is NaN.

STANDARDS

GNU, BSD.

NOTES

If either of the arguments is of a floating-point type, you might prefer to use *fmax*(3) or *fmin*(3), which can handle NaN.

The arguments may be evaluated more than once, or not at all.

Some UNIX systems might provide these macros in a different header, or not at all.

BUGS

Due to the usual arithmetic conversions, the result of these macros may be very different from either of the arguments. To avoid this, ensure that both arguments have the same type.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/param.h>

int
main(int argc, char *argv[])
{
    int a, b, x;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <num> <num>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    a = atoi(argv[1]);
    b = atoi(argv[2]);
    x = MAX(a, b);
    printf("MAX(%d, %d) is %d\n", a, b, x);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fmax(3), *fmin*(3)

NAME

MB_CUR_MAX − maximum length of a multibyte character in the current locale

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <stdlib.h>

DESCRIPTION

The **MB_CUR_MAX** macro defines an integer expression giving the maximum number of bytes needed to represent a single wide character in the current locale. This value is locale dependent and therefore not a compile-time constant.

RETURN VALUE

An integer in the range [1, **MB_LEN_MAX**]. The value 1 denotes traditional 8-bit encoded characters.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

SEE ALSO

MB_LEN_MAX(3), *mblen*(3), *mbstowcs*(3), *mbtowc*(3), *wcstombs*(3), *wctomb*(3)

NAME

MB_LEN_MAX − maximum multibyte length of a character across all locales

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <limits.h>

DESCRIPTION

The **MB_LEN_MAX** macro is the maximum number of bytes needed to represent a single wide character, in any of the supported locales.

RETURN VALUE

A constant integer greater than zero.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The entities **MB_LEN_MAX** and *sizeof(wchar_t)* are totally unrelated. In glibc, **MB_LEN_MAX** is typically 16 (6 in glibc versions earlier than 2.2), while *sizeof(wchar_t)* is 4.

SEE ALSO

MB_CUR_MAX(3)

NAME

mblen − determine number of bytes in next multibyte character

LIBRARY

Standard C library (*libc*, −*lc*)

SYNOPSIS

#include <stdlib.h>

int mblen(const char *s***[.***n***], size_t** *n***);**

DESCRIPTION

If *s* is not NULL, the **mblen**() function inspects at most *n* bytes of the multibyte string starting at *s* and extracts the next complete multibyte character. It uses a static anonymous shift state known only to the **mblen**() function. If the multibyte character is not the null wide character, it returns the number of bytes that were consumed from *s*. If the multibyte character is the null wide character, it returns 0.

If the *n* bytes starting at *s* do not contain a complete multibyte character, **mblen**() returns −1. This can happen even if *n* is greater than or equal to *MB_CUR_MAX*, if the multibyte string contains redundant shift sequences.

If the multibyte string starting at *s* contains an invalid multibyte sequence before the next complete character, **mblen**() also returns −1.

If *s* is NULL, the **mblen**() function resets the shift state, known to only this function, to the initial state, and returns nonzero if the encoding has nontrivial shift state, or zero if the encoding is stateless.

RETURN VALUE

The **mblen**() function returns the number of bytes parsed from the multibyte sequence starting at *s*, if a non-null wide character was recognized. It returns 0, if a null wide character was recognized. It returns −1, if an invalid multibyte sequence was encountered or if it couldn't parse a complete multibyte character.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mblen()	Thread safety	MT-Unsafe race

VERSIONS

The function *mbrlen*(3) provides a better interface to the same functionality.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mblen**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

mbrlen(3)

NAME

mbrlen − determine number of bytes in next multibyte character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t mbrlen(const char *s[restrict .n], size_t n,
              mbstate_t *restrict ps);
```

DESCRIPTION

The **mbrlen**() function inspects at most *n* bytes of the multibyte string starting at *s* and extracts the next complete multibyte character. It updates the shift state **ps*. If the multibyte character is not the null wide character, it returns the number of bytes that were consumed from *s*. If the multibyte character is the null wide character, it resets the shift state **ps* to the initial state and returns 0.

If the *n* bytes starting at *s* do not contain a complete multibyte character, **mbrlen**() returns *(size_t) −2*. This can happen even if *n >= MB_CUR_MAX*, if the multibyte string contains redundant shift sequences.

If the multibyte string starting at *s* contains an invalid multibyte sequence before the next complete character, **mbrlen**() returns *(size_t) −1* and sets *errno* to **EILSEQ**. In this case, the effects on **ps* are undefined.

If *ps* is NULL, a static anonymous state known only to the **mbrlen**() function is used instead.

RETURN VALUE

The **mbrlen**() function returns the number of bytes parsed from the multibyte sequence starting at *s*, if a non-null wide character was recognized. It returns 0, if a null wide character was recognized. It returns *(size_t) −1* and sets *errno* to **EILSEQ**, if an invalid multibyte sequence was encountered. It returns *(size_t) −2* if it couldn't parse a complete multibyte character, meaning that *n* should be increased.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbrlen()	Thread safety	MT-Unsafe race:mbrlen/!ps

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbrlen**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

mbrtowc(3)

NAME

mbrtowc − convert a multibyte sequence to a wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t mbrtowc(wchar_t *restrict pwc, const char s[restrict .n],
               size_t n, mbstate_t *restrict ps);
```

DESCRIPTION

The main case for this function is when *s* is not NULL and *pwc* is not NULL. In this case, the **mbrtowc**() function inspects at most *n* bytes of the multibyte string starting at *s*, extracts the next complete multibyte character, converts it to a wide character and stores it at **pwc*. It updates the shift state **ps*. If the converted wide character is not L'\0' (the null wide character), it returns the number of bytes that were consumed from *s*. If the converted wide character is L'\0', it resets the shift state **ps* to the initial state and returns 0.

If the *n* bytes starting at *s* do not contain a complete multibyte character, **mbrtowc**() returns *(size_t) −2*. This can happen even if *n >= MB_CUR_MAX*, if the multibyte string contains redundant shift sequences.

If the multibyte string starting at *s* contains an invalid multibyte sequence before the next complete character, **mbrtowc**() returns *(size_t) −1* and sets *errno* to **EILSEQ**. In this case, the effects on **ps* are undefined.

A different case is when *s* is not NULL but *pwc* is NULL. In this case, the **mbrtowc**() function behaves as above, except that it does not store the converted wide character in memory.

A third case is when *s* is NULL. In this case, *pwc* and *n* are ignored. If the conversion state represented by **ps* denotes an incomplete multibyte character conversion, the **mbrtowc**() function returns *(size_t) −1*, sets *errno* to **EILSEQ**, and leaves **ps* in an undefined state. Otherwise, the **mbrtowc**() function puts **ps* in the initial state and returns 0.

In all of the above cases, if *ps* is NULL, a static anonymous state known only to the **mbrtowc**() function is used instead. Otherwise, **ps* must be a valid *mbstate_t* object. An *mbstate_t* object *a* can be initialized to the initial state by zeroing it, for example using

```
memset(&a, 0, sizeof(a));
```

RETURN VALUE

The **mbrtowc**() function returns the number of bytes parsed from the multibyte sequence starting at *s*, if a non-L'\0' wide character was recognized. It returns 0, if a L'\0' wide character was recognized. It returns *(size_t) −1* and sets *errno* to **EILSEQ**, if an invalid multibyte sequence was encountered. It returns *(size_t) −2* if it couldn't parse a complete multibyte character, meaning that *n* should be increased.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbrtowc()	Thread safety	MT-Unsafe race:mbrtowc/!ps

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbrtowc**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

mbsinit(3), *mbsrtowcs*(3)

NAME

mbsinit − test for initial shift state

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int mbsinit(const mbstate_t * *ps***);**

DESCRIPTION

Character conversion between the multibyte representation and the wide character representation uses conversion state, of type *mbstate_t*. Conversion of a string uses a finite-state machine; when it is interrupted after the complete conversion of a number of characters, it may need to save a state for processing the remaining characters. Such a conversion state is needed for the sake of encodings such as ISO/IEC 2022 and UTF-7.

The initial state is the state at the beginning of conversion of a string. There are two kinds of state: the one used by multibyte to wide character conversion functions, such as *mbsrtowcs*(3), and the one used by wide character to multibyte conversion functions, such as *wcsrtombs*(3), but they both fit in a *mbstate_t*, and they both have the same representation for an initial state.

For 8-bit encodings, all states are equivalent to the initial state. For multibyte encodings like UTF-8, EUC-*, BIG5, or SJIS, the wide character to multibyte conversion functions never produce non-initial states, but the multibyte to wide-character conversion functions like *mbrtowc*(3) do produce non-initial states when interrupted in the middle of a character.

One possible way to create an *mbstate_t* in initial state is to set it to zero:

```
mbstate_t state;
memset(&state, 0, sizeof(state));
```

On Linux, the following works as well, but might generate compiler warnings:

```
mbstate_t state = { 0 };
```

The function **mbsinit**() tests whether **ps* corresponds to an initial state.

RETURN VALUE

mbsinit() returns nonzero if **ps* is an initial state, or if *ps* is NULL. Otherwise, it returns 0.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbsinit()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbsinit**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

mbrlen(3), *mbrtowc*(3), *mbsrtowcs*(3), *wcrtomb*(3), *wcsrtombs*(3)

NAME

mbsnrtowcs − convert a multibyte string to a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t mbsnrtowcs(wchar_t dest[restrict .len], const char **restrict src,
                  size_t nms, size_t len, mbstate_t *restrict ps);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mbsnrtowcs():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **mbsnrtowcs**() function is like the *mbsrtowcs*(3) function, except that the number of bytes to be converted, starting at **src*, is limited to at most *nms* bytes.

If *dest* is not NULL, the **mbsnrtowcs**() function converts at most *nms* bytes from the multibyte string **src* to a wide-character string starting at *dest*. At most *len* wide characters are written to *dest*. The shift state **ps* is updated. The conversion is effectively performed by repeatedly calling *mbrtowc(dest, *src, n, ps)* where *n* is some positive number, as long as this call succeeds, and then incrementing *dest* by one and **src* by the number of bytes consumed. The conversion can stop for three reasons:

- An invalid multibyte sequence has been encountered. In this case, **src* is left pointing to the invalid multibyte sequence, *(size_t) −1* is returned, and *errno* is set to **EILSEQ**.
- The *nms* limit forces a stop, or *len* non-L'\0' wide characters have been stored at *dest*. In this case, **src* is left pointing to the next multibyte sequence to be converted, and the number of wide characters written to *dest* is returned.
- The multibyte string has been completely converted, including the terminating null wide character ('\0') (which has the side effect of bringing back **ps* to the initial state). In this case, **src* is set to NULL, and the number of wide characters written to *dest*, excluding the terminating null wide character, is returned.

According to POSIX.1, if the input buffer ends with an incomplete character, it is unspecified whether conversion stops at the end of the previous character (if any), or at the end of the input buffer. The glibc implementation adopts the former behavior.

If *dest* is NULL, *len* is ignored, and the conversion proceeds as above, except that the converted wide characters are not written out to memory, and that no destination length limit exists.

In both of the above cases, if *ps* is NULL, a static anonymous state known only to the **mbsnrtowcs**() function is used instead.

The programmer must ensure that there is room for at least *len* wide characters at *dest*.

RETURN VALUE

The **mbsnrtowcs**() function returns the number of wide characters that make up the converted part of the wide-character string, not including the terminating null wide character. If an invalid multibyte sequence was encountered, *(size_t) −1* is returned, and *errno* set to **EILSEQ**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbsnrtowcs()	Thread safety	MT-Unsafe race:mbsnrtowcs/!ps

STANDARDS

POSIX.1-2008.

NOTES

The behavior of **mbsnrtowcs**() depends on the **LC_CTYPE** category of the current locale.

Passing NULL as *ps* is not multithread safe.

SEE ALSO

iconv(3), *mbrtowc*(3), *mbsinit*(3), *mbsrtowcs*(3)

NAME

mbsrtowcs − convert a multibyte string to a wide-character string (restartable)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t dest[restrict .dsize],
                 const char **restrict src,
                 size_t dsize, mbstate_t *restrict ps);
```

DESCRIPTION

If *dest* is not NULL, convert the multibyte string **src* to a wide-character string starting at *dest*. At most *dsize* wide characters are written to *dest*. The shift state **ps* is updated. The conversion is effectively performed by repeatedly calling *mbrtowc(dest, *src, n, ps)* where *n* is some positive number, as long as this call succeeds, and then incrementing *dest* by one and **src* by the number of bytes consumed. The conversion can stop for three reasons:

- An invalid multibyte sequence has been encountered. In this case, **src* is left pointing to the invalid multibyte sequence, *(size_t) −1* is returned, and *errno* is set to **EILSEQ**.
- *dsize* non-L'\0' wide characters have been stored at *dest*. In this case, **src* is left pointing to the next multibyte sequence to be converted, and the number of wide characters written to *dest* is returned.
- The multibyte string has been completely converted, including the terminating null wide character ('\0'), which has the side effect of bringing back **ps* to the initial state. In this case, **src* is set to NULL, and the number of wide characters written to *dest*, excluding the terminating null wide character, is returned.

If *dest* is NULL, *dsize* is ignored, and the conversion proceeds as above, except that the converted wide characters are not written out to memory, and that no length limit exists.

In both of the above cases, if *ps* is NULL, a static anonymous state known only to the **mbsrtowcs**() function is used instead.

In order to avoid the case 2 above, the programmer should make sure *dsize* is greater than or equal to *mbsrtowcs(NULL,src,0,ps)+1*.

The programmer must ensure that there is room for at least *dsize* wide characters at *dest*.

This function is a restartable version of *mbstowcs*(3).

RETURN VALUE

The number of wide characters that make up the converted part of the wide-character string, not including the terminating null wide character. If an invalid multibyte sequence was encountered, *(size_t) −1* is returned, and *errno* set to **EILSEQ**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbsrtowcs()	Thread safety	MT-Unsafe race:mbsrtowcs/!ps

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbsrtowcs**() depends on the **LC_CTYPE** category of the current locale.

Passing NULL as *ps* is not multithread safe.

SEE ALSO

iconv(3), *mbrtowc*(3), *mbsinit*(3), *mbsnrtowcs*(3), *mbstowcs*(3)

NAME

mbstowcs − convert a multibyte string to a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(wchar_t dest[restrict .dsize], const char *restrict src,
                size_t dsize);
```

DESCRIPTION

If *dest* is not NULL, convert the multibyte string *src* to a wide-character string starting at *dest*. At most *dsize* wide characters are written to *dest*. The sequence of characters in the string *src* shall begin in the initial shift state. The conversion can stop for three reasons:

- An invalid multibyte sequence has been encountered. In this case, *(size_t) −1* is returned.
- *dsize* non-L'\0' wide characters have been stored at *dest*. In this case, the number of wide characters written to *dest* is returned, but the shift state at this point is lost.
- The multibyte string has been completely converted, including the terminating null character ('\0'). In this case, the number of wide characters written to *dest*, excluding the terminating null wide character, is returned.

If *dest* is NULL, *dsize* is ignored, and the conversion proceeds as above, except that the converted wide characters are not written out to memory, and that no length limit exists.

In order to avoid the case 2 above, the programmer should make sure *dsize* is greater than or equal to *mbstowcs(NULL,src,0)+1*.

The programmer must ensure that there is room for at least *dsize* wide characters at *dest*.

RETURN VALUE

The number of wide characters that make up the converted part of the wide-character string, not including the terminating null wide character. If an invalid multibyte sequence was encountered, *(size_t) −1* is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbstowcs()	Thread safety	MT-Safe

VERSIONS

The function *mbsrtowcs*(3) provides a better interface to the same functionality.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbstowcs**() depends on the **LC_CTYPE** category of the current locale.

EXAMPLES

The program below illustrates the use of **mbstowcs**(), as well as some of the wide character classification functions. An example run is the following:

```
$ ./t_mbstowcs de_DE.UTF-8 Grüße!
Length of source string (excluding terminator):
    8 bytes
    6 multibyte characters

Wide character string is: Grüße! (6 characters)
    G alpha upper
    r alpha lower
    ü alpha lower
```

```
        ß alpha lower
        e alpha lower
        ! !alpha
```

Program source

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int
main(int argc, char *argv[])
{
    size_t mbslen;      /* Number of multibyte characters in source */
    wchar_t *wcs;       /* Pointer to converted wide character string */

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <locale> <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Apply the specified locale. */

    if (setlocale(LC_ALL, argv[1]) == NULL) {
        perror("setlocale");
        exit(EXIT_FAILURE);
    }

    /* Calculate the length required to hold argv[2] converted to
       a wide character string. */

    mbslen = mbstowcs(NULL, argv[2], 0);
    if (mbslen == (size_t) -1) {
        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    /* Describe the source string to the user. */

    printf("Length of source string (excluding terminator):\n");
    printf("    %zu bytes\n", strlen(argv[2]));
    printf("    %zu multibyte characters\n\n", mbslen);

    /* Allocate wide character string of the desired size.  Add 1
       to allow for terminating null wide character (L'\0'). */

    wcs = calloc(mbslen + 1, sizeof(*wcs));
    if (wcs == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }

    /* Convert the multibyte character string in argv[2] to a
       wide character string. */

    if (mbstowcs(wcs, argv[2], mbslen + 1) == (size_t) -1) {
```

```
            perror("mbstowcs");
            exit(EXIT_FAILURE);
        }

        printf("Wide character string is: %ls (%zu characters)\n",
               wcs, mbslen);

        /* Now do some inspection of the classes of the characters in
           the wide character string. */

        for (wchar_t *wp = wcs; *wp != 0; wp++) {
            printf("    %lc ", (wint_t) *wp);

            if (!iswalpha(*wp))
                printf("!");
            printf("alpha ");

            if (iswalpha(*wp)) {
                if (iswupper(*wp))
                    printf("upper ");

                if (iswlower(*wp))
                    printf("lower ");
            }

            putchar('\n');
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

mblen(3), *mbsrtowcs*(3), *mbtowc*(3), *wcstombs*(3), *wctomb*(3)

NAME

mbtowc − convert a multibyte sequence to a wide character

LIBRARY

Standard C library (*libc*, −*lc*)

SYNOPSIS

#include <stdlib.h>

int mbtowc(wchar_t *restrict *pwc***, const char** *s***[restrict .**n**], size_t** *n***);**

DESCRIPTION

The main case for this function is when *s* is not NULL and *pwc* is not NULL. In this case, the **mbtowc**() function inspects at most *n* bytes of the multibyte string starting at *s*, extracts the next complete multibyte character, converts it to a wide character and stores it at **pwc*. It updates an internal shift state known only to the **mbtowc**() function. If *s* does not point to a null byte ('\0'), it returns the number of bytes that were consumed from *s*, otherwise it returns 0.

If the *n* bytes starting at *s* do not contain a complete multibyte character, or if they contain an invalid multibyte sequence, **mbtowc**() returns −1. This can happen even if *n* >= *MB_CUR_MAX*, if the multibyte string contains redundant shift sequences.

A different case is when *s* is not NULL but *pwc* is NULL. In this case, the **mbtowc**() function behaves as above, except that it does not store the converted wide character in memory.

A third case is when *s* is NULL. In this case, *pwc* and *n* are ignored. The **mbtowc**() function resets the shift state, only known to this function, to the initial state, and returns nonzero if the encoding has nontrivial shift state, or zero if the encoding is stateless.

RETURN VALUE

If *s* is not NULL, the **mbtowc**() function returns the number of consumed bytes starting at *s*, or 0 if *s* points to a null byte, or −1 upon failure.

If *s* is NULL, the **mbtowc**() function returns nonzero if the encoding has nontrivial shift state, or zero if the encoding is stateless.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mbtowc()	Thread safety	MT-Unsafe race

VERSIONS

This function is not multithread safe. The function *mbrtowc*(3) provides a better interface to the same functionality.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **mbtowc**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

MB_CUR_MAX(3), *mblen*(3), *mbrtowc*(3), *mbstowcs*(3), *wcstombs*(3), *wctomb*(3)

NAME

mcheck, mcheck_check_all, mcheck_pedantic, mprobe − heap consistency checking

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <mcheck.h>

int mcheck(void (*abortfunc)(enum mcheck_status mstatus));
int mcheck_pedantic(void (*abortfunc)(enum mcheck_status mstatus));
void mcheck_check_all(void);

enum mcheck_status mprobe(void *ptr);
```

DESCRIPTION

The **mcheck**() function installs a set of debugging hooks for the *malloc*(3) family of memory-allocation functions. These hooks cause certain consistency checks to be performed on the state of the heap. The checks can detect application errors such as freeing a block of memory more than once or corrupting the bookkeeping data structures that immediately precede a block of allocated memory.

To be effective, the **mcheck**() function must be called before the first call to *malloc*(3) or a related function. In cases where this is difficult to ensure, linking the program with *−lmcheck* inserts an implicit call to **mcheck**() (with a NULL argument) before the first call to a memory-allocation function.

The **mcheck_pedantic**() function is similar to **mcheck**(), but performs checks on all allocated blocks whenever one of the memory-allocation functions is called. This can be very slow!

The **mcheck_check_all**() function causes an immediate check on all allocated blocks. This call is effective only if **mcheck**() is called beforehand.

If the system detects an inconsistency in the heap, the caller-supplied function pointed to by *abortfunc* is invoked with a single argument, *mstatus*, that indicates what type of inconsistency was detected. If *abortfunc* is NULL, a default function prints an error message on *stderr* and calls *abort*(3).

The **mprobe**() function performs a consistency check on the block of allocated memory pointed to by *ptr*. The **mcheck**() function should be called beforehand (otherwise **mprobe**() returns **MCHECK_DISABLED**).

The following list describes the values returned by **mprobe**() or passed as the *mstatus* argument when *abortfunc* is invoked:

MCHECK_DISABLED (**mprobe**() only)
: **mcheck**() was not called before the first memory allocation function was called. Consistency checking is not possible.

MCHECK_OK (**mprobe**() only)
: No inconsistency detected.

MCHECK_HEAD
: Memory preceding an allocated block was clobbered.

MCHECK_TAIL
: Memory following an allocated block was clobbered.

MCHECK_FREE
: A block of memory was freed twice.

RETURN VALUE

mcheck() and **mcheck_pedantic**() return 0 on success, or −1 on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mcheck(), **mcheck_pedantic**(), **mcheck_check_all**(), **mprobe**()	Thread safety	MT-Unsafe race:mcheck const:malloc_hooks

STANDARDS

GNU.

HISTORY

mcheck_pedantic()
mcheck_check_all()
: glibc 2.2.

mcheck()
mprobe()
: glibc 2.0.

NOTES

Linking a program with *−lmcheck* and using the **MALLOC_CHECK_** environment variable (described in *mallopt*(3)) cause the same kinds of errors to be detected. But, using **MALLOC_CHECK_** does not require the application to be relinked.

EXAMPLES

The program below calls **mcheck**() with a NULL argument and then frees the same block of memory twice. The following shell session demonstrates what happens when running the program:

```
$ ./a.out
About to free

About to free a second time
block freed twice
Aborted (core dumped)
```

Program source

```
#include <mcheck.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    char *p;

    if (mcheck(NULL) != 0) {
        fprintf(stderr, "mcheck() failed\n");

        exit(EXIT_FAILURE);
    }

    p = malloc(1000);

    fprintf(stderr, "About to free\n");
    free(p);
    fprintf(stderr, "\nAbout to free a second time\n");
    free(p);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

malloc(3), *mallopt*(3), *mtrace*(3)

NAME

memccpy − copy memory area

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

void *memccpy(void dest[restrict .n], const void src[restrict .n],
        int c, size_t n);
```

DESCRIPTION

The **memccpy**() function copies no more than *n* bytes from memory area *src* to memory area *dest*, stopping when the character *c* is found.

If the memory areas overlap, the results are undefined.

RETURN VALUE

The **memccpy**() function returns a pointer to the next character in *dest* after *c*, or NULL if *c* was not found in the first *n* characters of *src*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memccpy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

bcopy(3), *bstring*(3), *memcpy*(3), *memmove*(3), *strcpy*(3), *strncpy*(3)

NAME

memchr, memrchr, rawmemchr − scan memory for a character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

void *memchr(const void s[.n], int c, size_t n);
void *memrchr(const void s[.n], int c, size_t n);

[[deprecated]] void *rawmemchr(const void *s, int c);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

memrchr(), **rawmemchr**():
_GNU_SOURCE

DESCRIPTION

The **memchr**() function scans the initial *n* bytes of the memory area pointed to by *s* for the first instance of *c*. Both *c* and the bytes of the memory area pointed to by *s* are interpreted as *unsigned char*.

The **memrchr**() function is like the **memchr**() function, except that it searches backward from the end of the *n* bytes pointed to by *s* instead of forward from the beginning.

The **rawmemchr**() function is similar to **memchr**(), but it assumes (i.e., the programmer knows for certain) that an instance of *c* lies somewhere in the memory area starting at the location pointed to by *s*. If an instance of *c* is not found, the behavior is undefined. Use either *strlen*(3) or *memchr*(3) instead.

RETURN VALUE

The **memchr**() and **memrchr**() functions return a pointer to the matching byte or NULL if the character does not occur in the given memory area.

The **rawmemchr**() function returns a pointer to the matching byte.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memchr(), **memrchr**(), **rawmemchr**()	Thread safety	MT-Safe

STANDARDS

memchr()
C11, POSIX.1-2008.

memrchr()
rawmemchr()
GNU.

HISTORY

memchr()
POSIX.1-2001, C89, SVr4, 4.3BSD.

memrchr()
glibc 2.2.

rawmemchr()
glibc 2.1.

SEE ALSO

bstring(3), *ffs*(3), *memmem*(3), *strchr*(3), *strpbrk*(3), *strrchr*(3), *strsep*(3), *strspn*(3), *strstr*(3), *wmemchr*(3)

NAME

memcmp − compare memory areas

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

int memcmp(const void *s1***[.***n***], const void** *s2***[.***n***], size_t** *n***);**

DESCRIPTION

The **memcmp**() function compares the first *n* bytes (each interpreted as *unsigned char*) of the memory areas *s1* and *s2*.

RETURN VALUE

The **memcmp**() function returns an integer less than, equal to, or greater than zero if the first *n* bytes of *s1* is found, respectively, to be less than, to match, or be greater than the first *n* bytes of *s2*.

For a nonzero return value, the sign is determined by the sign of the difference between the first pair of bytes (interpreted as *unsigned char*) that differ in *s1* and *s2*.

If *n* is zero, the return value is zero.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memcmp()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS

Do not use **memcmp**() to compare confidential data, such as cryptographic secrets, because the CPU time required for the comparison depends on the contents of the addresses compared, this function is subject to timing-based side-channel attacks. In such cases, a function that performs comparisons in deterministic time, depending only on *n* (the quantity of bytes compared) is required. Some operating systems provide such a function (e.g., NetBSD's *consttime_memequal*()), but no such function is specified in POSIX. On Linux, you may need to implement such a function yourself.

SEE ALSO

bstring(3), *strcasecmp*(3), *strcmp*(3), *strcoll*(3), *strncasecmp*(3), *strncmp*(3), *wmemcmp*(3)

NAME

memcpy − copy memory area

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

void *memcpy(void dest[restrict .n], const void src[restrict .n],
        size_t n);
```

DESCRIPTION

The **memcpy**() function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas must not overlap. Use *memmove*(3) if the memory areas do overlap.

RETURN VALUE

The **memcpy**() function returns a pointer to *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memcpy()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS

Failure to observe the requirement that the memory areas do not overlap has been the source of significant bugs. (POSIX and the C standards are explicit that employing **memcpy**() with overlapping areas produces undefined behavior.) Most notably, in glibc 2.13 a performance optimization of **memcpy**() on some platforms (including x86-64) included changing the order in which bytes were copied from *src* to *dest*.

This change revealed breakages in a number of applications that performed copying with overlapping areas. Under the previous implementation, the order in which the bytes were copied had fortuitously hidden the bug, which was revealed when the copying order was reversed. In glibc 2.14, a versioned symbol was added so that old binaries (i.e., those linked against glibc versions earlier than 2.14) employed a **memcpy**() implementation that safely handles the overlapping buffers case (by providing an "older" **memcpy**() implementation that was aliased to *memmove*(3)).

SEE ALSO

bcopy(3), *bstring*(3), *memccpy*(3), *memmove*(3), *mempcpy*(3), *strcpy*(3), *strncpy*(3), *wmemcpy*(3)

NAME

memfrob − frobnicate (obfuscate) a memory area

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

void *memfrob(void s[.n], size_t n);
```

DESCRIPTION

The **memfrob**() function obfuscates the first *n* bytes of the memory area *s* by exclusive-ORing each character with the number 42. The effect can be reversed by using **memfrob**() on the obfuscated memory area.

Note that this function is not a proper encryption routine as the XOR constant is fixed, and is suitable only for hiding strings.

RETURN VALUE

The **memfrob**() function returns a pointer to the obfuscated memory area.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memfrob()	Thread safety	MT-Safe

STANDARDS

GNU.

SEE ALSO

bstring(3), *strfry*(3)

NAME

memmem − locate a substring

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

void *memmem(const void haystack[.haystacklen], size_t haystacklen,
        const void needle[.needlelen], size_t needlelen);
```

DESCRIPTION

The **memmem**() function finds the start of the first occurrence of the substring *needle* of length *needlelen* in the memory area *haystack* of length *haystacklen*.

RETURN VALUE

The **memmem**() function returns a pointer to the beginning of the substring, or NULL if the substring is not found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memmem()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

musl libc 0.9.7; FreeBSD 6.0, OpenBSD 5.4, NetBSD, Illumos.

BUGS

In glibc 2.0, if *needle* is empty, **memmem**() returns a pointer to the last byte of *haystack*. This is fixed in glibc 2.1.

SEE ALSO

bstring(3), *strstr*(3)

NAME

memmove − copy memory area

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

void *memmove(void *dest***[.***n***], const void** *src***[.***n***], size_t** *n***);**

DESCRIPTION

The **memmove**() function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas may overlap: copying takes place as though the bytes in *src* are first copied into a temporary array that does not overlap *src* or *dest*, and the bytes are then copied from the temporary array to *dest*.

RETURN VALUE

The **memmove**() function returns a pointer to *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memmove()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

bcopy(3), *bstring*(3), *memccpy*(3), *memcpy*(3), *strcpy*(3), *strncpy*(3), *wmemmove*(3)

NAME

mempcpy, wmempcpy − copy memory area

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

void *mempcpy(void dest[restrict .n], const void src[restrict .n],
        size_t n);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <wchar.h>

wchar_t *wmempcpy(wchar_t dest[restrict .n],
        const wchar_t src[restrict .n],
        size_t n);
```

DESCRIPTION

The **mempcpy**() function is nearly identical to the *memcpy*(3) function. It copies *n* bytes from the object beginning at *src* into the object pointed to by *dest*. But instead of returning the value of *dest* it returns a pointer to the byte following the last written byte.

This function is useful in situations where a number of objects shall be copied to consecutive memory positions.

The **wmempcpy**() function is identical but takes *wchar_t* type arguments and copies *n* wide characters.

RETURN VALUE

dest + *n*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mempcpy(), **wmempcpy**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

EXAMPLES

```
void *
combine(void *o1, size_t s1, void *o2, size_t s2)
{
    void *result = malloc(s1 + s2);
    if (result != NULL)
        mempcpy(mempcpy(result, o1, s1), o2, s2);
    return result;
}
```

SEE ALSO

memccpy(3), *memcpy*(3), *memmove*(3), *wmemcpy*(3)

NAME

memset − fill memory with a constant byte

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

void *memset(void *s***[.***n***], int** *c***, size_t** *n***);**

DESCRIPTION

The **memset**() function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

RETURN VALUE

The **memset**() function returns a pointer to the memory area *s*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
memset()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

bstring(3), *bzero*(3), *swab*(3), *wmemset*(3)

NAME

mkdtemp − create a unique temporary directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

char *mkdtemp(char **template*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mkdtemp():
/* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc 2.19 and earlier: */ _BSD_SOURCE
|| /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION

The **mkdtemp**() function generates a uniquely named temporary directory from *template*. The last six characters of *template* must be XXXXXX and these are replaced with a string that makes the directory name unique. The directory is then created with permissions 0700. Since it will be modified, *template* must not be a string constant, but should be declared as a character array.

RETURN VALUE

The **mkdtemp**() function returns a pointer to the modified template string on success, and NULL on failure, in which case *errno* is set to indicate the error.

ERRORS

EINVAL
The last six characters of *template* were not XXXXXX. Now *template* is unchanged.

Also see *mkdir*(2) for other possible values for *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mkdtemp()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1.91. NetBSD 1.4. POSIX.1-2008.

SEE ALSO

mktemp(1), *mkdir*(2), *mkstemp*(3), *mktemp*(3), *tempnam*(3), *tmpfile*(3), *tmpnam*(3)

NAME

mkfifo, mkfifoat − make a FIFO special file (a named pipe)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

#include <fcntl.h>          /* Definition of AT_* constants */
#include <sys/stat.h>

int mkfifoat(int dirfd, const char *pathname, mode_t mode);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mkfifoat():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

mkfifo() makes a FIFO special file with name *pathname*. *mode* specifies the FIFO's permissions. It is modified by the process's **umask** in the usual way: the permissions of the created file are **(***mode* **& ~umask)**.

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the filesystem by calling **mkfifo**().

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa. See *fifo*(7) for nonblocking handling of FIFO special files.

mkfifoat()

The **mkfifoat**() function operates in exactly the same way as **mkfifo**(), except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **mkfifo**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like *mkfifo*())

If *pathname* is absolute, then *dirfd* is ignored.

See *openat*(2) for an explanation of the need for **mkfifoat**().

RETURN VALUE

On success **mkfifo**() and **mkfifoat**() return 0. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES
One of the directories in *pathname* did not allow search (execute) permission.

EBADF
(**mkfifoat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EDQUOT
The user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST
pathname already exists. This includes the case where *pathname* is a symbolic link, dangling or not.

ENAMETOOLONG
Either the total length of *pathname* is greater than **PATH_MAX**, or an individual filename component has a length greater than **NAME_MAX**. In the GNU system, there is no imposed limit on overall filename length, but some filesystems may place limits on the length of a component.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOSPC
The directory or filesystem has no room for the new file.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR
(**mkfifoat**()) *pathname* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

EROFS
pathname refers to a read-only filesystem.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mkfifo(), **mkfifoat**()	Thread safety	MT-Safe

VERSIONS

It is implemented using *mknodat*(2).

STANDARDS

POSIX.1-2008.

HISTORY

mkfifo()
POSIX.1-2001.

mkfifoat()
glibc 2.4. POSIX.1-2008.

SEE ALSO

mkfifo(1), *close*(2), *open*(2), *read*(2), *stat*(2), *umask*(2), *write*(2), *fifo*(7)

NAME

mkstemp, mkostemp, mkstemps, mkostemps − create a unique temporary file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int mkstemp(char *template);
int mkostemp(char *template, int  flags);
int mkstemps(char *template, int suffixlen);
int mkostemps(char *template, int suffixlen, int  flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mkstemp():
  _XOPEN_SOURCE >= 500
    || /* glibc >= 2.12: */ _POSIX_C_SOURCE >= 200809L
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

mkostemp():
  _GNU_SOURCE

mkstemps():
  /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

mkostemps():
  _GNU_SOURCE

DESCRIPTION

The **mkstemp**() function generates a unique temporary filename from *template*, creates and opens the file, and returns an open file descriptor for the file.

The last six characters of *template* must be "XXXXXX" and these are replaced with a string that makes the filename unique. Since it will be modified, *template* must not be a string constant, but should be declared as a character array.

The file is created with permissions 0600, that is, read plus write for owner only. The returned file descriptor provides both read and write access to the file. The file is opened with the *open*(2) **O_EXCL** flag, guaranteeing that the caller is the process that creates the file.

The **mkostemp**() function is like **mkstemp**(), with the difference that the following bits—with the same meaning as for *open*(2)—may be specified in *flags*: **O_APPEND**, **O_CLOEXEC**, and **O_SYNC**. Note that when creating the file, **mkostemp**() includes the values **O_RDWR**, **O_CREAT**, and **O_EXCL** in the *flags* argument given to *open*(2); including these values in the *flags* argument given to **mkostemp**() is unnecessary, and produces errors on some systems.

The **mkstemps**() function is like **mkstemp**(), except that the string in *template* contains a suffix of *suffixlen* characters. Thus, *template* is of the form *prefixXXXXXXsuffix*, and the string XXXXXX is modified as for **mkstemp**().

The **mkostemps**() function is to **mkstemps**() as **mkostemp**() is to **mkstemp**().

RETURN VALUE

On success, these functions return the file descriptor of the temporary file. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EEXIST
: Could not create a unique temporary filename. Now the contents of *template* are undefined.

EINVAL
: For **mkstemp**() and **mkostemp**(): The last six characters of *template* were not XXXXXX; now *template* is unchanged.

  For **mkstemps**() and **mkostemps**(): *template* is less than *(6 + suffixlen)* characters long, or the last 6 characters before the suffix in *template* were not XXXXXX.

These functions may also fail with any of the errors described for *open*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mkstemp(), **mkostemp**(), **mkstemps**(), **mkostemps**()	Thread safety	MT-Safe

STANDARDS

mkstemp()
: POSIX.1-2001.

mkstemps()
: BSD.

mkostemp()
mkostemps()
: GNU.

HISTORY

mkstemp()
: 4.3BSD, POSIX.1-2001.

mkstemps()
: glibc 2.11. BSD, Mac OS X, Solaris, Tru64.

mkostemp()
: glibc 2.7.

mkostemps()
: glibc 2.11.

In glibc versions 2.06 and earlier, the file is created with permissions 0666, that is, read and write for all users. This old behavior may be a security risk, especially since other UNIX flavors use 0600, and somebody might overlook this detail when porting programs. POSIX.1-2008 adds a requirement that the file be created with mode 0600.

More generally, the POSIX specification of **mkstemp**() does not say anything about file modes, so the application should make sure its file mode creation mask (see *umask*(2)) is set appropriately before calling **mkstemp**() (and *mkostemp*())

SEE ALSO

mkdtemp(3), *mktemp*(3), *tempnam*(3), *tmpfile*(3), *tmpnam*(3)

NAME

mktemp − make a unique temporary filename

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

char *mktemp(char **template*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mktemp():
Since glibc 2.12:
(_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200112L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
Before glibc 2.12:
_BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

Never use this function; see BUGS.

The **mktemp**() function generates a unique temporary filename from *template*. The last six characters of *template* must be XXXXXX and these are replaced with a string that makes the filename unique. Since it will be modified, *template* must not be a string constant, but should be declared as a character array.

RETURN VALUE

The **mktemp**() function always returns *template*. If a unique name was created, the last six bytes of *template* will have been modified in such a way that the resulting name is unique (i.e., does not exist already) If a unique name could not be created, *template* is made an empty string, and *errno* is set to indicate the error.

ERRORS

EINVAL
The last six characters of *template* were not XXXXXX.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mktemp()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD, POSIX.1-2001. Removed in POSIX.1-2008.

BUGS

Never use **mktemp**(). Some implementations follow 4.3BSD and replace XXXXXX by the current process ID and a single letter, so that at most 26 different names can be returned. Since on the one hand the names are easy to guess, and on the other hand there is a race between testing whether the name exists and opening the file, every use of **mktemp**() is a security risk. The race is avoided by *mkstemp*(3) and *mkdtemp*(3).

SEE ALSO

mktemp(1), *mkdtemp*(3), *mkstemp*(3), *tempnam*(3), *tmpfile*(3), *tmpnam*(3)

NAME

modf, modff, modfl − extract signed integral and fractional values from floating-point number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double modf(double x, double *iptr);
float modff(float x, float *iptr);
long double modfl(long double x, long double *iptr);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

modff(), **modfl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions break the argument *x* into an integral part and a fractional part, each of which has the same sign as *x*. The integral part is stored in the location pointed to by *iptr*.

RETURN VALUE

These functions return the fractional part of *x*.

If *x* is a NaN, a NaN is returned, and **iptr* is set to a NaN.

If *x* is positive infinity (negative infinity), +0 (−0) is returned, and **iptr* is set to positive infinity (negative infinity).

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
modf(), **modff**(), **modfl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

frexp(3), *ldexp*(3)

NAME

mpool − shared memory buffer pool

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <db.h>
#include <mpool.h>

MPOOL *mpool_open(DBT *_key_, int _fd_, pgno_t _pagesize_, pgno_t _maxcache_);

void mpool_filter(MPOOL *_mp_, void (*pgin)(void *, pgno_t, void *),
void (*_pgout_)(void *, pgno_t, void *),
void *_pgcookie_);

void *mpool_new(MPOOL *_mp_, pgno_t *_pgnoaddr_);
void *mpool_get(MPOOL *_mp_, pgno_t _pgno_, unsigned int _flags_);
int mpool_put(MPOOL *_mp_, void *_pgaddr_, unsigned int _flags_);

int mpool_sync(MPOOL *_mp_);
int mpool_close(MPOOL *_mp_);

DESCRIPTION

Note well: This page documents interfaces provided up until glibc 2.1. Since glibc 2.2, glibc no longer provides these interfaces. Probably, you are looking for the APIs provided by the *libdb* library instead.

Mpool is the library interface intended to provide page oriented buffer management of files. The buffers may be shared between processes.

The function **mpool_open**() initializes a memory pool. The *key* argument is the byte string used to negotiate between multiple processes wishing to share buffers. If the file buffers are mapped in shared memory, all processes using the same key will share the buffers. If *key* is NULL, the buffers are mapped into private memory. The *fd* argument is a file descriptor for the underlying file, which must be seekable. If *key* is non-NULL and matches a file already being mapped, the *fd* argument is ignored.

The *pagesize* argument is the size, in bytes, of the pages into which the file is broken up. The *maxcache* argument is the maximum number of pages from the underlying file to cache at any one time. This value is not relative to the number of processes which share a file's buffers, but will be the largest value specified by any of the processes sharing the file.

The **mpool_filter**() function is intended to make transparent input and output processing of the pages possible. If the *pgin* function is specified, it is called each time a buffer is read into the memory pool from the backing file. If the *pgout* function is specified, it is called each time a buffer is written into the backing file. Both functions are called with the *pgcookie* pointer, the page number and a pointer to the page to being read or written.

The function **mpool_new**() takes an *MPOOL* pointer and an address as arguments. If a new page can be allocated, a pointer to the page is returned and the page number is stored into the *pgnoaddr* address. Otherwise, NULL is returned and *errno* is set.

The function **mpool_get**() takes an *MPOOL* pointer and a page number as arguments. If the page exists, a pointer to the page is returned. Otherwise, NULL is returned and *errno* is set. The *flags* argument is not currently used.

The function **mpool_put**() unpins the page referenced by *pgaddr*. *pgaddr* must be an address previously returned by **mpool_get**() or **mpool_new**(). The flag value is specified by ORing any of the following values:

MPOOL_DIRTY
The page has been modified and needs to be written to the backing file.

mpool_put() returns 0 on success and −1 if an error occurs.

The function **mpool_sync**() writes all modified pages associated with the *MPOOL* pointer to the backing file. **mpool_sync**() returns 0 on success and −1 if an error occurs.

The **mpool_close**() function free's up any allocated memory associated with the memory pool cookie. Modified pages are **not** written to the backing file. **mpool_close**() returns 0 on success and −1 if an

error occurs.

ERRORS

The **mpool_open**() function may fail and set *errno* for any of the errors specified for the library routine *malloc*(3).

The **mpool_get**() function may fail and set *errno* for the following:

EINVAL The requested record doesn't exist.

The **mpool_new**() and **mpool_get**() functions may fail and set *errno* for any of the errors specified for the library routines *read*(2), *write*(2), and *malloc*(3).

The **mpool_sync**() function may fail and set *errno* for any of the errors specified for the library routine *write*(2).

The **mpool_close**() function may fail and set *errno* for any of the errors specified for the library routine *free*(3).

STANDARDS

BSD.

SEE ALSO

btree(3), *dbopen*(3), *hash*(3), *recno*(3)

NAME

mq_close − close a message queue descriptor

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <mqueue.h>

int mq_close(mqd_t *mqdes***);**

DESCRIPTION

mq_close() closes the message queue descriptor *mqdes*.

If the calling process has attached a notification request (see *mq_notify*(3)) to this message queue via *mqdes*, then this request is removed, and another process can now attach a notification request.

RETURN VALUE

On success **mq_close**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF

The message queue descriptor specified in *mqdes* is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_close()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

All open message queues are automatically closed on process termination, or upon *execve*(2).

SEE ALSO

mq_getattr(3), *mq_notify*(3), *mq_open*(3), *mq_receive*(3), *mq_send*(3), *mq_unlink*(3), *mq_overview*(7)

NAME

mq_getattr, mq_setattr − get/set message queue attributes

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict newattr,
        struct mq_attr *restrict oldattr);
```

DESCRIPTION

mq_getattr() and **mq_setattr**() respectively retrieve and modify attributes of the message queue referred to by the message queue descriptor *mqdes*.

mq_getattr() returns an *mq_attr* structure in the buffer pointed by *attr*. This structure is defined as:

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

The *mq_flags* field contains flags associated with the open message queue description. This field is initialized when the queue is created by *mq_open*(3). The only flag that can appear in this field is **O_NONBLOCK**.

The *mq_maxmsg* and *mq_msgsize* fields are set when the message queue is created by *mq_open*(3). The *mq_maxmsg* field is an upper limit on the number of messages that may be placed on the queue using *mq_send*(3). The *mq_msgsize* field is an upper limit on the size of messages that may be placed on the queue. Both of these fields must have a value greater than zero. Two */proc* files that place ceilings on the values for these fields are described in *mq_overview*(7).

The *mq_curmsgs* field returns the number of messages currently held in the queue.

mq_setattr() sets message queue attributes using information supplied in the *mq_attr* structure pointed to by *newattr*. The only attribute that can be modified is the setting of the **O_NONBLOCK** flag in *mq_flags*. The other fields in *newattr* are ignored. If the *oldattr* field is not NULL, then the buffer that it points to is used to return an *mq_attr* structure that contains the same information that is returned by **mq_getattr**().

RETURN VALUE

On success **mq_getattr**() and **mq_setattr**() return 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF
: The message queue descriptor specified in *mqdes* is invalid.

EINVAL
: *newattr−>mq_flags* contained set bits other than **O_NONBLOCK**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_getattr(), **mq_setattr**()	Thread safety	MT-Safe

VERSIONS

On Linux, **mq_getattr**() and **mq_setattr**() are library functions layered on top of the *mq_getsetattr*(2) system call.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The program below can be used to show the default *mq_maxmsg* and *mq_msgsize* values that are assigned to a message queue that is created with a call to *mq_open*(3) in which the *attr* argument is NULL. Here is an example run of the program:

```
$ ./a.out /testq
Maximum # of messages on queue:   10
Maximum message size:             8192
```

Since Linux 3.5, the following */proc* files (described in *mq_overview*(7)) can be used to control the defaults:

```
$ uname -sr
Linux 3.8.0
$ cat /proc/sys/fs/mqueue/msg_default
10
$ cat /proc/sys/fs/mqueue/msgsize_default
8192
```

Program source

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    mqd_t mqd;
    struct mq_attr attr;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s mq-name\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    mqd = mq_open(argv[1], O_CREAT | O_EXCL, 0600, NULL);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    printf("Maximum # of messages on queue:   %ld\n", attr.mq_maxmsg);
    printf("Maximum message size:             %ld\n", attr.mq_msgsize);

    if (mq_unlink(argv[1]) == -1)
        errExit("mq_unlink");

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

mq_close(3), *mq_notify*(3), *mq_open*(3), *mq_receive*(3), *mq_send*(3), *mq_unlink*(3), *mq_overview*(7)

NAME

mq_notify − register for notification when a message is available

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <mqueue.h>
#include <signal.h>          /* Definition of SIGEV_* constants */

int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

DESCRIPTION

mq_notify() allows the calling process to register or unregister for delivery of an asynchronous notification when a new message arrives on the empty message queue referred to by the message queue descriptor *mqdes*.

The *sevp* argument is a pointer to a *sigevent* structure. For the definition and general details of this structure, see *sigevent*(3type).

If *sevp* is a non-null pointer, then **mq_notify**() registers the calling process to receive message notification. The *sigev_notify* field of the *sigevent* structure to which *sevp* points specifies how notification is to be performed. This field has one of the following values:

SIGEV_NONE
: A "null" notification: the calling process is registered as the target for notification, but when a message arrives, no notification is sent.

SIGEV_SIGNAL
: Notify the process by sending the signal specified in *sigev_signo*. See *sigevent*(3type) for general details. The *si_code* field of the *siginfo_t* structure will be set to **SI_MESGQ**. In addition, *si_pid* will be set to the PID of the process that sent the message, and *si_uid* will be set to the real user ID of the sending process.

SIGEV_THREAD
: Upon message delivery, invoke *sigev_notify_function* as if it were the start function of a new thread. See *sigevent*(3type) for details.

Only one process can be registered to receive notification from a message queue.

If *sevp* is NULL, and the calling process is currently registered to receive notifications for this message queue, then the registration is removed; another process can then register to receive a message notification for this queue.

Message notification occurs only when a new message arrives and the queue was previously empty. If the queue was not empty at the time **mq_notify**() was called, then a notification will occur only after the queue is emptied and a new message arrives.

If another process or thread is waiting to read a message from an empty queue using *mq_receive*(3), then any message notification registration is ignored: the message is delivered to the process or thread calling *mq_receive*(3), and the message notification registration remains in effect.

Notification occurs once: after a notification is delivered, the notification registration is removed, and another process can register for message notification. If the notified process wishes to receive the next notification, it can use **mq_notify**() to request a further notification. This should be done before emptying all unread messages from the queue. (Placing the queue in nonblocking mode is useful for emptying the queue of messages without blocking once it is empty.)

RETURN VALUE

On success **mq_notify**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF
: The message queue descriptor specified in *mqdes* is invalid.

EBUSY
: Another process has already registered to receive notification for this message queue.

EINVAL
sevp->sigev_notify is not one of the permitted values; or *sevp->sigev_notify* is **SIGEV_SIGNAL** and *sevp->sigev_signo* is not a valid signal number.

ENOMEM
Insufficient memory.

POSIX.1-2008 says that an implementation *may* generate an **EINVAL** error if *sevp* is NULL, and the caller is not currently registered to receive notifications for the queue *mqdes*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_notify()	Thread safety	MT-Safe

VERSIONS

C library/kernel differences

In the glibc implementation, the **mq_notify**() library function is implemented on top of the system call of the same name. When *sevp* is NULL, or specifies a notification mechanism other than **SIGEV_THREAD**, the library function directly invokes the system call. For **SIGEV_THREAD**, much of the implementation resides within the library, rather than the kernel. (This is necessarily so, since the thread involved in handling the notification is one that must be managed by the C library POSIX threads implementation.) The implementation involves the use of a raw *netlink*(7) socket and creates a new thread for each notification that is delivered to the process.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The following program registers a notification request for the message queue named in its command-line argument. Notification is performed by creating a thread. The thread executes a function which reads one message from the queue and then terminates the process.

Program source

```
#include <mqueue.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void                     /* Thread start function */
tfunc(union sigval sv)
{
    struct mq_attr attr;
    ssize_t nr;
    void *buf;
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);

    /* Determine max. msg size; allocate buffer to receive msg */

    if (mq_getattr(mqdes, &attr) == -1)
        handle_error("mq_getattr");
    buf = malloc(attr.mq_msgsize);
    if (buf == NULL)
        handle_error("malloc");
```

```
        nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
        if (nr == -1)
            handle_error("mq_receive");

        printf("Read %zd bytes from MQ\n", nr);
        free(buf);
        exit(EXIT_SUCCESS);         /* Terminate the process */
    }

    int
    main(int argc, char *argv[])
    {
        mqd_t mqdes;
        struct sigevent sev;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s <mq-name>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        mqdes = mq_open(argv[1], O_RDONLY);
        if (mqdes == (mqd_t) -1)
            handle_error("mq_open");

        sev.sigev_notify = SIGEV_THREAD;
        sev.sigev_notify_function = tfunc;
        sev.sigev_notify_attributes = NULL;
        sev.sigev_value.sival_ptr = &mqdes;   /* Arg. to thread func. */
        if (mq_notify(mqdes, &sev) == -1)
            handle_error("mq_notify");

        pause();    /* Process will be terminated by thread function */
    }
```

SEE ALSO

mq_close(3), *mq_getattr*(3), *mq_open*(3), *mq_receive*(3), *mq_send*(3), *mq_unlink*(3), *mq_overview*(7), *sigevent*(3type)

NAME

mq_open − open a message queue

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>       /* For mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

DESCRIPTION

mq_open() creates a new POSIX message queue or opens an existing queue. The queue is identified by *name*. For details of the construction of *name*, see *mq_overview*(7).

The *oflag* argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including <*fcntl.h*>.) Exactly one of the following must be specified in *oflag*:

O_RDONLY
: Open the queue to receive messages only.

O_WRONLY
: Open the queue to send messages only.

O_RDWR
: Open the queue to both send and receive messages.

Zero or more of the following flags can additionally be *OR*ed in *oflag*:

O_CLOEXEC (since Linux 2.6.26)
: Set the close-on-exec flag for the message queue descriptor. See *open*(2) for a discussion of why this flag is useful.

O_CREAT
: Create the message queue if it does not exist. The owner (user ID) of the message queue is set to the effective user ID of the calling process. The group ownership (group ID) is set to the effective group ID of the calling process.

O_EXCL
: If **O_CREAT** was specified in *oflag*, and a queue with the given *name* already exists, then fail with the error **EEXIST**.

O_NONBLOCK
: Open the queue in nonblocking mode. In circumstances where *mq_receive*(3) and *mq_send*(3) would normally block, these functions instead fail with the error **EAGAIN**.

If **O_CREAT** is specified in *oflag*, then two additional arguments must be supplied. The *mode* argument specifies the permissions to be placed on the new queue, as for *open*(2). (Symbolic definitions for the permissions bits can be obtained by including <*sys/stat.h*>.) The permissions settings are masked against the process umask.

The fields of the *struct mq_attr* pointed to *attr* specify the maximum number of messages and the maximum size of messages that the queue will allow. This structure is defined as follows:

```
struct mq_attr {
    long mq_flags;       /* Flags (ignored for mq_open()) */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue
                            (ignored for mq_open()) */
};
```

Only the *mq_maxmsg* and *mq_msgsize* fields are employed when calling **mq_open**(); the values in the remaining fields are ignored.

If *attr* is NULL, then the queue is created with implementation-defined default attributes. Since Linux 3.5, two */proc* files can be used to control these defaults; see *mq_overview*(7) for details.

RETURN VALUE

On success, **mq_open**() returns a message queue descriptor for use by other message queue functions. On error, **mq_open**() returns *(mqd_t) −1*, with *errno* set to indicate the error.

ERRORS

EACCES
The queue exists, but the caller does not have permission to open it in the specified mode.

EACCES
name contained more than one slash.

EEXIST
Both **O_CREAT** and **O_EXCL** were specified in *oflag*, but a queue with this *name* already exists.

EINVAL
name doesn't follow the format in *mq_overview*(7).

EINVAL
O_CREAT was specified in *oflag*, and *attr* was not NULL, but *attr−>mq_maxmsg* or *attr−>mq_msqsize* was invalid. Both of these fields must be greater than zero. In a process that is unprivileged (does not have the **CAP_SYS_RESOURCE** capability), *attr−>mq_maxmsg* must be less than or equal to the *msg_max* limit, and *attr−>mq_msgsize* must be less than or equal to the *msgsize_max* limit. In addition, even in a privileged process, *attr−>mq_maxmsg* cannot exceed the **HARD_MAX** limit. (See *mq_overview*(7) for details of these limits.)

EMFILE
The per-process limit on the number of open file and message queue descriptors has been reached (see the description of **RLIMIT_NOFILE** in *getrlimit*(2)).

ENAMETOOLONG
name was too long.

ENFILE
The system-wide limit on the total number of open files and message queues has been reached.

ENOENT
The **O_CREAT** flag was not specified in *oflag*, and no queue with this *name* exists.

ENOENT
name was just "/" followed by no other characters.

ENOMEM
Insufficient memory.

ENOSPC
Insufficient space for the creation of a new message queue. This probably occurred because the *queues_max* limit was encountered; see *mq_overview*(7).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_open()	Thread safety	MT-Safe

VERSIONS

C library/kernel differences

The **mq_open**() library function is implemented on top of a system call of the same name. The library function performs the check that the *name* starts with a slash (/), giving the **EINVAL** error if it does not. The kernel system call expects *name* to contain no preceding slash, so the C library function passes *name* without the preceding slash (i.e., *name+1*) to the system call.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

BUGS

Before Linux 2.6.14, the process umask was not applied to the permissions specified in *mode*.

SEE ALSO

mq_close(3), *mq_getattr*(3), *mq_notify*(3), *mq_receive*(3), *mq_send*(3), *mq_unlink*(3), *mq_overview*(7)

NAME

mq_receive, mq_timedreceive − receive a message from a message queue

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char msg_ptr[.msg_len],
                   size_t msg_len, unsigned int *msg_prio);

#include <time.h>
#include <mqueue.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr[.msg_len],
                        size_t msg_len, unsigned int *restrict msg_prio,
                        const struct timespec *restrict abs_timeout);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mq_timedreceive():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

mq_receive() removes the oldest message with the highest priority from the message queue referred to by the message queue descriptor *mqdes*, and places it in the buffer pointed to by *msg_ptr*. The *msg_len* argument specifies the size of the buffer pointed to by *msg_ptr*; this must be greater than or equal to the *mq_msgsize* attribute of the queue (see *mq_getattr*(3)). If *msg_prio* is not NULL, then the buffer to which it points is used to return the priority associated with the received message.

If the queue is empty, then, by default, **mq_receive**() blocks until a message becomes available, or the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

mq_timedreceive() behaves just like **mq_receive**(), except that if the queue is empty and the **O_NONBLOCK** flag is not enabled for the message queue description, then *abs_timeout* points to a structure which specifies how long the call will block. This value is an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC), specified in a *timespec*(3) structure.

If no message is available, and the timeout has already expired by the time of the call, **mq_timedreceive**() returns immediately.

RETURN VALUE

On success, **mq_receive**() and **mq_timedreceive**() return the number of bytes in the received message; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EAGAIN
: The queue was empty, and the **O_NONBLOCK** flag was set for the message queue description referred to by *mqdes*.

EBADF
: The descriptor specified in *mqdes* was invalid or not opened for reading.

EINTR
: The call was interrupted by a signal handler; see *signal*(7).

EINVAL
: The call would have blocked, and *abs_timeout* was invalid, either because *tv_sec* was less than zero, or because *tv_nsec* was less than zero or greater than 1000 million.

EMSGSIZE
: *msg_len* was less than the *mq_msgsize* attribute of the message queue.

ETIMEDOUT
: The call timed out before a message could be transferred.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_receive(), **mq_timedreceive**()	Thread safety	MT-Safe

VERSIONS

On Linux, **mq_timedreceive**() is a system call, and **mq_receive**() is a library function layered on top of that system call.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

mq_close(3), *mq_getattr*(3), *mq_notify*(3), *mq_open*(3), *mq_send*(3), *mq_unlink*(3), *timespec*(3), *mq_overview*(7), *time*(7)

NAME

mq_send, mq_timedsend − send a message to a message queue

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char msg_ptr[.msg_len],
            size_t msg_len, unsigned int msg_prio);

#include <time.h>
#include <mqueue.h>

int mq_timedsend(mqd_t mqdes, const char msg_ptr[.msg_len],
            size_t msg_len, unsigned int msg_prio,
            const struct timespec *abs_timeout);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

mq_timedsend():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

mq_send() adds the message pointed to by *msg_ptr* to the message queue referred to by the message queue descriptor *mqdes*. The *msg_len* argument specifies the length of the message pointed to by *msg_ptr*; this length must be less than or equal to the queue's *mq_msgsize* attribute. Zero-length messages are allowed.

The *msg_prio* argument is a nonnegative integer that specifies the priority of this message. Messages are placed on the queue in decreasing order of priority, with newer messages of the same priority being placed after older messages with the same priority. See *mq_overview*(7) for details on the range for the message priority.

If the message queue is already full (i.e., the number of messages on the queue equals the queue's *mq_maxmsg* attribute), then, by default, **mq_send**() blocks until sufficient space becomes available to allow the message to be queued, or until the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

mq_timedsend() behaves just like **mq_send**(), except that if the queue is full and the **O_NONBLOCK** flag is not enabled for the message queue description, then *abs_timeout* points to a structure which specifies how long the call will block. This value is an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC), specified in a *timespec*(3) structure.

If the message queue is full, and the timeout has already expired by the time of the call, **mq_timedsend**() returns immediately.

RETURN VALUE

On success, **mq_send**() and **mq_timedsend**() return zero; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EAGAIN
: The queue was full, and the **O_NONBLOCK** flag was set for the message queue description referred to by *mqdes*.

EBADF
: The descriptor specified in *mqdes* was invalid or not opened for writing.

EINTR
: The call was interrupted by a signal handler; see *signal*(7).

EINVAL
: The call would have blocked, and *abs_timeout* was invalid, either because *tv_sec* was less than zero, or because *tv_nsec* was less than zero or greater than 1000 million.

EMSGSIZE
msg_len was greater than the *mq_msgsize* attribute of the message queue.

ETIMEDOUT
The call timed out before a message could be transferred.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_send(), **mq_timedsend**()	Thread safety	MT-Safe

VERSIONS

On Linux, **mq_timedsend**() is a system call, and **mq_send**() is a library function layered on top of that system call.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

mq_close(3), *mq_getattr*(3), *mq_notify*(3), *mq_open*(3), *mq_receive*(3), *mq_unlink*(3), *timespec*(3), *mq_overview*(7), *time*(7)

NAME

mq_unlink − remove a message queue

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

#include <mqueue.h>

int mq_unlink(const char *****name*);**

DESCRIPTION

mq_unlink() removes the specified message queue *name*. The message queue name is removed immediately. The queue itself is destroyed once any other processes that have the queue open close their descriptors referring to the queue.

RETURN VALUE

On success **mq_unlink**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EACCES

The caller does not have permission to unlink this message queue.

ENAMETOOLONG

name was too long.

ENOENT

There is no message queue with the given *name*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mq_unlink()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

mq_close(3), *mq_getattr*(3), *mq_notify*(3), *mq_open*(3), *mq_receive*(3), *mq_send*(3), *mq_overview*(7)

NAME

mtrace, muntrace − malloc tracing

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <mcheck.h>

void mtrace(void);
void muntrace(void);

DESCRIPTION

The **mtrace**() function installs hook functions for the memory-allocation functions (**malloc**(3), *realloc*(3) *memalign*(3), *free*(3)). These hook functions record tracing information about memory allocation and deallocation. The tracing information can be used to discover memory leaks and attempts to free nonallocated memory in a program.

The **muntrace**() function disables the hook functions installed by **mtrace**(), so that tracing information is no longer recorded for the memory-allocation functions. If no hook functions were successfully installed by **mtrace**(), **muntrace**() does nothing.

When **mtrace**() is called, it checks the value of the environment variable **MALLOC_TRACE**, which should contain the pathname of a file in which the tracing information is to be recorded. If the pathname is successfully opened, it is truncated to zero length.

If **MALLOC_TRACE** is not set, or the pathname it specifies is invalid or not writable, then no hook functions are installed, and **mtrace**() has no effect. In set-user-ID and set-group-ID programs, **MALLOC_TRACE** is ignored, and **mtrace**() has no effect.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
mtrace(), **muntrace**()	Thread safety	MT-Unsafe

STANDARDS

GNU.

NOTES

In normal usage, **mtrace**() is called once at the start of execution of a program, and **muntrace**() is never called.

The tracing output produced after a call to **mtrace**() is textual, but not designed to be human readable. The GNU C library provides a Perl script, *mtrace*(1), that interprets the trace log and produces human-readable output. For best results, the traced program should be compiled with debugging enabled, so that line-number information is recorded in the executable.

The tracing performed by **mtrace**() incurs a performance penalty (if **MALLOC_TRACE** points to a valid, writable pathname).

BUGS

The line-number information produced by *mtrace*(1) is not always precise: the line number references may refer to the previous or following (nonblank) line of the source code.

EXAMPLES

The shell session below demonstrates the use of the **mtrace**() function and the *mtrace*(1) command in a program that has memory leaks at two different locations. The demonstration uses the following program:

$ **cat t_mtrace.c**

```
#include <mcheck.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
```

```
    mtrace();

    for (unsigned int j = 0; j < 2; j++)
        malloc(100);            /* Never freed--a memory leak */

    calloc(16, 16);             /* Never freed--a memory leak */
    exit(EXIT_SUCCESS);
}
```

When we run the program as follows, we see that **mtrace**() diagnosed memory leaks at two different locations in the program:

```
$ cc -g t_mtrace.c -o t_mtrace
$ export MALLOC_TRACE=/tmp/t
$ ./t_mtrace
$ mtrace ./t_mtrace $MALLOC_TRACE
Memory not freed:
-----------------
   Address     Size     Caller
0x084c9378     0x64  at /home/cecilia/t_mtrace.c:12
0x084c93e0     0x64  at /home/cecilia/t_mtrace.c:12
0x084c9448    0x100  at /home/cecilia/t_mtrace.c:16
```

The first two messages about unfreed memory correspond to the two *malloc*(3) calls inside the *for* loop. The final message corresponds to the call to *calloc*(3) (which in turn calls *malloc*(3)).

SEE ALSO

mtrace(1), *malloc*(3), *malloc_hook*(3), *mcheck*(3)

NAME

nan, nanf, nanl − return 'Not a Number'

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double nan(const char ****tagp*);**
float nanf(const char ****tagp*);**
long double nanl(const char ****tagp*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nan(), **nanf**(), **nanl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions return a representation (determined by *tagp*) of a quiet NaN. If the implementation does not support quiet NaNs, these functions return zero.

The call *nan("char−sequence")* is equivalent to:

```
strtod("NAN(char-sequence)", NULL);
```

Similarly, calls to **nanf**() and **nanl**() are equivalent to analogous calls to *strtof*(3) and *strtold*(3).

The argument *tagp* is used in an unspecified manner. On IEEE 754 systems, there are many representations of NaN, and *tagp* selects one. On other systems it may do nothing.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nan(), **nanf**(), **nanl**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

See also IEC 559 and the appendix with recommended functions in IEEE 754/IEEE 854.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

isnan(3), *strtod*(3), *math_error*(7)

NAME

netlink − Netlink macros

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <asm/types.h>
#include <linux/netlink.h>

int NLMSG_ALIGN(size_t *len***);**
int NLMSG_LENGTH(size_t *len***);**
int NLMSG_SPACE(size_t *len***);**
void *NLMSG_DATA(struct nlmsghdr *****nlh*);**
struct nlmsghdr *NLMSG_NEXT(struct nlmsghdr *****nlh*, **int** *len***);**
int NLMSG_OK(struct nlmsghdr *****nlh*, **int** *len***);**
int NLMSG_PAYLOAD(struct nlmsghdr *****nlh*, **int** *len***);**

DESCRIPTION

<*linux/netlink.h*> defines several standard macros to access or create a netlink datagram. They are similar in spirit to the macros defined in *cmsg*(3) for auxiliary data. The buffer passed to and from a netlink socket should be accessed using only these macros.

NLMSG_ALIGN()
: Round the length of a netlink message up to align it properly.

NLMSG_LENGTH()
: Given the payload length, *len*, this macro returns the aligned length to store in the *nlmsg_len* field of the *nlmsghdr*.

NLMSG_SPACE()
: Return the number of bytes that a netlink message with payload of *len* would occupy.

NLMSG_DATA()
: Return a pointer to the payload associated with the passed *nlmsghdr*.

NLMSG_NEXT()
: Get the next *nlmsghdr* in a multipart message. The caller must check if the current *nlmsghdr* didn't have the **NLMSG_DONE** set—this function doesn't return NULL on end. The *len* argument is an lvalue containing the remaining length of the message buffer. This macro decrements it by the length of the message header.

NLMSG_OK()
: Return true if the netlink message is not truncated and is in a form suitable for parsing.

NLMSG_PAYLOAD()
: Return the length of the payload associated with the *nlmsghdr*.

VERSIONS

It is often better to use netlink via *libnetlink* than via the low-level kernel interface.

STANDARDS

Linux.

SEE ALSO

libnetlink(3), *netlink*(7)

NAME

newlocale, freelocale − create, modify, and free a locale object

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <locale.h>

locale_t newlocale(int category_mask, const char *locale,
                   locale_t base);
void freelocale(locale_t locobj);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

newlocale(), **freelocale**():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **newlocale**() function creates a new locale object, or modifies an existing object, returning a reference to the new or modified object as the function result. Whether the call creates a new object or modifies an existing object is determined by the value of *base*:

- If *base* is *(locale_t) 0*, a new object is created.
- If *base* refers to valid existing locale object (i.e., an object returned by a previous call to **newlocale**() or *duplocale*(3)), then that object is modified by the call. If the call is successful, the contents of *base* are unspecified (in particular, the object referred to by *base* may be freed, and a new object created). Therefore, the caller should ensure that it stops using *base* before the call to **newlocale**(), and should subsequently refer to the modified object via the reference returned as the function result. If the call fails, the contents of *base* remain valid and unchanged.

If *base* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)), or is not *(locale_t) 0* and is not a valid locale object handle, the behavior is undefined.

The *category_mask* argument is a bit mask that specifies the locale categories that are to be set in a newly created locale object or modified in an existing object. The mask is constructed by a bitwise OR of the constants **LC_ADDRESS_MASK**, **LC_CTYPE_MASK**, **LC_COLLATE_MASK**, **LC_IDENTIFICATION_MASK**, **LC_MEASUREMENT_MASK**, **LC_MESSAGES_MASK**, **LC_MONETARY_MASK**, **LC_NUMERIC_MASK**, **LC_NAME_MASK**, **LC_PAPER_MASK**, **LC_TELEPHONE_MASK**, and **LC_TIME_MASK**. Alternatively, the mask can be specified as **LC_ALL_MASK**, which is equivalent to ORing all of the preceding constants.

For each category specified in *category_mask*, the locale data from *locale* will be used in the object returned by **newlocale**(). If a new locale object is being created, data for all categories not specified in *category_mask* is taken from the default ("POSIX") locale.

The following preset values of *locale* are defined for all categories that can be specified in *category_mask*:

"POSIX"
: A minimal locale environment for C language programs.

"C"
: Equivalent to "POSIX".

""
: An implementation-defined native environment corresponding to the values of the **LC_*** and **LANG** environment variables (see *locale*(7)).

freelocale()

The **freelocale**() function deallocates the resources associated with *locobj*, a locale object previously returned by a call to **newlocale**() or *duplocale*(3). If *locobj* is **LC_GLOBAL_LOCALE** or is not valid locale object handle, the results are undefined.

Once a locale object has been freed, the program should make no further use of it.

RETURN VALUE

On success, **newlocale**() returns a handle that can be used in calls to *duplocale*(3), **freelocale**(), and other functions that take a *locale_t* argument. On error, **newlocale**() returns *(locale_t) 0*, and sets *errno* to indicate the error.

ERRORS

EINVAL
One or more bits in *category_mask* do not correspond to a valid locale category.

EINVAL
locale is NULL.

ENOENT
locale is not a string pointer referring to a valid locale.

ENOMEM
Insufficient memory to create a locale object.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.3.

NOTES

Each locale object created by **newlocale**() should be deallocated using **freelocale**().

EXAMPLES

The program below takes up to two command-line arguments, which each identify locales. The first argument is required, and is used to set the **LC_NUMERIC** category in a locale object created using **newlocale**(). The second command-line argument is optional; if it is present, it is used to set the **LC_TIME** category of the locale object.

Having created and initialized the locale object, the program then applies it using *uselocale*(3), and then tests the effect of the locale changes by:

(1) Displaying a floating-point number with a fractional part. This output will be affected by the **LC_NUMERIC** setting. In many European-language locales, the fractional part of the number is separated from the integer part using a comma, rather than a period.

(2) Displaying the date. The format and language of the output will be affected by the **LC_TIME** setting.

The following shell sessions show some example runs of this program.

Set the **LC_NUMERIC** category to *fr_FR* (French):

```
$ ./a.out fr_FR
123456,789
Fri Mar  7 00:25:08 2014
```

Set the **LC_NUMERIC** category to *fr_FR* (French), and the **LC_TIME** category to *it_IT* (Italian):

```
$ ./a.out fr_FR it_IT
123456,789
ven 07 mar 2014 00:26:01 CET
```

Specify the **LC_TIME** setting as an empty string, which causes the value to be taken from environment variable settings (which, here, specify *mi_NZ*, New Zealand Māori):

```
$ LC_ALL=mi_NZ ./a.out fr_FR ""
123456,789
Te Paraire, te 07 o Poutū-te-rangi, 2014 00:38:44 CET
```

Program source

```
#define _XOPEN_SOURCE 700
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    char buf[100];
    time_t t;
    size_t s;
    struct tm *tm;
    locale_t loc, nloc;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s locale1 [locale2]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Create a new locale object, taking the LC_NUMERIC settings
       from the locale specified in argv[1]. */

    loc = newlocale(LC_NUMERIC_MASK, argv[1], (locale_t) 0);
    if (loc == (locale_t) 0)
        errExit("newlocale");

    /* If a second command-line argument was specified, modify the
       locale object to take the LC_TIME settings from the locale
       specified in argv[2]. We assign the result of this newlocale()
       call to 'nloc' rather than 'loc', since in some cases, we might
       want to preserve 'loc' if this call fails. */

    if (argc > 2) {
        nloc = newlocale(LC_TIME_MASK, argv[2], loc);
        if (nloc == (locale_t) 0)
            errExit("newlocale");
        loc = nloc;
    }

    /* Apply the newly created locale to this thread. */

    uselocale(loc);

    /* Test effect of LC_NUMERIC. */

    printf("%8.3f\n", 123456.789);

    /* Test effect of LC_TIME. */

    t = time(NULL);
    tm = localtime(&t);
    if (tm == NULL)
        errExit("time");

    s = strftime(buf, sizeof(buf), "%c", tm);
    if (s == 0)
        errExit("strftime");

    printf("%s\n", buf);
```

```
        /* Free the locale object. */

        uselocale(LC_GLOBAL_LOCALE);    /* So 'loc' is no longer in use */
        freelocale(loc);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

locale(1), *duplocale*(3), *setlocale*(3), *uselocale*(3), *locale*(5), *locale*(7)

NAME

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl − floating-point number manipulation

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nextafter():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

nextafterf(), **nextafterl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

nexttoward(), **nexttowardf**(), **nexttowardl**():
_XOPEN_SOURCE >= 600 || _ISOC99_SOURCE
|| _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **nextafter**(), **nextafterf**(), and **nextafterl**() functions return the next representable floating-point value following *x* in the direction of *y*. If *y* is less than *x*, these functions will return the largest representable number less than *x*.

If *x* equals *y*, the functions return *y*.

The **nexttoward**(), **nexttowardf**(), and **nexttowardl**() functions do the same as the corresponding **nextafter**() functions, except that they have a *long double* second argument.

RETURN VALUE

On success, these functions return the next representable floating-point value after *x* in the direction of *y*.

If *x* equals *y*, then *y* (cast to the same type as *x*) is returned.

If *x* or *y* is a NaN, a NaN is returned.

If *x* is finite, and the result would overflow, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the correct mathematical sign.

If *x* is not equal to *y*, and the correct function result would be subnormal, zero, or underflow, a range error occurs, and either the correct value (if it can be represented), or 0.0, is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result overflow
errno is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error: result is subnormal or underflows
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nextafter(), **nextafterf**(), **nextafterl**(), **nexttoward**(), **nexttowardf**(), **nexttowardl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

HISTORY

C99, POSIX.1-2001.

BUGS

In glibc 2.5 and earlier, these functions do not raise an underflow floating-point (**FE_UNDERFLOW**) exception when an underflow occurs.

Before glibc 2.23 these functions did not set *errno*.

SEE ALSO

nearbyint(3)

NAME

nextup, nextupf, nextupl, nextdown, nextdownf, nextdownl − return next floating-point number toward positive/negative infinity

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#define _GNU_SOURCE     /* See feature_test_macros(7) */
#include <math.h>

double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);

double nextdown(double x);
float nextdownf(float x);
long double nextdownl(long double x);
```

DESCRIPTION

The **nextup**(), **nextupf**(), and **nextupl**() functions return the next representable floating-point number greater than *x*.

If *x* is the smallest representable negative number in the corresponding type, these functions return −0. If *x* is 0, the returned value is the smallest representable positive number of the corresponding type.

If *x* is positive infinity, the returned value is positive infinity. If *x* is negative infinity, the returned value is the largest representable finite negative number of the corresponding type.

If *x* is Nan, the returned value is NaN.

The value returned by *nextdown(x)* is *−nextup(−x)*, and similarly for the other types.

RETURN VALUE

See DESCRIPTION.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nextup(), **nextupf**(), **nextupl**(), **nextdown**(), **nextdownf**(), **nextdownl**()	Thread safety	MT-Safe

STANDARDS

These functions are described in *IEEE Std 754-2008 - Standard for Floating-Point Arithmetic* and *ISO/IEC TS 18661*.

HISTORY

glibc 2.24.

SEE ALSO

nearbyint(3), *nextafter*(3)

NAME

nl_langinfo, nl_langinfo_l − query language and locale information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <langinfo.h>

char *nl_langinfo(nl_item *item***);**
char *nl_langinfo_l(nl_item *item***, locale_t** *locale***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nl_langinfo_l():
Since glibc 2.24:
_POSIX_C_SOURCE >= 200809L
glibc 2.23 and earlier:
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **nl_langinfo**() and **nl_langinfo_l**() functions provide access to locale information in a more flexible way than *localeconv*(3). **nl_langinfo**() returns a string which is the value corresponding to *item* in the program's current global locale. **nl_langinfo_l**() returns a string which is the value corresponding to *item* for the locale identified by the locale object *locale*, which was previously created by *newlocale*(3). Individual and additional elements of the locale categories can be queried.

Examples for the locale elements that can be specified in *item* using the constants defined in <*langinfo.h*> are:

CODESET (LC_CTYPE)
Return a string with the name of the character encoding used in the selected locale, such as "UTF−8", "ISO−8859−1", or "ANSI_X3.4−1968" (better known as US-ASCII). This is the same string that you get with "locale charmap". For a list of character encoding names, try "locale −m" (see *locale*(1)).

D_T_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) to represent time and date in a locale-specific way (**%c** conversion specification).

D_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) to represent a date in a locale-specific way (**%x** conversion specification).

T_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) to represent a time in a locale-specific way (**%X** conversion specification).

AM_STR (LC_TIME)
Return a string that represents affix for ante meridiem (before noon, "AM") time. (Used in **%p** *strftime*(3) conversion specification.)

PM_STR (LC_TIME)
Return a string that represents affix for post meridiem (before midnight, "PM") time. (Used in **%p** *strftime*(3) conversion specification.)

T_FMT_AMPM (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) to represent a time in a.m. or p.m. notation in a locale-specific way (**%r** conversion specification).

ERA (LC_TIME)
Return era description, which contains information about how years are counted and displayed for each era in a locale. Each era description segment shall have the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below:

direction Either a "+" or a "-" character. The "+" means that years increase from the *start_date* towards the *end_date*, " -" means the opposite.

offset The epoch year of the *start_date*.

start_date A date in the form *yyyy*/*mm*/*dd*, where *yyyy*, *mm*, and *dd* are the year, month, and day numbers respectively of the start of the era.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values "**-***" (minus infinity) or "**+***" (plus infinity).

era_name The name of the era, corresponding to the **%EC** *strftime*(3) conversion specification.

era_format The format of the year in the era, corresponding to the **%EY** *strftime*(3) conversion specification.

Era description segments are separated by semicolons. Most locales do not define this value. Examples of locales that do define this value are the Japanese and Thai locales.

ERA_D_T_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) for alternative representation of time and date in a locale-specific way (**%Ec** conversion specification).

ERA_D_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) for alternative representation of a date in a locale-specific way (**%Ex** conversion specification).

ERA_T_FMT (LC_TIME)
Return a string that can be used as a format string for *strftime*(3) for alternative representation of a time in a locale-specific way (**%EX** conversion specification).

DAY_{1–7} (LC_TIME)
Return name of the *n*-th day of the week. [Warning: this follows the US convention DAY_1 = Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.] (Used in **%A** *strftime*(3) conversion specification.)

ABDAY_{1–7} (LC_TIME)
Return abbreviated name of the *n*-th day of the week. (Used in **%a** *strftime*(3) conversion specification.)

MON_{1–12} (LC_TIME)
Return name of the *n*-th month. (Used in **%B** *strftime*(3) conversion specification.)

ABMON_{1–12} (LC_TIME)
Return abbreviated name of the *n*-th month. (Used in **%b** *strftime*(3) conversion specification.)

RADIXCHAR (LC_NUMERIC)
Return radix character (decimal dot, decimal comma, etc.).

THOUSEP (LC_NUMERIC)
Return separator character for thousands (groups of three digits).

YESEXPR (LC_MESSAGES)
Return a regular expression that can be used with the *regex*(3) function to recognize a positive response to a yes/no question.

NOEXPR (LC_MESSAGES)
Return a regular expression that can be used with the *regex*(3) function to recognize a negative response to a yes/no question.

CRNCYSTR (LC_MONETARY)
Return the currency symbol, preceded by "−" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

The above list covers just some examples of items that can be requested. For a more detailed list, consult *The GNU C Library Reference Manual*.

RETURN VALUE

On success, these functions return a pointer to a string which is the value corresponding to *item* in the specified locale.

If no locale has been selected by *setlocale*(3) for the appropriate category, **nl_langinfo**() return a pointer to the corresponding string in the "C" locale. The same is true of **nl_langinfo_l**() if *locale* specifies a locale where *langinfo* data is not defined.

If *item* is not valid, a pointer to an empty string is returned.

The pointer returned by these functions may point to static data that may be overwritten, or the pointer itself may be invalidated, by a subsequent call to **nl_langinfo**(), **nl_langinfo_l**(), or *setlocale*(3). The same statements apply to **nl_langinfo_l**() if the locale object referred to by *locale* is freed or modified by *freelocale*(3) or *newlocale*(3).

POSIX specifies that the application may not modify the string returned by these functions.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nl_langinfo()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SUSv2.

NOTES

The behavior of **nl_langinfo_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** or is not a valid locale object handle.

EXAMPLES

The following program sets the character type and the numeric locale according to the environment and queries the terminal character set and the radix character.

```
#include <langinfo.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    setlocale(LC_CTYPE, "");
    setlocale(LC_NUMERIC, "");

    printf("%s\n", nl_langinfo(CODESET));
    printf("%s\n", nl_langinfo(RADIXCHAR));

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

locale(1), *localeconv*(3), *setlocale*(3), *charsets*(7), *locale*(7)

The GNU C Library Reference Manual

NAME

ntp_gettime, ntp_gettimex − get time parameters (NTP daemon interface)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/timex.h>

int ntp_gettime(struct ntptimeval ****ntv*);**
int ntp_gettimex(struct ntptimeval ****ntv*);**

DESCRIPTION

Both of these APIs return information to the caller via the *ntv* argument, a structure of the following type:

```
struct ntptimeval {
    struct timeval time;    /* Current time */
    long maxerror;          /* Maximum error */
    long esterror;          /* Estimated error */
    long tai;               /* TAI offset */

    /* Further padding bytes allowing for future expansion */
};
```

The fields of this structure are as follows:

time The current time, expressed as a *timeval* structure:

```
struct timeval {
    time_t      tv_sec;   /* Seconds since the Epoch */
    suseconds_t tv_usec;  /* Microseconds */
};
```

maxerror
: Maximum error, in microseconds. This value can be initialized by *ntp_adjtime*(3), and is increased periodically (on Linux: each second), but is clamped to an upper limit (the kernel constant **NTP_PHASE_MAX**, with a value of 16,000).

esterror
: Estimated error, in microseconds. This value can be set via *ntp_adjtime*(3) to contain an estimate of the difference between the system clock and the true time. This value is not used inside the kernel.

tai TAI (Atomic International Time) offset.

ntp_gettime() returns an *ntptimeval* structure in which the *time*, *maxerror*, and *esterror* fields are filled in.

ntp_gettimex() performs the same task as **ntp_gettime**(), but also returns information in the *tai* field.

RETURN VALUE

The return values for **ntp_gettime**() and **ntp_gettimex**() are as for *adjtimex*(2). Given a correct pointer argument, these functions always succeed.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ntp_gettime(), **ntp_gettimex**()	Thread safety	MT-Safe

STANDARDS

ntp_gettime()
: NTP Kernel Application Program Interface.

ntp_gettimex()
: GNU.

HISTORY

ntp_gettime()
glibc 2.1.

ntp_gettimex()
glibc 2.12.

SEE ALSO

adjtimex(2), *ntp_adjtime*(3), *time*(7)

NTP "Kernel Application Program Interface"

NAME

offsetof − offset of a structure member

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stddef.h>

size_t offsetof(*type***,** *member***);**

DESCRIPTION

The macro **offsetof**() returns the offset of the field *member* from the start of the structure *type*.

This macro is useful because the sizes of the fields that compose a structure can vary across implementations, and compilers may insert different numbers of padding bytes between fields. Consequently, an element's offset is not necessarily given by the sum of the sizes of the previous elements.

A compiler error will result if *member* is not aligned to a byte boundary (i.e., it is a bit field).

RETURN VALUE

offsetof() returns the offset of the given *member* within the given *type*, in units of bytes.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

EXAMPLES

On a Linux/i386 system, when compiled using the default *gcc*(1) options, the program below produces the following output:

```
$ ./a.out
offsets: i=0; c=4; d=8 a=16
sizeof(struct s)=16
```

Program source

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    struct s {
        int i;
        char c;
        double d;
        char a[];
    };

    /* Output is compiler dependent */

    printf("offsets: i=%zu; c=%zu; d=%zu a=%zu\n",
           offsetof(struct s, i), offsetof(struct s, c),
           offsetof(struct s, d), offsetof(struct s, a));
    printf("sizeof(struct s)=%zu\n", sizeof(struct s));

    exit(EXIT_SUCCESS);
}
```

NAME

on_exit − register a function to be called at normal process termination

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int on_exit(void (* *function*)(int, void *), void **arg*);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

on_exit():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **on_exit**() function registers the given *function* to be called at normal process termination, whether via *exit*(3) or via return from the program's *main*(). The *function* is passed the status argument given to the last call to *exit*(3) and the *arg* argument from **on_exit**().

The same function may be registered multiple times: it is called once for each registration.

When a child process is created via *fork*(2), it inherits copies of its parent's registrations. Upon a successful call to one of the *exec*(3) functions, all registrations are removed.

RETURN VALUE

The **on_exit**() function returns the value 0 if successful; otherwise it returns a nonzero value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
on_exit()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

SunOS 4, glibc. Removed in Solaris (SunOS 5). Use the standard *atexit*(3) instead.

CAVEATS

By the time *function* is executed, stack (*auto*) variables may already have gone out of scope. Therefore, *arg* should not be a pointer to a stack variable; it may however be a pointer to a heap variable or a global variable.

SEE ALSO

_exit(2), *atexit*(3), *exit*(3)

NAME

open_memstream, open_wmemstream − open a dynamic memory buffer stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

FILE *open_memstream(char **ptr, size_t *sizeloc);

#include <wchar.h>

FILE *open_wmemstream(wchar_t **ptr, size_t *sizeloc);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

open_memstream(), **open_wmemstream**():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **open_memstream**() function opens a stream for writing to a memory buffer. The function dynamically allocates the buffer, and the buffer automatically grows as needed. Initially, the buffer has a size of zero. After closing the stream, the caller should *free*(3) this buffer.

The locations pointed to by *ptr* and *sizeloc* are used to report, respectively, the current location and the size of the buffer. The locations referred to by these pointers are updated each time the stream is flushed (**fflush**(3)) and when the stream is closed (**fclose**(3)). These values remain valid only as long as the caller performs no further output on the stream. If further output is performed, then the stream must again be flushed before trying to access these values.

A null byte is maintained at the end of the buffer. This byte is *not* included in the size value stored at *sizeloc*.

The stream maintains the notion of a current position, which is initially zero (the start of the buffer). Each write operation implicitly adjusts the buffer position. The stream's buffer position can be explicitly changed with *fseek*(3) or *fseeko*(3). Moving the buffer position past the end of the data already written fills the intervening space with null characters.

The **open_wmemstream**() is similar to **open_memstream**(), but operates on wide characters instead of bytes.

RETURN VALUE

Upon successful completion, **open_memstream**() and **open_wmemstream**() return a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
open_memstream(), **open_wmemstream**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

open_memstream()
glibc 1.0.x.

open_wmemstream()
glibc 2.4.

NOTES

There is no file descriptor associated with the file stream returned by these functions (i.e., *fileno*(3) will return an error if called on the returned stream).

BUGS

Before glibc 2.7, seeking past the end of a stream created by **open_memstream**() does not enlarge the buffer; instead the *fseek*(3) call fails, returning −1.

EXAMPLES

See *fmemopen*(3).

SEE ALSO

fmemopen(3), *fopen*(3), *setbuf*(3)

NAME

opendir, fdopendir − open a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

fdopendir():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE

The **opendir**() and **fdopendir**() functions return a pointer to the directory stream. On error, NULL is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
Permission denied.

EBADF
fd is not a valid file descriptor opened for reading.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
Directory does not exist, or *name* is an empty string.

ENOMEM
Insufficient memory to complete the operation.

ENOTDIR
name is not a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
opendir(), **fdopendir**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

STANDARDS

opendir()
SVr4, 4.3BSD, POSIX.1-2001.

fdopendir()
POSIX.1-2008. glibc 2.4.

NOTES

Filename entries can be read from a directory stream using *readdir*(3).

The underlying file descriptor of the directory stream can be obtained using *dirfd*(3).

The **opendir**() function sets the close-on-exec flag for the file descriptor underlying the *DIR* *. The **fdopendir**() function leaves the setting of the close-on-exec flag unchanged for the file descriptor, *fd*. POSIX.1-200x leaves it unspecified whether a successful call to **fdopendir**() will set the close-on-exec flag for the file descriptor, *fd*.

SEE ALSO

open(2), *closedir*(3), *dirfd*(3), *readdir*(3), *rewinddir*(3), *scandir*(3), *seekdir*(3), *telldir*(3)

NAME

openpty, login_tty, forkpty − terminal utility functions

LIBRARY

System utilities library (*libutil*, −*lutil*)

SYNOPSIS

```
#include <pty.h>

int openpty(int *amaster, int *aslave, char *name,
        const struct termios *termp,
        const struct winsize *winp);
pid_t forkpty(int *amaster, char *name,
        const struct termios *termp,
        const struct winsize *winp);

#include <utmp.h>

int login_tty(int fd);
```

DESCRIPTION

The **openpty**() function finds an available pseudoterminal and returns file descriptors for the master and slave in *amaster* and *aslave*. If *name* is not NULL, the filename of the slave is returned in *name*. If *termp* is not NULL, the terminal parameters of the slave will be set to the values in *termp*. If *winp* is not NULL, the window size of the slave will be set to the values in *winp*.

The **login_tty**() function prepares for a login on the terminal referred to by the file descriptor *fd* (which may be a real terminal device, or the slave of a pseudoterminal as returned by *openpty*()) by creating a new session, making *fd* the controlling terminal for the calling process, setting *fd* to be the standard input, output, and error streams of the current process, and closing *fd*.

The **forkpty**() function combines **openpty**(), *fork*(2), and **login_tty**() to create a new process operating in a pseudoterminal. A file descriptor referring to master side of the pseudoterminal is returned in *amaster*. If *name* is not NULL, the buffer it points to is used to return the filename of the slave. The *termp* and *winp* arguments, if not NULL, will determine the terminal attributes and window size of the slave side of the pseudoterminal.

RETURN VALUE

If a call to **openpty**(), **login_tty**(), or **forkpty**() is not successful, −1 is returned and *errno* is set to indicate the error. Otherwise, **openpty**(), **login_tty**(), and the child process of **forkpty**() return 0, and the parent process of **forkpty**() returns the process ID of the child process.

ERRORS

openpty() fails if:

ENOENT
There are no available terminals.

login_tty() fails if *ioctl*(2) fails to set *fd* to the controlling terminal of the calling process.

forkpty() fails if either **openpty**() or *fork*(2) fails.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
forkpty(), **openpty**()	Thread safety	MT-Safe locale
login_tty()	Thread safety	MT-Unsafe race:ttyname

STANDARDS

BSD.

HISTORY

The **const** modifiers were added to the structure pointer arguments of **openpty**() and **forkpty**() in glibc 2.8.

Before glibc 2.0.92, **openpty**() returns file descriptors for a BSD pseudoterminal pair; since glibc 2.0.92, it first attempts to open a UNIX 98 pseudoterminal pair, and falls back to opening a BSD pseudoterminal pair if that fails.

BUGS

Nobody knows how much space should be reserved for *name*. So, calling **openpty**() or **forkpty**() with non-NULL *name* may not be secure.

SEE ALSO

fork(2), *ttyname*(3), *pty*(7)

NAME

perror − print a system error message

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

int errno;      /* Not really declared this way; see errno(3) */

[[deprecated]] const char *const sys_errlist[];
[[deprecated]] int sys_nerr;
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sys_errlist, *sys_nerr*:
From glibc 2.19 to glibc 2.31:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **perror**() function produces a message on standard error describing the last error encountered during a call to a system or library function.

First (if *s* is not NULL and **s* is not a null byte ('\0')), the argument string *s* is printed, followed by a colon and a blank. Then an error message corresponding to the current value of *errno* and a new-line.

To be of most use, the argument string should include the name of the function that incurred the error.

The global error list *sys_errlist*[], which can be indexed by *errno*, can be used to obtain the error message without the newline. The largest message number provided in the table is *sys_nerr*−1. Be careful when directly accessing this list, because new error values may not have been added to *sys_errlist*[]. The use of *sys_errlist*[] is nowadays deprecated; use *strerror*(3) instead.

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong. (These values can be found in <*errno.h*>.) Many library functions do likewise. The function **perror**() serves to translate this error code into human-readable form. Note that *errno* is undefined after a successful system call or library function call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to **perror**(), the value of *errno* should be saved.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
perror()	Thread safety	MT-Safe race:stderr

STANDARDS

errno
perror()
C11, POSIX.1-2008.

sys_nerr
sys_errlist
BSD.

HISTORY

errno
perror()
POSIX.1-2001, C89, 4.3BSD.

sys_nerr
sys_errlist
Removed in glibc 2.32.

SEE ALSO

err(3), *errno*(3), *error*(3), *strerror*(3)

NAME

popen, pclose − pipe stream to or from a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

popen(), **pclose**():
_POSIX_C_SOURCE >= 2
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **popen**() function opens a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the *type* argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

The *command* argument is a pointer to a null-terminated string containing a shell command line. This command is passed to */bin/sh* using the **−c** flag; interpretation, if any, is performed by the shell.

The *type* argument is a pointer to a null-terminated string which must contain either the letter 'r' for reading or the letter 'w' for writing. Since glibc 2.9, this argument can additionally include the letter 'e', which causes the close-on-exec flag (**FD_CLOEXEC**) to be set on the underlying file descriptor; see the description of the **O_CLOEXEC** flag in *open*(2) for reasons why this may be useful.

The return value from **popen**() is a normal standard I/O stream in all respects save that it must be closed with **pclose**() rather than *fclose*(3). Writing to such a stream writes to the standard input of the command; the command's standard output is the same as that of the process that called **popen**(), unless this is altered by the command itself. Conversely, reading from the stream reads the command's standard output, and the command's standard input is the same as that of the process that called **popen**().

Note that output **popen**() streams are block buffered by default.

The **pclose**() function waits for the associated process to terminate and returns the exit status of the command as returned by *wait4*(2).

RETURN VALUE

popen(): on success, returns a pointer to an open stream that can be used to read or write to the pipe; if the *fork*(2) or *pipe*(2) calls fail, or if the function cannot allocate memory, NULL is returned.

pclose(): on success, returns the exit status of the command; if *wait4*(2) returns an error, or some other error is detected, −1 is returned.

On failure, both functions set *errno* to indicate the error.

ERRORS

The **popen**() function does not set *errno* if memory allocation fails. If the underlying *fork*(2) or *pipe*(2) fails, *errno* is set to indicate the error. If the *type* argument is invalid, and this condition is detected, *errno* is set to **EINVAL**.

If **pclose**() cannot obtain the child status, *errno* is set to **ECHILD**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
popen(), **pclose**()	Thread safety	MT-Safe

VERSIONS

The 'e' value for *type* is a Linux extension.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

CAVEATS

Carefully read Caveats in *system*(3).

BUGS

Since the standard input of a command opened for reading shares its seek offset with the process that called **popen**(), if the original process has done a buffered read, the command's input position may not be as expected. Similarly, the output from a command opened for writing may become intermingled with that of the original process. The latter can be avoided by calling *fflush*(3) before **popen**().

Failure to execute the shell is indistinguishable from the shell's failure to execute the command, or an immediate exit of the command. The only hint is an exit status of 127.

SEE ALSO

sh(1), *fork*(2), *pipe*(2), *wait4*(2), *fclose*(3), *fflush*(3), *fopen*(3), *stdio*(3), *system*(3)

NAME

posix_fallocate − allocate file space

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, off_t len);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

posix_fallocate():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The function **posix_fallocate**() ensures that disk space is allocated for the file referred to by the file descriptor *fd* for the bytes in the range starting at *offset* and continuing for *len* bytes. After a successful call to **posix_fallocate**(), subsequent writes to bytes in the specified range are guaranteed not to fail because of lack of disk space.

If the size of the file is less than *offset*+*len*, then the file is increased to this size; otherwise the file size is left unchanged.

RETURN VALUE

posix_fallocate() returns zero on success, or an error number on failure. Note that *errno* is not set.

ERRORS

EBADF
fd is not a valid file descriptor, or is not opened for writing.

EFBIG
offset+*len* exceeds the maximum file size.

EINTR
A signal was caught during execution.

EINVAL
offset was less than 0, or *len* was less than or equal to 0, or the underlying filesystem does not support the operation.

ENODEV
fd does not refer to a regular file.

ENOSPC
There is not enough space left on the device containing the file referred to by *fd*.

EOPNOTSUPP
The filesystem containing the file referred to by *fd* does not support this operation. This error code can be returned by C libraries that don't perform the emulation shown in NOTES, such as musl libc.

ESPIPE
fd refers to a pipe.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
posix_fallocate()	Thread safety	MT-Safe (but see NOTES)

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1.94. POSIX.1-2001

POSIX.1-2008 says that an implementation *shall* give the **EINVAL** error if *len* was 0, or *offset* was less than 0. POSIX.1-2001 says that an implementation *shall* give the **EINVAL** error if *len* is less than 0, or *offset* was less than 0, and *may* give the error if *len* equals zero.

CAVEATS

In the glibc implementation, **posix_fallocate**() is implemented using the *fallocate*(2) system call, which is MT-safe. If the underlying filesystem does not support *fallocate*(2), then the operation is emulated with the following caveats:

- The emulation is inefficient.
- There is a race condition where concurrent writes from another thread or process could be overwritten with null bytes.
- There is a race condition where concurrent file size increases by another thread or process could result in a file whose size is smaller than expected.
- If *fd* has been opened with the **O_APPEND** or **O_WRONLY** flags, the function fails with the error **EBADF**.

In general, the emulation is not MT-safe. On Linux, applications may use *fallocate*(2) if they cannot tolerate the emulation caveats. In general, this is only recommended if the application plans to terminate the operation if **EOPNOTSUPP** is returned, otherwise the application itself will need to implement a fallback with all the same problems as the emulation provided by glibc.

SEE ALSO

fallocate(1), *fallocate*(2), *lseek*(2), *posix_fadvise*(2)

NAME

posix_madvise − give advice about patterns of memory usage

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/mman.h>

int posix_madvise(void *addr***[.***len***], size_t** *len***, int** *advice***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

posix_madvise():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **posix_madvise**() function allows an application to advise the system about its expected patterns of usage of memory in the address range starting at *addr* and continuing for *len* bytes. The system is free to use this advice in order to improve the performance of memory accesses (or to ignore the advice altogether), but calling **posix_madvise**() shall not affect the semantics of access to memory in the specified range.

The *advice* argument is one of the following:

POSIX_MADV_NORMAL
The application has no special advice regarding its memory usage patterns for the specified address range. This is the default behavior.

POSIX_MADV_SEQUENTIAL
The application expects to access the specified address range sequentially, running from lower addresses to higher addresses. Hence, pages in this region can be aggressively read ahead, and may be freed soon after they are accessed.

POSIX_MADV_RANDOM
The application expects to access the specified address range randomly. Thus, read ahead may be less useful than normally.

POSIX_MADV_WILLNEED
The application expects to access the specified address range in the near future. Thus, read ahead may be beneficial.

POSIX_MADV_DONTNEED
The application expects that it will not access the specified address range in the near future.

RETURN VALUE

On success, **posix_madvise**() returns 0. On failure, it returns a positive error number.

ERRORS

EINVAL
addr is not a multiple of the system page size or *len* is negative.

EINVAL
advice is invalid.

ENOMEM
Addresses in the specified range are partially or completely outside the caller's address space.

VERSIONS

POSIX.1 permits an implementation to generate an error if *len* is 0. On Linux, specifying *len* as 0 is permitted (as a successful no-op).

In glibc, this function is implemented using *madvise*(2). However, since glibc 2.6, **POSIX_MADV_DONTNEED** is treated as a no-op, because the corresponding *madvise*(2) value, **MADV_DONTNEED**, has destructive semantics.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

SEE ALSO

madvise(2), *posix_fadvise*(2)

NAME

posix_memalign, aligned_alloc, memalign, valloc, pvalloc − allocate aligned memory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
[[deprecated]] void *valloc(size_t size);

#include <malloc.h>

[[deprecated]] void *memalign(size_t alignment, size_t size);
[[deprecated]] void *pvalloc(size_t size);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

posix_memalign():
_POSIX_C_SOURCE >= 200112L

aligned_alloc():
_ISOC11_SOURCE

valloc():
Since glibc 2.12:
(_XOPEN_SOURCE >= 500) && !(_POSIX_C_SOURCE >= 200112L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

posix_memalign() allocates *size* bytes and places the address of the allocated memory in **memptr*. The address of the allocated memory will be a multiple of *alignment*, which must be a power of two and a multiple of *sizeof(void *)*. This address can later be successfully passed to *free*(3). If *size* is 0, then the value placed in **memptr* is either NULL or a unique pointer value.

The obsolete function **memalign**() allocates *size* bytes and returns a pointer to the allocated memory. The memory address will be a multiple of *alignment*, which must be a power of two.

aligned_alloc() is the same as **memalign**(), except for the added restriction that *alignment* must be a power of two.

The obsolete function **valloc**() allocates *size* bytes and returns a pointer to the allocated memory. The memory address will be a multiple of the page size. It is equivalent to *memalign(sysconf(_SC_PAGESIZE),size)*.

The obsolete function **pvalloc**() is similar to **valloc**(), but rounds the size of the allocation up to the next multiple of the system page size.

For all of these functions, the memory is not zeroed.

RETURN VALUE

aligned_alloc(), **memalign**(), **valloc**(), and **pvalloc**() return a pointer to the allocated memory on success. On error, NULL is returned, and *errno* is set to indicate the error.

posix_memalign() returns zero on success, or one of the error values listed in the next section on failure. The value of *errno* is not set. On Linux (and other systems), **posix_memalign**() does not modify *memptr* on failure. A requirement standardizing this behavior was added in POSIX.1-2008 TC2.

ERRORS

EINVAL
The *alignment* argument was not a power of two, or was not a multiple of *sizeof(void *)*.

ENOMEM
Out of memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
aligned_alloc(), **memalign**(), **posix_memalign**()	Thread safety	MT-Safe
valloc(), **pvalloc**()	Thread safety	MT-Unsafe init

STANDARDS

aligned_alloc()
: C11.

posix_memalign()
: POSIX.1-2008.

memalign()
valloc()
: None.

pvalloc()
: GNU.

HISTORY

aligned_alloc()
: glibc 2.16. C11.

posix_memalign()
: glibc 2.1.91. POSIX.1d, POSIX.1-2001.

memalign()
: glibc 2.0. SunOS 4.1.3.

valloc()
: glibc 2.0. 3.0BSD. Documented as obsolete in 4.3BSD, and as legacy in SUSv2.

pvalloc()
: glibc 2.0.

Headers

Everybody agrees that **posix_memalign**() is declared in <*stdlib.h*>.

On some systems **memalign**() is declared in <*stdlib.h*> instead of <*malloc.h*>.

According to SUSv2, **valloc**() is declared in <*stdlib.h*>. glibc declares it in <*malloc.h*>, and also in <*stdlib.h*> if suitable feature test macros are defined (see above).

NOTES

On many systems there are alignment restrictions, for example, on buffers used for direct block device I/O. POSIX specifies the *pathconf(path,_PC_REC_XFER_ALIGN)* call that tells what alignment is needed. Now one can use **posix_memalign**() to satisfy this requirement.

posix_memalign() verifies that *alignment* matches the requirements detailed above. **memalign**() may not check that the *alignment* argument is correct.

POSIX requires that memory obtained from **posix_memalign**() can be freed using *free*(3). Some systems provide no way to reclaim memory allocated with **memalign**() or **valloc**() (because one can pass to *free*(3) only a pointer obtained from *malloc*(3), while, for example, **memalign**() would call *malloc*(3) and then align the obtained value). The glibc implementation allows memory obtained from any of these functions to be reclaimed with *free*(3).

The glibc *malloc*(3) always returns 8-byte aligned memory addresses, so these functions are needed only if you require larger alignment values.

SEE ALSO

brk(2), *getpagesize*(2), *free*(3), *malloc*(3)

NAME

posix_openpt − open a pseudoterminal device

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int flags);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

posix_openpt():
_XOPEN_SOURCE >= 600

DESCRIPTION

The **posix_openpt**() function opens an unused pseudoterminal master device, returning a file descriptor that can be used to refer to that device.

The *flags* argument is a bit mask that ORs together zero or more of the following flags:

O_RDWR
Open the device for both reading and writing. It is usual to specify this flag.

O_NOCTTY
Do not make this device the controlling terminal for the process.

RETURN VALUE

On success, **posix_openpt**() returns a file descriptor (a nonnegative integer) which is the lowest numbered unused file descriptor. On failure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

See *open*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
posix_openpt()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2.1. POSIX.1-2001.

It is part of the UNIX 98 pseudoterminal support (see *pts*(4)).

NOTES

Some older UNIX implementations that support System V (aka UNIX 98) pseudoterminals don't have this function, but it can be easily implemented by opening the pseudoterminal multiplexor device:

```
int
posix_openpt(int flags)
{
    return open("/dev/ptmx", flags);
}
```

Calling **posix_openpt**() creates a pathname for the corresponding pseudoterminal slave device. The pathname of the slave device can be obtained using *ptsname*(3). The slave device pathname exists only as long as the master device is open.

SEE ALSO

open(2), *getpt*(3), *grantpt*(3), *ptsname*(3), *unlockpt*(3), *pts*(4), *pty*(7)

NAME

posix_spawn, posix_spawnp − spawn a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
        const posix_spawn_file_actions_t *restrict file_actions,
        const posix_spawnattr_t *restrict attrp,
        char *const argv[restrict],
        char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
        const posix_spawn_file_actions_t *restrict file_actions,
        const posix_spawnattr_t *restrict attrp,
        char *const argv[restrict],
        char *const envp[restrict]);
```

DESCRIPTION

The **posix_spawn**() and **posix_spawnp**() functions are used to create a new child process that executes a specified file. These functions were specified by POSIX to provide a standardized method of creating new processes on machines that lack the capability to support the *fork*(2) system call. These machines are generally small, embedded systems lacking MMU support.

The **posix_spawn**() and **posix_spawnp**() functions provide the functionality of a combined *fork*(2) and *exec*(3), with some optional housekeeping steps in the child process before the *exec*(3). These functions are not meant to replace the *fork*(2) and *execve*(2) system calls. In fact, they provide only a subset of the functionality that can be achieved by using the system calls.

The only difference between **posix_spawn**() and **posix_spawnp**() is the manner in which they specify the file to be executed by the child process. With **posix_spawn**(), the executable file is specified as a pathname (which can be absolute or relative). With **posix_spawnp**(), the executable file is specified as a simple filename; the system searches for this file in the list of directories specified by **PATH** (in the same way as for *execvp*(3)). For the remainder of this page, the discussion is phrased in terms of **posix_spawn**(), with the understanding that **posix_spawnp**() differs only on the point just described.

The remaining arguments to these two functions are as follows:

pid points to a buffer that is used to return the process ID of the new child process.

file_actions
: points to a *spawn file actions object* that specifies file-related actions to be performed in the child between the *fork*(2) and *exec*(3) steps. This object is initialized and populated before the **posix_spawn**() call using *posix_spawn_file_actions_init*(3) and the **posix_spawn_file_actions_***() functions.

attrp points to an *attributes objects* that specifies various attributes of the created child process. This object is initialized and populated before the **posix_spawn**() call using *posix_spawnattr_init*(3) and the **posix_spawnattr_***() functions.

argv
envp specify the argument list and environment for the program that is executed in the child process, as for *execve*(2).

Below, the functions are described in terms of a three-step process: the **fork**() step, the pre-**exec**() step (executed in the child), and the **exec**() step (executed in the child).

fork() step

Since glibc 2.24, the **posix_spawn**() function commences by calling *clone*(2) with **CLONE_VM** and **CLONE_VFORK** flags. Older implementations use *fork*(2), or possibly *vfork*(2) (see below).

The PID of the new child process is placed in **pid*. The **posix_spawn**() function then returns control to the parent process.

Subsequently, the parent can use one of the system calls described in *wait*(2) to check the status of the child process. If the child fails in any of the housekeeping steps described below, or fails to execute the

desired file, it exits with a status of 127.

Before glibc 2.24, the child process is created using *vfork*(2) instead of *fork*(2) when either of the following is true:

- the *spawn-flags* element of the attributes object pointed to by *attrp* contains the GNU-specific flag **POSIX_SPAWN_USEVFORK**; or
- *file_actions* is NULL and the *spawn-flags* element of the attributes object pointed to by *attrp* does *not* contain **POSIX_SPAWN_SETSIGMASK**, **POSIX_SPAWN_SETSIGDEF**, **POSIX_SPAWN_SETSCHEDPARAM**, **POSIX_SPAWN_SETSCHEDULER**, **POSIX_SPAWN_SETPGROUP**, or **POSIX_SPAWN_RESETIDS**.

In other words, *vfork*(2) is used if the caller requests it, or if there is no cleanup expected in the child before it *exec*(3)s the requested file.

pre-exec() step: housekeeping

In between the **fork()** and the **exec()** steps, a child process may need to perform a set of housekeeping actions. The **posix_spawn**() and **posix_spawnp**() functions support a small, well-defined set of system tasks that the child process can accomplish before it executes the executable file. These operations are controlled by the attributes object pointed to by *attrp* and the file actions object pointed to by *file_actions*. In the child, processing is done in the following sequence:

(1) Process attribute actions: signal mask, signal default handlers, scheduling algorithm and parameters, process group, and effective user and group IDs are changed as specified by the attributes object pointed to by *attrp*.

(2) File actions, as specified in the *file_actions* argument, are performed in the order that they were specified using calls to the **posix_spawn_file_actions_add***() functions.

(3) File descriptors with the **FD_CLOEXEC** flag set are closed.

All process attributes in the child, other than those affected by attributes specified in the object pointed to by *attrp* and the file actions in the object pointed to by *file_actions*, will be affected as though the child was created with *fork*(2) and it executed the program with *execve*(2).

The process attributes actions are defined by the attributes object pointed to by *attrp*. The *spawn-flags* attribute (set using *posix_spawnattr_setflags*(3)) controls the general actions that occur, and other attributes in the object specify values to be used during those actions.

The effects of the flags that may be specified in *spawn-flags* are as follows:

POSIX_SPAWN_SETSIGMASK
Set the signal mask to the signal set specified in the *spawn-sigmask* attribute of the object pointed to by *attrp*. If the **POSIX_SPAWN_SETSIGMASK** flag is not set, then the child inherits the parent's signal mask.

POSIX_SPAWN_SETSIGDEF
Reset the disposition of all signals in the set specified in the *spawn-sigdefault* attribute of the object pointed to by *attrp* to the default. For the treatment of the dispositions of signals not specified in the *spawn-sigdefault* attribute, or the treatment when **POSIX_SPAWN_SETSIGDEF** is not specified, see *execve*(2).

POSIX_SPAWN_SETSCHEDPARAM
If this flag is set, and the **POSIX_SPAWN_SETSCHEDULER** flag is not set, then set the scheduling parameters to the parameters specified in the *spawn-schedparam* attribute of the object pointed to by *attrp*.

POSIX_SPAWN_SETSCHEDULER
Set the scheduling policy algorithm and parameters of the child, as follows:

- The scheduling policy is set to the value specified in the *spawn-schedpolicy* attribute of the object pointed to by *attrp*.
- The scheduling parameters are set to the value specified in the *spawn-schedparam* attribute of the object pointed to by *attrp* (but see BUGS).

If the **POSIX_SPAWN_SETSCHEDPARAM** and **POSIX_SPAWN_SETSCHEDPOLICY** flags are not specified, the child inherits the corresponding scheduling attributes from the

parent.

POSIX_SPAWN_RESETIDS
If this flag is set, reset the effective UID and GID to the real UID and GID of the parent process. If this flag is not set, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID (se *execve*(2)).

POSIX_SPAWN_SETPGROUP
Set the process group to the value specified in the *spawn-pgroup* attribute of the object pointed to by *attrp*. If the *spawn-pgroup* attribute has the value 0, the child's process group ID is made the same as its process ID. If the **POSIX_SPAWN_SETPGROUP** flag is not set, the child inherits the parent's process group ID.

POSIX_SPAWN_USEVFORK
Since glibc 2.24, this flag has no effect. On older implementations, setting this flag forces the **fork()** step to use *vfork*(2) instead of *fork*(2). The **_GNU_SOURCE** feature test macro must be defined to obtain the definition of this constant.

POSIX_SPAWN_SETSID (since glibc 2.26)
If this flag is set, the child process shall create a new session and become the session leader. The child process shall also become the process group leader of the new process group in the session (see *setsid*(2)). The **_GNU_SOURCE** feature test macro must be defined to obtain the definition of this constant.

If *attrp* is NULL, then the default behaviors described above for each flag apply.

The *file_actions* argument specifies a sequence of file operations that are performed in the child process after the general processing described above, and before it performs the *exec*(3). If *file_actions* is NULL, then no special action is taken, and standard *exec*(3) semantics apply—file descriptors open before the exec remain open in the new process, except those for which the **FD_CLOEXEC** flag has been set. File locks remain in place.

If *file_actions* is not NULL, then it contains an ordered set of requests to *open*(2), *close*(2), and *dup2*(2) files. These requests are added to the *file_actions* by *posix_spawn_file_actions_addopen*(3), *posix_spawn_file_actions_addclose*(3), and *posix_spawn_file_actions_adddup2*(3)The requested operations are performed in the order they were added to *file_actions*.

If any of the housekeeping actions fails (due to bogus values being passed or other reasons why signal handling, process scheduling, process group ID functions, and file descriptor operations might fail), the child process exits with exit value 127.

exec() step
Once the child has successfully forked and performed all requested pre-exec steps, the child runs the requested executable.

The child process takes its environment from the *envp* argument, which is interpreted as if it had been passed to *execve*(2). The arguments to the created process come from the *argv* argument, which is processed as for *execve*(2).

RETURN VALUE

Upon successful completion, **posix_spawn**() and **posix_spawnp**() place the PID of the child process in *pid*, and return 0. If there is an error during the **fork()** step, then no child is created, the contents of **pid* are unspecified, and these functions return an error number as described below.

Even when these functions return a success status, the child process may still fail for a plethora of reasons related to its pre-**exec**() initialization. In addition, the *exec*(3) may fail. In all of these cases, the child process will exit with the exit value of 127.

ERRORS

The **posix_spawn**() and **posix_spawnp**() functions fail only in the case where the underlying *fork*(2), *vfork*(2), or *clone*(2) call fails; in these cases, these functions return an error number, which will be one of the errors described for *fork*(2), *vfork*(2), or *clone*(2).

In addition, these functions fail if:

ENOSYS
Function not supported on this system.

STANDARDS
POSIX.1-2008.

HISTORY
glibc 2.2. POSIX.1-2001.

NOTES
The housekeeping activities in the child are controlled by the objects pointed to by *attrp* (for non-file actions) and *file_actions* In POSIX parlance, the *posix_spawnattr_t* and *posix_spawn_file_actions_t* data types are referred to as objects, and their elements are not specified by name. Portable programs should initialize these objects using only the POSIX-specified functions. (In other words, although these objects may be implemented as structures containing fields, portable programs must avoid dependence on such implementation details.)

According to POSIX, it is unspecified whether fork handlers established with *pthread_atfork*(3) are called when **posix_spawn**() is invoked. Since glibc 2.24, the fork handlers are not executed in any case. On older implementations, fork handlers are called only if the child is created using *fork*(2).

There is no "posix_fspawn" function (i.e., a function that is to **posix_spawn**() as *fexecve*(3) is to *execve*(2)). However, this functionality can be obtained by specifying the *path* argument as one of the files in the caller's */proc/self/fd* directory.

BUGS
POSIX.1 says that when **POSIX_SPAWN_SETSCHEDULER** is specified in *spawn-flags*, then the **POSIX_SPAWN_SETSCHEDPARAM** (if present) is ignored. However, before glibc 2.14, calls to **posix_spawn**() failed with an error if **POSIX_SPAWN_SETSCHEDULER** was specified without also specifying **POSIX_SPAWN_SETSCHEDPARAM**.

EXAMPLES
The program below demonstrates the use of various functions in the POSIX spawn API. The program accepts command-line attributes that can be used to create file actions and attributes objects. The remaining command-line arguments are used as the executable name and command-line arguments of the program that is executed in the child.

In the first run, the *date*(1) command is executed in the child, and the **posix_spawn**() call employs no file actions or attributes objects.

```
$ ./a.out date
PID of child: 7634
Tue Feb  1 19:47:50 CEST 2011
Child status: exited, status=0
```

In the next run, the *-c* command-line option is used to create a file actions object that closes standard output in the child. Consequently, *date*(1) fails when trying to perform output and exits with a status of 1.

```
$ ./a.out -c date
PID of child: 7636
date: write error: Bad file descriptor
Child status: exited, status=1
```

In the next run, the *-s* command-line option is used to create an attributes object that specifies that all (blockable) signals in the child should be blocked. Consequently, trying to kill child with the default signal sent by *kill*(1) (i.e., **SIGTERM**) fails, because that signal is blocked. Therefore, to kill the child, **SIGKILL** is necessary (**SIGKILL** can't be blocked).

```
$ ./a.out -s sleep 60 &
[1] 7637
$ PID of child: 7638

$ kill 7638
$ kill -KILL 7638
$ Child status: killed by signal 9
```

```
[1]+  Done                    ./a.out -s sleep 60
```

When we try to execute a nonexistent command in the child, the *exec*(3) fails and the child exits with a status of 127.

```
$ ./a.out xxxxx
PID of child: 10190
Child status: exited, status=127
```

Program source

```
#include <errno.h>
#include <spawn.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wait.h>

#define errExit(msg)    do { perror(msg); \
                             exit(EXIT_FAILURE); } while (0)

#define errExitEN(en, msg) \
                        do { errno = en; perror(msg); \
                             exit(EXIT_FAILURE); } while (0)

char **environ;

int
main(int argc, char *argv[])
{
    pid_t child_pid;
    int s, opt, status;
    sigset_t mask;
    posix_spawnattr_t attr;
    posix_spawnattr_t *attrp;
    posix_spawn_file_actions_t file_actions;
    posix_spawn_file_actions_t *file_actionsp;

    /* Parse command-line options, which can be used to specify an
       attributes object and file actions object for the child. */

    attrp = NULL;
    file_actionsp = NULL;

    while ((opt = getopt(argc, argv, "sc")) != -1) {
        switch (opt) {
        case 'c':       /* -c: close standard output in child */

            /* Create a file actions object and add a "close"
               action to it. */

            s = posix_spawn_file_actions_init(&file_actions);
            if (s != 0)
                errExitEN(s, "posix_spawn_file_actions_init");

            s = posix_spawn_file_actions_addclose(&file_actions,
                                                  STDOUT_FILENO);
            if (s != 0)
```

```
                errExitEN(s, "posix_spawn_file_actions_addclose");

            file_actionsp = &file_actions;
            break;

        case 's':       /* -s: block all signals in child */

            /* Create an attributes object and add a "set signal mask"
               action to it. */

            s = posix_spawnattr_init(&attr);
            if (s != 0)
                errExitEN(s, "posix_spawnattr_init");
            s = posix_spawnattr_setflags(&attr, POSIX_SPAWN_SETSIGMASK);
            if (s != 0)
                errExitEN(s, "posix_spawnattr_setflags");

            sigfillset(&mask);
            s = posix_spawnattr_setsigmask(&attr, &mask);
            if (s != 0)
                errExitEN(s, "posix_spawnattr_setsigmask");

            attrp = &attr;
            break;
        }
    }

    /* Spawn the child. The name of the program to execute and the
       command-line arguments are taken from the command-line arguments
       of this program. The environment of the program execed in the
       child is made the same as the parent's environment. */

    s = posix_spawnp(&child_pid, argv[optind], file_actionsp, attrp,
                     &argv[optind], environ);
    if (s != 0)
        errExitEN(s, "posix_spawn");

    /* Destroy any objects that we created earlier. */

    if (attrp != NULL) {
        s = posix_spawnattr_destroy(attrp);
        if (s != 0)
            errExitEN(s, "posix_spawnattr_destroy");
    }

    if (file_actionsp != NULL) {
        s = posix_spawn_file_actions_destroy(file_actionsp);
        if (s != 0)
            errExitEN(s, "posix_spawn_file_actions_destroy");
    }

    printf("PID of child: %jd\n", (intmax_t) child_pid);

    /* Monitor status of the child until it terminates. */

    do {
        s = waitpid(child_pid, &status, WUNTRACED | WCONTINUED);
        if (s == -1)
            errExit("waitpid");
```

```
            printf("Child status: ");
            if (WIFEXITED(status)) {
                printf("exited, status=%d\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("killed by signal %d\n", WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("stopped by signal %d\n", WSTOPSIG(status));
            } else if (WIFCONTINUED(status)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

close(2), *dup2*(2), *execl*(2), *execlp*(2), *fork*(2), *open*(2), *sched_setparam*(2), *sched_setscheduler*(2), *setpgid*(2), *setuid*(2), *sigaction*(2), *sigprocmask*(2), *posix_spawn_file_actions_addclose*(3), *posix_spawn_file_actions_adddup2*(3), *posix_spawn_file_actions_addopen*(3), *posix_spawn_file_actions_destroy*(3), *posix_spawn_file_actions_init*(3), *posix_spawnattr_destroy*(3), *posix_spawnattr_getflags*(3), *posix_spawnattr_getpgroup*(3), *posix_spawnattr_getschedparam*(3), *posix_spawnattr_getschedpolicy*(3), *posix_spawnattr_getsigdefault*(3), *posix_spawnattr_getsigmask*(3), *posix_spawnattr_init*(3), *posix_spawnattr_setflags*(3), *posix_spawnattr_setpgroup*(3), *posix_spawnattr_setschedparam*(3), *posix_spawnattr_setschedpolicy*(3), *posix_spawnattr_setsigdefault*(3), *posix_spawnattr_setsigmask*(3), *pthread_atfork*(3), <*spawn.h*>, Base Definitions volume of POSIX.1-2001, *http://www.opengroup.org/unix/online.html*

NAME

pow, powf, powl − power functions

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

powf(), **powl**():

```
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

These functions return the value of *x* raised to the power of *y*.

RETURN VALUE

On success, these functions return the value of *x* to the power of *y*.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the mathematically correct sign.

If result underflows, and is not representable, a range error occurs, and 0.0 with the appropriate sign is returned.

If *x* is +0 or −0, and *y* is an odd integer less than 0, a pole error occurs and **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, is returned, with the same sign as *x*.

If *x* is +0 or −0, and *y* is less than 0 and not an odd integer, a pole error occurs and +**HUGE_VAL**, +**HUGE_VALF**, or +**HUGE_VALL**, is returned.

If *x* is +0 (−0), and *y* is an odd integer greater than 0, the result is +0 (−0).

If *x* is 0, and *y* greater than 0 and not an odd integer, the result is +0.

If *x* is −1, and *y* is positive infinity or negative infinity, the result is 1.0.

If *x* is +1, the result is 1.0 (even if *y* is a NaN).

If *y* is 0, the result is 1.0 (even if *x* is a NaN).

If *x* is a finite value less than 0, and *y* is a finite noninteger, a domain error occurs, and a NaN is returned.

If the absolute value of *x* is less than 1, and *y* is negative infinity, the result is positive infinity.

If the absolute value of *x* is greater than 1, and *y* is negative infinity, the result is +0.

If the absolute value of *x* is less than 1, and *y* is positive infinity, the result is +0.

If the absolute value of *x* is greater than 1, and *y* is positive infinity, the result is positive infinity.

If *x* is negative infinity, and *y* is an odd integer less than 0, the result is −0.

If *x* is negative infinity, and *y* less than 0 and not an odd integer, the result is +0.

If *x* is negative infinity, and *y* is an odd integer greater than 0, the result is negative infinity.

If *x* is negative infinity, and *y* greater than 0 and not an odd integer, the result is positive infinity.

If *x* is positive infinity, and *y* less than 0, the result is +0.

If *x* is positive infinity, and *y* greater than 0, the result is positive infinity.

Except as specified above, if *x* or *y* is a NaN, the result is a NaN.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is negative, and *y* is a finite noninteger
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

Pole error: *x* is zero, and *y* is negative
: *errno* is set to **ERANGE** (but see BUGS). A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

Range error: the result overflows
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error: the result underflows
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pow(), **powf**(), **powl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

BUGS

Historical bugs (now fixed)

Before glibc 2.28, on some architectures (e.g., x86-64) **pow**() may be more than 10,000 times slower for some inputs than for other nearby inputs. This affects only **pow**(), and not **powf**() nor **powl**(). This problem was fixed in glibc 2.28.

A number of bugs in the glibc implementation of **pow**() were fixed in glibc 2.16.

In glibc 2.9 and earlier, when a pole error occurs, *errno* is set to **EDOM** instead of the POSIX-mandated **ERANGE**. Since glibc 2.10, glibc does the right thing.

In glibc 2.3.2 and earlier, when an overflow or underflow error occurs, glibc's **pow**() generates a bogus invalid floating-point exception (**FE_INVALID**) in addition to the overflow or underflow exception.

SEE ALSO

cbrt(3), *cpow*(3), *sqrt*(3)

NAME

pow10, pow10f, pow10l − base-10 power functions

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <math.h>

double pow10(double x);
float pow10f(float x);
long double pow10l(long double x);
```

DESCRIPTION

These functions return the value of 10 raised to the power *x*.

Note well: These functions perform exactly the same task as the functions described in *exp10*(3), with the difference that the latter functions are now standardized in TS 18661-4:2015. Those latter functions should be used in preference to the functions described in this page.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pow10(), **pow10f**(), **pow10l**()	Thread safety	MT-Safe

STANDARDS

GNU.

VERSIONS

glibc 2.1. Removed in glibc 2.27.

SEE ALSO

exp10(3), *pow*(3)

NAME

powerof2 − test if a value is a power of 2

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <sys/param.h>

int powerof2(*x***);**

DESCRIPTION

This macro returns true if *x* is a power of 2, and false otherwise.

0 is considered a power of 2. This can make sense considering wrapping of unsigned integers, and has interesting properties.

RETURN VALUE

True or false, if *x* is a power of 2 or not, respectively.

STANDARDS

BSD.

CAVEATS

The arguments may be evaluated more than once.

Because this macro is implemented using bitwise operations, some negative values can invoke undefined behavior. For example, the following invokes undefined behavior: *powerof2(INT_MIN);* . Call it only with unsigned types to be safe.

SEE ALSO

stdc_bit_ceil(3), *stdc_bit_floor*(3)

NAME

__ppc_get_timebase, __ppc_get_timebase_freq − get the current value of the Time Base Register on Power architecture and its frequency.

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/platform/ppc.h>

uint64_t __ppc_get_timebase(void);
uint64_t __ppc_get_timebase_freq(void);

DESCRIPTION

__ppc_get_timebase() reads the current value of the Time Base Register and returns its value, while **__ppc_get_timebase_freq**() returns the frequency in which the Time Base Register is updated.

The Time Base Register is a 64-bit register provided by Power Architecture processors. It stores a monotonically incremented value that is updated at a system-dependent frequency that may be different from the processor frequency.

RETURN VALUE

__ppc_get_timebase() returns a 64-bit unsigned integer that represents the current value of the Time Base Register.

__ppc_get_timebase_freq() returns a 64-bit unsigned integer that represents the frequency at which the Time Base Register is updated.

STANDARDS

GNU.

HISTORY

__ppc_get_timebase()
: glibc 2.16.

__ppc_get_timebase_freq()
: glibc 2.17.

EXAMPLES

The following program will calculate the time, in microseconds, spent between two calls to **__ppc_get_timebase**().

Program source

```
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/platform/ppc.h>

/* Maximum value of the Time Base Register: 2^60 − 1.
   Source: POWER ISA.  */
#define MAX_TB 0xFFFFFFFFFFFFFFF

int
main(void)
{
    uint64_t tb1, tb2, diff;
    uint64_t freq;

    freq = __ppc_get_timebase_freq();
    printf("Time Base frequency = %"PRIu64" Hz\n", freq);

    tb1 = __ppc_get_timebase();

    // Do some stuff...
```

```
        tb2 = __ppc_get_timebase();

        if (tb2 > tb1) {
            diff = tb2 - tb1;
        } else {
            /* Treat Time Base Register overflow.  */
            diff = (MAX_TB - tb2) + tb1;
        }

        printf("Elapsed time  = %1.2f usecs\n",
               (double) diff * 1000000 / freq);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

time(2), *usleep*(3)

Programmer's Manual"

NAME

__ppc_set_ppr_med, __ppc_set_ppr_very_low, __ppc_set_ppr_low, __ppc_set_ppr_med_low, __ppc_set_ppr_med_high − Set the Program Priority Register

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/platform/ppc.h>

void __ppc_set_ppr_med(void);
void __ppc_set_ppr_very_low(void);
void __ppc_set_ppr_low(void);
void __ppc_set_ppr_med_low(void);
void __ppc_set_ppr_med_high(void);

DESCRIPTION

These functions provide access to the *Program Priority Register* (PPR) on the Power architecture.

The PPR is a 64-bit register that controls the program's priority. By adjusting the PPR value the programmer may improve system throughput by causing system resources to be used more efficiently, especially in contention situations. The available unprivileged states are covered by the following functions:

__ppc_set_ppr_med()
: sets the Program Priority Register value to *medium* (default).

__ppc_set_ppr_very_low()
: sets the Program Priority Register value to *very low*.

__ppc_set_ppr_low()
: sets the Program Priority Register value to *low*.

__ppc_set_ppr_med_low()
: sets the Program Priority Register value to *medium low*.

The privileged state *medium high* may also be set during certain time intervals by problem-state (unprivileged) programs, with the following function:

__ppc_set_ppr_med_high()
: sets the Program Priority to *medium high*.

If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the priority is set to medium.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
__ppc_set_ppr_med(), **__ppc_set_ppr_very_low**(), **__ppc_set_ppr_low**(), **__ppc_set_ppr_med_low**(), **__ppc_set_ppr_med_high**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

__ppc_set_ppr_med()
__ppc_set_ppr_low()
__ppc_set_ppr_med_low()
: glibc 2.18.

__ppc_set_ppr_very_low()
__ppc_set_ppr_med_high()
: glibc 2.23.

NOTES

The functions **__ppc_set_ppr_very_low**() and **__ppc_set_ppr_med_high**() will be defined by <*sys/platform/ppc.h*> if **_ARCH_PWR8** is defined. Availability of these functions can be tested using **#ifdef _ARCH_PWR8**.

SEE ALSO

__ppc_yield(3)

Power ISA, Book II - Section 3.1 (Program Priority Registers)

NAME

__ppc_yield, __ppc_mdoio, __ppc_mdoom − Hint the processor to release shared resources

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/platform/ppc.h>

void __ppc_yield(void);
void __ppc_mdoio(void);
void __ppc_mdoom(void);

DESCRIPTION

These functions provide hints about the usage of resources that are shared with other processors on the Power architecture. They can be used, for example, if a program waiting on a lock intends to divert the shared resources to be used by other processors.

__ppc_yield() provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released for use by other processors.

__ppc_mdoio() provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to caching-inhibited storage have been completed.

__ppc_mdoom() provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to cacheable storage for which the data is not in the cache have been completed.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
__ppc_yield(), **__ppc_mdoio**(), **__ppc_mdoom**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.18.

SEE ALSO

__ppc_set_ppr_med(3)

Power ISA, Book II - Section 3.2 ("or" architecture)

NAME

printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf − formatted output conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int printf(const char *restrict format, ...);
int fprintf(FILE *restrict stream,
        const char *restrict format, ...);
int dprintf(int fd,
        const char *restrict format, ...);
int sprintf(char *restrict str,
        const char *restrict format, ...);
int snprintf(char str[restrict .size], size_t size,
        const char *restrict format, ...);

int vprintf(const char *restrict format, va_list ap);
int vfprintf(FILE *restrict stream,
        const char *restrict format, va_list ap);
int vdprintf(int fd,
        const char *restrict format, va_list ap);
int vsprintf(char *restrict str,
        const char *restrict format, va_list ap);
int vsnprintf(char str[restrict .size], size_t size,
        const char *restrict format, va_list ap);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

snprintf(), **vsnprintf**():
  _XOPEN_SOURCE >= 500 || _ISOC99_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE

dprintf(), **vdprintf**():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _GNU_SOURCE

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The functions **printf**() and **vprintf**() write output to *stdout*, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**() write to the character string *str*.

The function **dprintf**() is the same as **fprintf**() except that it outputs to a file descriptor, *fd*, instead of to a *stdio*(3) stream.

The functions **snprintf**() and **vsnprintf**() write at most *size* bytes (including the terminating null byte ('\0')) to *str*.

The functions **vprintf**(), **vfprintf**(), **vdprintf**(), **vsprintf**(), **vsnprintf**() are equivalent to the functions **printf**(), **fprintf**(), **dprintf**(), **sprintf**(), **snprintf**(), respectively, except that they are called with a *va_list* instead of a variable number of arguments. These functions do not call the *va_end* macro. Because they invoke the *va_arg* macro, the value of *ap* is undefined after the call. See *stdarg*(3).

All of these functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of *stdarg*(3)) are converted for output.

C99 and POSIX.1-2001 specify that the results are undefined if a call to **sprintf**(), **snprintf**(), **vsprintf**(), or **vsnprintf**() would cause copying to take place between objects that overlap (e.g., if the target string array and one of the supplied input arguments refer to the same buffer). See CAVEATS.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The overall syntax of a conversion specification is:

%[$][flags][width][.precision][length modifier]conversion

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each '*' (see *Field width* and *Precision* below) and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m$" instead of '%' and "*m$" instead of '*', where the decimal integer *m* denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
printf("%*d", width, num);
```

and

```
printf("%2$*1$d", width, num);
```

are equivalent. The second style allows repeated references to the same argument. The C99 standard does not include the style using '$', which comes from the Single UNIX Specification. If the style using '$' is used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with "%%" formats, which do not consume an argument. There may be no gaps in the numbers of arguments specified using '$'; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

For some numeric conversions a radix character ("decimal point") or thousands' grouping character is used. The actual character used depends on the **LC_NUMERIC** part of the locale. (See *setlocale*(3).) The POSIX locale uses '.' as radix character, and does not have a grouping character. Thus,

```
printf("%'.2f", 1234567.89);
```

results in "1234567.89" in the POSIX locale, in "1234567,89" in the nl_NL locale, and in "1.234.567,89" in the da_DK locale.

Flag characters

The character % is followed by zero or more of the following flags:

The value should be converted to an "alternate form". For **o** conversions, the first character of the output string is made zero (by prefixing a 0 if it was not zero already). For **x** and **X** conversions, a nonzero result has the string "0x" (or "0X" for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be. For **m**, if *errno* contains a valid error code, the output of *strerrorname_np(errno)* is printed; otherwise, the value stored in *errno* is printed as a decimal number. For other conversions, the result is undefined.

0 The value should be zero padded. For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the converted value is padded on the left with zeros rather than blanks. If the **0** and − flags both appear, the **0** flag is ignored. If a precision is given with an integer conversion (**d**, **i**, **o**, **u**, **x**, and **X**), the **0** flag is ignored. For other conversions, the behavior is undefined.

− The converted value is to be left adjusted on the field boundary. (The default is right justification.) The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A − overrides a **0** if both are given.

' ' (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

+ A sign (+ or −) should always be placed before a number produced by a signed conversion. By default, a sign is used only for negative numbers. A **+** overrides a space if both are used.

The five flag characters above are defined in the C99 standard. The Single UNIX Specification specifies one further flag character.

' For decimal conversion (**i**, **d**, **u**, **f**, **F**, **g**, **G**) the output is to be grouped with thousands' grouping characters if the locale information indicates any. (See *setlocale*(3).) Note that many versions of *gcc*(1) cannot parse this option and will issue a warning. (SUSv2 did not include *%'F*, but SUSv3 added it.) Note also that the default locale of a C program is "C" whose locale information indicates no thousands' grouping character. Therefore, without a prior call to *setlocale*(3), no thousands' grouping characters will be printed.

glibc 2.2 adds one further flag character.

I For decimal integer conversion (**i**, **d**, **u**) the output uses the locale's alternative output digits, if any. For example, since glibc 2.2.3 this will give Arabic-Indic digits in the Persian ("fa_IR") locale.

Field width

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write "*" or "*m$" (for some decimal integer *m*) to specify that the field width is given in the next argument, or in the *m*-th argument, respectively, which must be of type *int*. A negative field width is taken as a '−' flag followed by a positive field width. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Precision

An optional precision, in the form of a period ('.') followed by an optional decimal digit string. Instead of a decimal digit string one may write "*" or "*m$" (for some decimal integer *m*) to specify that the precision is given in the next argument, or in the *m*-th argument, respectively, which must be of type *int*. If the precision is given as just '.', the precision is taken to be zero. A negative precision is taken as if the precision were omitted. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the radix character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** and **S** conversions.

Length modifier

Here, "integer conversion" stands for **d**, **i**, **o**, **u**, **x**, or **X** conversion.

hh A following integer conversion corresponds to a *signed char* or *unsigned char* argument, or a following **n** conversion corresponds to a pointer to a *signed char* argument.

h A following integer conversion corresponds to a *short* or *unsigned short* argument, or a following **n** conversion corresponds to a pointer to a *short* argument.

l (ell) A following integer conversion corresponds to a *long* or *unsigned long* argument, or a following **n** conversion corresponds to a pointer to a *long* argument, or a following **c** conversion corresponds to a *wint_t* argument, or a following **s** conversion corresponds to a pointer to *wchar_t* argument. On a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion, this length modifier is ignored (C99; not in SUSv2).

ll (ell-ell). A following integer conversion corresponds to a *long long* or *unsigned long long* argument, or a following **n** conversion corresponds to a pointer to a *long long* argument.

q A synonym for **ll**. This is a nonstandard extension, derived from BSD; avoid its use in new code.

L A following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion corresponds to a *long double* argument. (C99 allows %LF, but SUSv2 does not.)

j A following integer conversion corresponds to an *intmax_t* or *uintmax_t* argument, or a following **n** conversion corresponds to a pointer to an *intmax_t* argument.

z A following integer conversion corresponds to a *size_t* or *ssize_t* argument, or a following **n** conversion corresponds to a pointer to a *size_t* argument.

Z A nonstandard synonym for **z** that predates the appearance of **z**. Do not use in new code.

t A following integer conversion corresponds to a *ptrdiff_t* argument, or a following **n** conversion corresponds to a pointer to a *ptrdiff_t* argument.

SUSv3 specifies all of the above, except for those modifiers explicitly noted as being nonstandard extensions. SUSv2 specified only the length modifiers **h** (in **hd**, **hi**, **ho**, **hx**, **hX**, **hn**) and **l** (in **ld**, **li**, **lo**, **lx**, **lX**, **ln**, **lc**, **ls**) and **L** (in **Le**, **LE**, **Lf**, **Lg**, **LG**).

As a nonstandard extension, the GNU implementations treats **ll** and **L** as synonyms, so that one can, for example, write **llg** (as a synonym for the standards-compliant **Lg**) and **Ld** (as a synonym for the standards compliant **lld**). Such usage is nonportable.

Conversion specifiers

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, **i** The *int* argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

o, **u**, **x**, **X**

The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters **abcdef** are used for **x** conversions; the letters **ABCDEF** are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

e, **E** The *double* argument is rounded and converted in the style [−]d**.**ddd**e**±dd where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter **E** (rather than **e**) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

f, **F** The *double* argument is rounded and converted to decimal notation in the style [−]ddd**.**ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

(SUSv2 does not know about **F** and says that character string representations for infinity and NaN may be made available. SUSv3 adds a specification for **F**. The C99 standard specifies "[−]inf" or "[−]infinity" for infinity, and a string starting with "nan" for NaN, in the case of **f** conversion, and "[−]INF" or "[−]INFINITY" or "NAN" in the case of **F** conversion.)

g, **G** The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

a, **A** (C99; not in SUSv2, but added in SUSv3) For **a** conversion, the *double* argument is converted to hexadecimal notation (using the letters abcdef) in the style [−]**0x**h**.**hhhh**p**±d; for **A** conversion the prefix **0X**, the letters ABCDEF, and the exponent separator **P** is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type *double*. The digit before the decimal point is unspecified for nonnormalized numbers, and nonzero but otherwise unspecified for normalized numbers. The exponent always contains at least one digit; if the value is zero, the exponent is 0.

c If no **l** modifier is present, the *int* argument is converted to an *unsigned char*, and the resulting character is written. If an **l** modifier is present, the *wint_t* (wide character) argument is converted to a multibyte sequence by a call to the *wcrtomb*(3) function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

s If no **l** modifier is present: the *const char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

If an **l** modifier is present: the *const wchar_t* * argument is expected to be a pointer to an array of wide characters. Wide characters from the array are converted to multibyte characters (each by a call to the *wcrtomb*(3) function, with a conversion state starting in the initial state before the first wide character), up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null byte. If a precision is specified, no more bytes than the number specified are written, but no partial multibyte characters are written. Note that the precision determines the number of *bytes* written, not the number of *wide characters* or *screen positions*. The array must contain a terminating null wide character, unless a precision is given and it is so small that the number of bytes written exceeds it before the end of the array is reached.

C (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for **lc**. Don't use.

S (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for **ls**. Don't use.

p The *void* * pointer argument is printed in hexadecimal (as if by **%#x** or **%#lx**).

n The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an *int* *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. (This specifier is not supported by the bionic C library.) The behavior is undefined if the conversion specification includes any flags, a field width, or a precision.

m (glibc extension; supported by uClibc and musl.) Print output of *strerror(errno)* (or *strerrorname_np(errno)* in the alternate form). No argument is required.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

RETURN VALUE

Upon successful return, these functions return the number of bytes printed (excluding the null byte used to end output to strings).

The functions **snprintf**() and **vsnprintf**() do not write more than *size* bytes (including the terminating null byte ('\0')). If the output was truncated due to this limit, then the return value is the number of characters (excluding the terminating null byte) which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated. (See also below under CAVEATS.)

If an output error is encountered, a negative value is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
printf(), **fprintf**(), **sprintf**(), **snprintf**(), **vprintf**(), **vfprintf**(), **vsprintf**(), **vsnprintf**()	Thread safety	MT-Safe locale

STANDARDS

fprintf()
printf()
sprintf()
vprintf()
vfprintf()
vsprintf()

snprintf()
vsnprintf()
C11, POSIX.1-2008.

dprintf()
vdprintf()
GNU, POSIX.1-2008.

HISTORY

fprintf()
printf()
sprintf()
vprintf()
vfprintf()
vsprintf()
C89, POSIX.1-2001.

snprintf()
vsnprintf()
SUSv2, C99, POSIX.1-2001.

Concerning the return value of **snprintf**(), SUSv2 and C99 contradict each other: when **snprintf**() is called with *size*=0 then SUSv2 stipulates an unspecified return value less than 1, while C99 allows *str* to be NULL in this case, and gives the return value (as always) as the number of characters that would have been written in case the output string has been large enough. POSIX.1-2001 and later align their specification of **snprintf**() with C99.

dprintf()
vdprintf()
GNU, POSIX.1-2008.

glibc 2.1 adds length modifiers **hh**, **j**, **t**, and **z** and conversion characters **a** and **A**.

glibc 2.2 adds the conversion character **F** with C99 semantics, and the flag character **I**.

glibc 2.35 gives a meaning to the alternate form (**#**) of the **m** conversion specifier, that is *%#m*.

CAVEATS

Some programs imprudently rely on code such as the following

```
sprintf(buf, "%s some further text", buf);
```

to append text to *buf*. However, the standards explicitly note that the results are undefined if source and destination buffers overlap when calling **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**(). Depending on the version of *gcc*(1) used, and the compiler options employed, calls such as the above will **not** produce the expected results.

The glibc implementation of the functions **snprintf**() and **vsnprintf**() conforms to the C99 standard, that is, behaves as described above, since glibc 2.1. Until glibc 2.0.6, they would return −1 when the output was truncated.

BUGS

Because **sprintf**() and **vsprintf**() assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to predict. Use **snprintf**() and **vsnprintf**() instead (or *asprintf*(3) and *vasprintf*(3)).

Code such as **printf(** *foo***);** often indicates a bug, since *foo* may contain a % character. If *foo* comes from untrusted user input, it may contain **%n**, causing the **printf**() call to write to memory and creating a security hole.

EXAMPLES

To print *Pi* to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to

strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

Many countries use the day-month-year order. Hence, an internationalized version must be able to print the arguments in an order specified by the format:

```
#include <stdio.h>
fprintf(stdout, format,
        weekday, month, day, hour, min);
```

where *format* depends on locale, and may permute the arguments. With the value:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

one might obtain "Sonntag, 3. Juli, 10:02".

To allocate a sufficiently large string and print into it (code correct for both glibc 2.0 and glibc 2.1):

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

char *
make_message(const char *fmt, ...)
{
    int n = 0;
    size_t size = 0;
    char *p = NULL;
    va_list ap;

    /* Determine required size. */

    va_start(ap, fmt);
    n = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (n < 0)
        return NULL;

    size = (size_t) n + 1;      /* One extra byte for '\0' */
    p = malloc(size);
    if (p == NULL)
        return NULL;

    va_start(ap, fmt);
    n = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (n < 0) {
        free(p);
        return NULL;
    }

    return p;
}
```

If truncation occurs in glibc versions prior to glibc 2.0.6, this is treated as an error instead of being handled gracefully.

SEE ALSO

printf (1), *asprintf*(3), *puts*(3), *scanf*(3), *setlocale*(3), *strfromd*(3), *wcrtomb*(3), *wprintf*(3), *locale*(5)

NAME

profil − execution time profile

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int profil(unsigned short *buf, size_t bufsiz,
           size_t offset, unsigned int scale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

profil():
Since glibc 2.21:
_DEFAULT_SOURCE
In glibc 2.19 and 2.20:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
Up to and including glibc 2.19:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

This routine provides a means to find out in what areas your program spends most of its time. The argument *buf* points to *bufsiz* bytes of core. Every virtual 10 milliseconds, the user's program counter (PC) is examined: *offset* is subtracted and the result is multiplied by *scale* and divided by 65536. If the resulting value is less than *bufsiz*, then the corresponding entry in *buf* is incremented. If *buf* is NULL, profiling is disabled.

RETURN VALUE

Zero is always returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
profil()	Thread safety	MT-Unsafe

STANDARDS

None.

HISTORY

Similar to a call in SVr4.

BUGS

profil() cannot be used on a program that also uses **ITIMER_PROF** interval timers (see *setitimer*(2)).

True kernel profiling provides more accurate results.

SEE ALSO

gprof (1), *sprof*(1), *setitimer*(2), *sigaction*(2), *signal*(2)

NAME

program_invocation_name, program_invocation_short_name − obtain name used to invoke calling program

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <errno.h>

extern char *program_invocation_name;
extern char *program_invocation_short_name;
```

DESCRIPTION

program_invocation_name contains the name that was used to invoke the calling program. This is the same as the value of *argv[0]* in *main*(), with the difference that the scope of *program_invocation_name* is global.

program_invocation_short_name contains the basename component of name that was used to invoke the calling program. That is, it is the same value as *program_invocation_name*, with all text up to and including the final slash (/), if any, removed.

These variables are automatically initialized by the glibc run-time startup code.

VERSIONS

The Linux-specific */proc/*pid*/cmdline* file provides access to similar information.

STANDARDS

GNU.

SEE ALSO

proc(5)

NAME

psignal, psiginfo − print signal description

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

void psignal(int *sig***, const char ****s***);**
void psiginfo(const siginfo_t ****pinfo*, const char ****s***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

psignal():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

psiginfo():
_POSIX_C_SOURCE >= 200809L

DESCRIPTION

The **psignal**() function displays a message on *stderr* consisting of the string *s*, a colon, a space, a string describing the signal number *sig*, and a trailing newline. If the string *s* is NULL or empty, the colon and space are omitted. If *sig* is invalid, the message displayed will indicate an unknown signal.

The **psiginfo**() function is like **psignal**(), except that it displays information about the signal described by *pinfo*, which should point to a valid *siginfo_t* structure. As well as the signal description, **psiginfo**() displays information about the origin of the signal, and other information relevant to the signal (e.g., the relevant memory address for hardware-generated signals, the child process ID for **SIGCHLD**, and the user ID and process ID of the sender, for signals set using *kill*(2) or *sigqueue*(3)).

RETURN VALUE

The **psignal**() and **psiginfo**() functions return no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
psignal(), **psiginfo**()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.10. POSIX.1-2008, 4.3BSD.

BUGS

Up to glibc 2.12, **psiginfo**() had the following bugs:

- In some circumstances, a trailing newline is not printed.
- Additional details are not displayed for real-time signals.

SEE ALSO

sigaction(2), *perror*(3), *strsignal*(3), *signal*(7)

NAME

pthread_atfork − register fork handlers

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_atfork(void (**prepare***)(void), void (****parent***)(void),**
void (**child***)(void));**

DESCRIPTION

The **pthread_atfork**() function registers fork handlers that are to be executed when *fork*(2) is called by any thread in a process. The handlers are executed in the context of the thread that calls *fork*(2).

Three kinds of handler can be registered:

- *prepare* specifies a handler that is executed in the parent process before *fork*(2) processing starts.
- *parent* specifies a handler that is executed in the parent process after *fork*(2) processing completes.
- *child* specifies a handler that is executed in the child process after *fork*(2) processing completes.

Any of the three arguments may be NULL if no handler is needed in the corresponding phase of *fork*(2) processing.

RETURN VALUE

On success, **pthread_atfork**() returns zero. On error, it returns an error number. **pthread_atfork**() may be called multiple times by a process to register additional handlers. The handlers for each phase are called in a specified order: the *prepare* handlers are called in reverse order of registration; the *parent* and *child* handlers are called in the order of registration.

ERRORS

ENOMEM
Could not allocate memory to record the fork handler list entry.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

When *fork*(2) is called in a multithreaded process, only the calling thread is duplicated in the child process. The original intention of **pthread_atfork**() was to allow the child process to be returned to a consistent state. For example, at the time of the call to *fork*(2), other threads may have locked mutexes that are visible in the user-space memory duplicated in the child. Such mutexes would never be unlocked, since the threads that placed the locks are not duplicated in the child. The intent of **pthread_atfork**() was to provide a mechanism whereby the application (or a library) could ensure that mutexes and other process and thread state would be restored to a consistent state. In practice, this task is generally too difficult to be practicable.

After a *fork*(2) in a multithreaded process returns in the child, the child should call only async-signal-safe functions (see *signal-safety*(7)) until such time as it calls *execve*(2) to execute a new program.

POSIX.1 specifies that **pthread_atfork**() shall not fail with the error **EINTR**.

SEE ALSO

fork(2), *atexit*(3), *pthreads*(7)

NAME

pthread_attr_init, pthread_attr_destroy − initialize and destroy thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_attr_init(pthread_attr_t *_attr_);
int pthread_attr_destroy(pthread_attr_t *_attr_);

DESCRIPTION

The **pthread_attr_init**() function initializes the thread attributes object pointed to by *attr* with default attribute values. After this call, individual attributes of the object can be set using various related functions (listed under SEE ALSO), and then the object can be used in one or more *pthread_create*(3) calls that create threads.

Calling **pthread_attr_init**() on a thread attributes object that has already been initialized results in undefined behavior.

When a thread attributes object is no longer required, it should be destroyed using the **pthread_attr_destroy**() function. Destroying a thread attributes object has no effect on threads that were created using that object.

Once a thread attributes object has been destroyed, it can be reinitialized using **pthread_attr_init**(). Any other use of a destroyed thread attributes object has undefined results.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

POSIX.1 documents an **ENOMEM** error for **pthread_attr_init**(); on Linux these functions always succeed (but portable and future-proof applications should nevertheless handle a possible error return).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_init(), **pthread_attr_destroy**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The *pthread_attr_t* type should be treated as opaque: any access to the object other than via pthreads functions is nonportable and produces undefined results.

EXAMPLES

The program below optionally makes use of **pthread_attr_init**() and various related functions to initialize a thread attributes object that is used to create a single thread. Once created, the thread uses the *pthread_getattr_np*(3) function (a nonstandard GNU extension) to retrieve the thread's attributes, and then displays those attributes.

If the program is run with no command-line argument, then it passes NULL as the *attr* argument of *pthread_create*(3), so that the thread is created with default attributes. Running the program on Linux/x86-32 with the NPTL threading implementation, we see the following:

```
$ ulimit -s       # No stack limit ==> default stack size is 2 MB
unlimited
$ ./a.out
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
```

```
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

When we supply a stack size as a command-line argument, the program initializes a thread attributes object, sets various attributes in that object, and passes a pointer to the object in the call to *pthread_create*(3). Running the program on Linux/x86-32 with the NPTL threading implementation, we see the following:

```
$ ./a.out 0x3000000
posix_memalign() allocated at 0x40197000
Thread attributes:
        Detach state        = PTHREAD_CREATE_DETACHED
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_EXPLICIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 0 bytes
        Stack address       = 0x40197000
        Stack size          = 0x3000000 bytes
```

Program source

```
#define _GNU_SOURCE     /* To get pthread_getattr_np() declaration */
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
display_pthread_attr(pthread_attr_t *attr, char *prefix)
{
    int s, i;
    size_t v;
    void *stkaddr;
    struct sched_param sp;

    s = pthread_attr_getdetachstate(attr, &i);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getdetachstate");
    printf("%sDetach state        = %s\n", prefix,
           (i == PTHREAD_CREATE_DETACHED) ? "PTHREAD_CREATE_DETACHED" :
           (i == PTHREAD_CREATE_JOINABLE) ? "PTHREAD_CREATE_JOINABLE" :
           "???");

    s = pthread_attr_getscope(attr, &i);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getscope");
    printf("%sScope               = %s\n", prefix,
           (i == PTHREAD_SCOPE_SYSTEM)  ? "PTHREAD_SCOPE_SYSTEM" :
           (i == PTHREAD_SCOPE_PROCESS) ? "PTHREAD_SCOPE_PROCESS" :
           "???");

    s = pthread_attr_getinheritsched(attr, &i);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getinheritsched");
    printf("%sInherit scheduler   = %s\n", prefix,
```

```
                (i == PTHREAD_INHERIT_SCHED)  ? "PTHREAD_INHERIT_SCHED" :
                (i == PTHREAD_EXPLICIT_SCHED) ? "PTHREAD_EXPLICIT_SCHED" :
                "???");

        s = pthread_attr_getschedpolicy(attr, &i);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_attr_getschedpolicy");
        printf("%sScheduling policy   = %s\n", prefix,
                (i == SCHED_OTHER) ? "SCHED_OTHER" :
                (i == SCHED_FIFO)  ? "SCHED_FIFO" :
                (i == SCHED_RR)    ? "SCHED_RR" :
                "???");

        s = pthread_attr_getschedparam(attr, &sp);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_attr_getschedparam");
        printf("%sScheduling priority = %d\n", prefix, sp.sched_priority);

        s = pthread_attr_getguardsize(attr, &v);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_attr_getguardsize");
        printf("%sGuard size          = %zu bytes\n", prefix, v);

        s = pthread_attr_getstack(attr, &stkaddr, &v);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_attr_getstack");
        printf("%sStack address       = %p\n", prefix, stkaddr);
        printf("%sStack size          = %#zx bytes\n", prefix, v);
    }

    static void *
    thread_start(void *arg)
    {
        int s;
        pthread_attr_t gattr;

        /* pthread_getattr_np() is a non-standard GNU extension that
           retrieves the attributes of the thread specified in its
           first argument. */

        s = pthread_getattr_np(pthread_self(), &gattr);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_getattr_np");

        printf("Thread attributes:\n");
        display_pthread_attr(&gattr, "\t");

        exit(EXIT_SUCCESS);         /* Terminate all threads */
    }

    int
    main(int argc, char *argv[])
    {
        pthread_t thr;
        pthread_attr_t attr;
        pthread_attr_t *attrp;      /* NULL or &attr */
        int s;

        attrp = NULL;
```

```
        /* If a command-line argument was supplied, use it to set the
           stack-size attribute and set a few other thread attributes,
           and set attrp pointing to thread attributes object. */

        if (argc > 1) {
            size_t stack_size;
            void *sp;

            attrp = &attr;

            s = pthread_attr_init(&attr);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_init");

            s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_setdetachstate");

            s = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_setinheritsched");

            stack_size = strtoul(argv[1], NULL, 0);

            s = posix_memalign(&sp, sysconf(_SC_PAGESIZE), stack_size);
            if (s != 0)
                errc(EXIT_FAILURE, s, "posix_memalign");

            printf("posix_memalign() allocated at %p\n", sp);

            s = pthread_attr_setstack(&attr, sp, stack_size);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_setstack");
        }

        s = pthread_create(&thr, attrp, &thread_start, NULL);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_create");

        if (attrp != NULL) {
            s = pthread_attr_destroy(attrp);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_destroy");
        }

        pause();    /* Terminates when other thread calls exit() */
    }
```

SEE ALSO

pthread_attr_setaffinity_np(3), *pthread_attr_setdetachstate*(3), *pthread_attr_setguardsize*(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedparam*(3), *pthread_attr_setschedpolicy*(3), *pthread_attr_setscope*(3), *pthread_attr_setsigmask_np*(3), *pthread_attr_setstack*(3), *pthread_attr_setstackaddr*(3), *pthread_attr_setstacksize*(3), *pthread_create*(3), *pthread_getattr_np*(3), *pthread_setattr_default_np*(3), *pthreads*(7)

NAME

pthread_attr_setaffinity_np, pthread_attr_getaffinity_np − set/get CPU affinity attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_attr_setaffinity_np(pthread_attr_t *attr,
                size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_attr_getaffinity_np(const pthread_attr_t *attr,
                size_t cpusetsize, cpu_set_t *cpuset);
```

DESCRIPTION

The **pthread_attr_setaffinity_np**() function sets the CPU affinity mask attribute of the thread attributes object referred to by *attr* to the value specified in *cpuset*. This attribute determines the CPU affinity mask of a thread created using the thread attributes object *attr*.

The **pthread_attr_getaffinity_np**() function returns the CPU affinity mask attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *cpuset*.

The argument *cpusetsize* is the length (in bytes) of the buffer pointed to by *cpuset*. Typically, this argument would be specified as *sizeof(cpu_set_t)*.

For more details on CPU affinity masks, see *sched_setaffinity*(2). For a description of a set of macros that can be used to manipulate and inspect CPU sets, see *CPU_SET*(3).

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

EINVAL

(**pthread_attr_setaffinity_np**()) *cpuset* specified a CPU that was outside the set supported by the kernel. (The kernel configuration option **CONFIG_NR_CPUS** defines the range of the set supported by the kernel data type used to represent CPU sets.)

EINVAL

(**pthread_attr_getaffinity_np**()) A CPU in the affinity mask of the thread attributes object referred to by *attr* lies outside the range specified by *cpusetsize* (i.e., *cpuset*/*cpusetsize* is too small).

ENOMEM

(**pthread_attr_setaffinity_np**()) Could not allocate memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setaffinity_np(), **pthread_attr_getaffinity_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.3.4.

NOTES

In glibc 2.3.3 only, versions of these functions were provided that did not have a *cpusetsize* argument. Instead the CPU set size given to the underlying system calls was always *sizeof(cpu_set_t)*.

SEE ALSO

sched_setaffinity(2), *pthread_attr_init*(3), *pthread_setaffinity_np*(3), *cpuset*(7), *pthreads*(7)

NAME

pthread_attr_setdetachstate, pthread_attr_getdetachstate − set/get detach state attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

DESCRIPTION

The **pthread_attr_setdetachstate**() function sets the detach state attribute of the thread attributes object referred to by *attr* to the value specified in *detachstate*. The detach state attribute determines whether a thread created using the thread attributes object *attr* will be created in a joinable or a detached state.

The following values may be specified in *detachstate*:

PTHREAD_CREATE_DETACHED
: Threads that are created using *attr* will be created in a detached state.

PTHREAD_CREATE_JOINABLE
: Threads that are created using *attr* will be created in a joinable state.

The default setting of the detach state attribute in a newly initialized thread attributes object is **PTHREAD_CREATE_JOINABLE**.

The **pthread_attr_getdetachstate**() returns the detach state attribute of the thread attributes object *attr* in the buffer pointed to by *detachstate*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setdetachstate() can fail with the following error:

EINVAL
: An invalid value was specified in *detachstate*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setdetachstate(), **pthread_attr_getdetachstate**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

See *pthread_create*(3) for more details on detached and joinable threads.

A thread that is created in a joinable state should eventually either be joined using *pthread_join*(3) or detached using *pthread_detach*(3); see *pthread_create*(3).

It is an error to specify the thread ID of a thread that was created in a detached state in a later call to *pthread_detach*(3) or *pthread_join*(3).

EXAMPLES

See *pthread_attr_init*(3).

SEE ALSO

pthread_attr_init(3), *pthread_create*(3), *pthread_detach*(3), *pthread_join*(3), *pthreads*(7)

NAME

pthread_attr_setguardsize, pthread_attr_getguardsize − set/get guard size attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);
```

DESCRIPTION

The **pthread_attr_setguardsize**() function sets the guard size attribute of the thread attributes object referred to by *attr* to the value specified in *guardsize*.

If *guardsize* is greater than 0, then for each new thread created using *attr* the system allocates an additional region of at least *guardsize* bytes at the end of the thread's stack to act as the guard area for the stack (but see BUGS).

If *guardsize* is 0, then new threads created with *attr* will not have a guard area.

The default guard size is the same as the system page size.

If the stack address attribute has been set in *attr* (using *pthread_attr_setstack*(3) or *pthread_attr_setstackaddr*(3)), meaning that the caller is allocating the thread's stack, then the guard size attribute is ignored (i.e., no guard area is created by the system): it is the application's responsibility to handle stack overflow (perhaps by using *mprotect*(2) to manually define a guard area at the end of the stack that it has allocated).

The **pthread_attr_getguardsize**() function returns the guard size attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *guardsize*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

POSIX.1 documents an **EINVAL** error if *attr* or *guardsize* is invalid. On Linux these functions always succeed (but portable and future-proof applications should nevertheless handle a possible error return).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setguardsize(), **pthread_attr_getguardsize**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

A guard area consists of virtual memory pages that are protected to prevent read and write access. If a thread overflows its stack into the guard area, then, on most hard architectures, it receives a **SIGSEGV** signal, thus notifying it of the overflow. Guard areas start on page boundaries, and the guard size is internally rounded up to the system page size when creating a thread. (Nevertheless, **pthread_attr_getguardsize**() returns the guard size that was set by *pthread_attr_setguardsize*().)

Setting a guard size of 0 may be useful to save memory in an application that creates many threads and knows that stack overflow can never occur.

Choosing a guard size larger than the default size may be necessary for detecting stack overflows if a thread allocates large data structures on the stack.

BUGS

As at glibc 2.8, the NPTL threading implementation includes the guard area within the stack size allocation, rather than allocating extra space at the end of the stack, as POSIX.1 requires. (This can result

in an **EINVAL** error from *pthread_create*(3) if the guard size value is too large, leaving no space for the actual stack.)

The obsolete LinuxThreads implementation did the right thing, allocating extra space at the end of the stack for the guard area.

EXAMPLES

See *pthread_getattr_np*(3).

SEE ALSO

mmap(2), *mprotect*(2), *pthread_attr_init*(3), *pthread_attr_setstack*(3), *pthread_attr_setstacksize*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_attr_setinheritsched, pthread_attr_getinheritsched − set/get inherit-scheduler attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
                                 int *restrict inheritsched);
```

DESCRIPTION

The **pthread_attr_setinheritsched**() function sets the inherit-scheduler attribute of the thread attributes object referred to by *attr* to the value specified in *inheritsched*. The inherit-scheduler attribute determines whether a thread created using the thread attributes object *attr* will inherit its scheduling attributes from the calling thread or whether it will take them from *attr*.

The following scheduling attributes are affected by the inherit-scheduler attribute: scheduling policy (**pthread_attr_setschedpolicy**(3)), scheduling priority (**pthread_attr_setschedparam**(3)), and contention scope (**pthread_attr_setscope**(3)).

The following values may be specified in *inheritsched*:

PTHREAD_INHERIT_SCHED
: Threads that are created using *attr* inherit scheduling attributes from the creating thread; the scheduling attributes in *attr* are ignored.

PTHREAD_EXPLICIT_SCHED
: Threads that are created using *attr* take their scheduling attributes from the values specified by the attributes object.

The default setting of the inherit-scheduler attribute in a newly initialized thread attributes object is **PTHREAD_INHERIT_SCHED**.

The **pthread_attr_getinheritsched**() returns the inherit-scheduler attribute of the thread attributes object *attr* in the buffer pointed to by *inheritsched*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setinheritsched() can fail with the following error:

EINVAL
: Invalid value in *inheritsched*.

POSIX.1 also documents an optional **ENOTSUP** error ("attempt was made to set the attribute to an unsupported value") for **pthread_attr_setinheritsched**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setinheritsched(), **pthread_attr_getinheritsched**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0. POSIX.1-2001.

BUGS

As at glibc 2.8, if a thread attributes object is initialized using *pthread_attr_init*(3), then the scheduling policy of the attributes object is set to **SCHED_OTHER** and the scheduling priority is set to 0. However, if the inherit-scheduler attribute is then set to **PTHREAD_EXPLICIT_SCHED**, then a thread created using the attribute object wrongly inherits its scheduling attributes from the creating thread.

This bug does not occur if either the scheduling policy or scheduling priority attribute is explicitly set in the thread attributes object before calling *pthread_create*(3).

EXAMPLES

See *pthread_setschedparam*(3).

SEE ALSO

pthread_attr_init(3), *pthread_attr_setschedparam*(3), *pthread_attr_setschedpolicy*(3), *pthread_attr_setscope*(3), *pthread_create*(3), *pthread_setschedparam*(3), *pthread_setschedprio*(3), *pthreads*(7), *sched*(7)

NAME

pthread_attr_setschedparam, pthread_attr_getschedparam − set/get scheduling parameter attributes in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict param);
int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                               struct sched_param *restrict param);
```

DESCRIPTION

The **pthread_attr_setschedparam**() function sets the scheduling parameter attributes of the thread attributes object referred to by *attr* to the values specified in the buffer pointed to by *param*. These attributes determine the scheduling parameters of a thread created using the thread attributes object *attr*.

The **pthread_attr_getschedparam**() returns the scheduling parameter attributes of the thread attributes object *attr* in the buffer pointed to by *param*.

Scheduling parameters are maintained in the following structure:

```
struct sched_param {
    int sched_priority;     /* Scheduling priority */
};
```

As can be seen, only one scheduling parameter is supported. For details of the permitted ranges for scheduling priorities in each scheduling policy, see *sched*(7).

In order for the parameter setting made by **pthread_attr_setschedparam**() to have effect when calling *pthread_create*(3), the caller must use *pthread_attr_setinheritsched*(3) to set the inherit-scheduler attribute of the attributes object *attr* to **PTHREAD_EXPLICIT_SCHED**.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setschedparam() can fail with the following error:

EINVAL
The priority specified in *param* does not make sense for the current scheduling policy of *attr*.

POSIX.1 also documents an **ENOTSUP** error for **pthread_attr_setschedparam**(). This value is never returned on Linux (but portable and future-proof applications should nevertheless handle this error return value).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setschedparam(), **pthread_attr_getschedparam**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. glibc 2.0.

NOTES

See *pthread_attr_setschedpolicy*(3) for a list of the thread scheduling policies supported on Linux.

EXAMPLES

See *pthread_setschedparam*(3).

SEE ALSO

sched_get_priority_min(2), *pthread_attr_init*(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedpolicy*(3), *pthread_create*(3), *pthread_setschedparam*(3), *pthread_setschedprio*(3), *pthreads*(7), *sched*(7)

NAME

pthread_attr_setschedpolicy, pthread_attr_getschedpolicy − set/get scheduling policy attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                                int *restrict policy);
```

DESCRIPTION

The **pthread_attr_setschedpolicy**() function sets the scheduling policy attribute of the thread attributes object referred to by *attr* to the value specified in *policy*. This attribute determines the scheduling policy of a thread created using the thread attributes object *attr*.

The supported values for *policy* are **SCHED_FIFO**, **SCHED_RR**, and **SCHED_OTHER**, with the semantics described in *sched*(7).

The **pthread_attr_getschedpolicy**() returns the scheduling policy attribute of the thread attributes object *attr* in the buffer pointed to by *policy*.

In order for the policy setting made by **pthread_attr_setschedpolicy**() to have effect when calling *pthread_create*(3), the caller must use *pthread_attr_setinheritsched*(3) to set the inherit-scheduler attribute of the attributes object *attr* to **PTHREAD_EXPLICIT_SCHED**.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setschedpolicy() can fail with the following error:

EINVAL
: Invalid value in *policy*.

POSIX.1 also documents an optional **ENOTSUP** error ("attempt was made to set the attribute to an unsupported value") for **pthread_attr_setschedpolicy**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setschedpolicy(), **pthread_attr_getschedpolicy**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0. POSIX.1-2001.

EXAMPLES

See *pthread_setschedparam*(3).

SEE ALSO

pthread_attr_init(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedparam*(3), *pthread_create*(3), *pthread_setschedparam*(3), *pthread_setschedprio*(3), *pthreads*(7), *sched*(7)

NAME

pthread_attr_setscope, pthread_attr_getscope − set/get contention scope attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                          int *restrict scope);
```

DESCRIPTION

The **pthread_attr_setscope**() function sets the contention scope attribute of the thread attributes object referred to by *attr* to the value specified in *scope*. The contention scope attribute defines the set of threads against which a thread competes for resources such as the CPU. POSIX.1 specifies two possible values for *scope*:

PTHREAD_SCOPE_SYSTEM
: The thread competes for resources with all other threads in all processes on the system that are in the same scheduling allocation domain (a group of one or more processors). **PTHREAD_SCOPE_SYSTEM** threads are scheduled relative to one another according to their scheduling policy and priority.

PTHREAD_SCOPE_PROCESS
: The thread competes for resources with all other threads in the same process that were also created with the **PTHREAD_SCOPE_PROCESS** contention scope. **PTHREAD_SCOPE_PROCESS** threads are scheduled relative to other threads in the process according to their scheduling policy and priority. POSIX.1 leaves it unspecified how these threads contend with other threads in other process on the system or with other threads in the same process that were created with the **PTHREAD_SCOPE_SYSTEM** contention scope.

POSIX.1 requires that an implementation support at least one of these contention scopes. Linux supports **PTHREAD_SCOPE_SYSTEM**, but not **PTHREAD_SCOPE_PROCESS**.

On systems that support multiple contention scopes, then, in order for the parameter setting made by **pthread_attr_setscope**() to have effect when calling *pthread_create*(3), the caller must use *pthread_attr_setinheritsched*(3) to set the inherit-scheduler attribute of the attributes object *attr* to **PTHREAD_EXPLICIT_SCHED**.

The **pthread_attr_getscope**() function returns the contention scope attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *scope*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setscope() can fail with the following errors:

EINVAL
: An invalid value was specified in *scope*.

ENOTSUP
: *scope* specified the value **PTHREAD_SCOPE_PROCESS**, which is not supported on Linux.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setscope(), **pthread_attr_getscope**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The **PTHREAD_SCOPE_SYSTEM** contention scope typically indicates that a user-space thread is bound directly to a single kernel-scheduling entity. This is the case on Linux for the obsolete LinuxThreads implementation and the modern NPTL implementation, which are both 1:1 threading implementations.

POSIX.1 specifies that the default contention scope is implementation-defined.

SEE ALSO

pthread_attr_init(3), *pthread_attr_setaffinity_np*(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedparam*(3), *pthread_attr_setschedpolicy*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_attr_setsigmask_np, pthread_attr_getsigmask_np − set/get signal mask attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_attr_setsigmask_np(pthread_attr_t *attr,
                               const sigset_t *sigmask);
int pthread_attr_getsigmask_np(const pthread_attr_t *attr,
                               sigset_t *sigmask);
```

DESCRIPTION

The **pthread_attr_setsigmask_np**() function sets the signal mask attribute of the thread attributes object referred to by *attr* to the value specified in **sigmask*. If *sigmask* is specified as NULL, then any existing signal mask attribute in *attr* is unset.

The **pthread_attr_getsigmask_np**() function returns the signal mask attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *sigmask*. If the signal mask attribute is currently unset, then this function returns the special value **PTHREAD_ATTR_NO_SIGMASK_NP** as its result.

RETURN VALUE

The **pthread_attr_setsigmask_np**() function returns 0 on success, or a nonzero error number on failure.

the **pthread_attr_getsigmask_np**() function returns either 0 or **PTHREAD_ATTR_NO_SIGMASK_NP**. When 0 is returned, the signal mask attribute is returned via *sigmask*. A return value of **PTHREAD_ATTR_NO_SIGMASK_NP** indicates that the signal mask attribute is not set in *attr*.

On error, these functions return a positive error number.

ERRORS

ENOMEM
(**pthread_attr_setsigmask_np**()) Could not allocate memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setsigmask_np(), **pthread_attr_getsigmask_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.32.

NOTES

The signal mask attribute determines the signal mask that will be assigned to a thread created using the thread attributes object *attr*. If this attribute is not set, then a thread created using *attr* will inherit a copy of the creating thread's signal mask.

For more details on signal masks, see *sigprocmask*(2). For a description of a set of macros that can be used to manipulate and inspect signal sets, see *sigsetops*(3).

In the absence of **pthread_attr_setsigmask_np**() it is possible to create a thread with a desired signal mask as follows:

- The creating thread uses *pthread_sigmask*(3) to save its current signal mask and set its mask to block all signals.
- The new thread is then created using **pthread_create**(); the new thread will inherit the creating thread's signal mask.

- The new thread sets its signal mask to the desired value using *pthread_sigmask*(3).
- The creating thread restores its signal mask to the original value.

Following the above steps, there is no possibility for the new thread to receive a signal before it has adjusted its signal mask to the desired value.

SEE ALSO

sigprocmask(2), *pthread_attr_init*(3), *pthread_sigmask*(3), *pthreads*(7), *signal*(7)

NAME

pthread_attr_setstack, pthread_attr_getstack − set/get stack attributes in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr,
                          void stackaddr[.stacksize],
                          size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_attr_getstack(), **pthread_attr_setstack**():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **pthread_attr_setstack**() function sets the stack address and stack size attributes of the thread attributes object referred to by *attr* to the values specified in *stackaddr* and *stacksize*, respectively. These attributes specify the location and size of the stack that should be used by a thread that is created using the thread attributes object *attr*.

stackaddr should point to the lowest addressable byte of a buffer of *stacksize* bytes that was allocated by the caller. The pages of the allocated buffer should be both readable and writable.

The **pthread_attr_getstack**() function returns the stack address and stack size attributes of the thread attributes object referred to by *attr* in the buffers pointed to by *stackaddr* and *stacksize*, respectively.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setstack() can fail with the following error:

EINVAL
: *stacksize* is less than **PTHREAD_STACK_MIN** (16384) bytes. On some systems, this error may also occur if *stackaddr* or *stackaddr* + *stacksize* is not suitably aligned.

POSIX.1 also documents an **EACCES** error if the stack area described by *stackaddr* and *stacksize* is not both readable and writable by the caller.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setstack(), **pthread_attr_getstack**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

NOTES

These functions are provided for applications that must ensure that a thread's stack is placed in a particular location. For most applications, this is not necessary, and the use of these functions should be avoided. (Use *pthread_attr_setstacksize*(3) if an application simply requires a stack size other than the default.)

When an application employs **pthread_attr_setstack**(), it takes over the responsibility of allocating the stack. Any guard size value that was set using *pthread_attr_setguardsize*(3) is ignored. If deemed necessary, it is the application's responsibility to allocate a guard area (one or more pages protected against reading and writing) to handle the possibility of stack overflow.

The address specified in *stackaddr* should be suitably aligned: for full portability, align it on a page

boundary (*sysconf(_SC_PAGESIZE)*). *posix_memalign*(3) may be useful for allocation. Probably, *stacksize* should also be a multiple of the system page size.

If *attr* is used to create multiple threads, then the caller must change the stack address attribute between calls to *pthread_create*(3); otherwise, the threads will attempt to use the same memory area for their stacks, and chaos will ensue.

EXAMPLES

See *pthread_attr_init*(3).

SEE ALSO

mmap(2), *mprotect*(2), *posix_memalign*(3), *pthread_attr_init*(3), *pthread_attr_setguardsize*(3), *pthread_attr_setstackaddr*(3), *pthread_attr_setstacksize*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_attr_setstackaddr, pthread_attr_getstackaddr − set/get stack address attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

[[deprecated]]
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
[[deprecated]]
int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                              void **restrict stackaddr);
```

DESCRIPTION

These functions are obsolete: **do not use them.** Use *pthread_attr_setstack*(3) and *pthread_attr_getstack*(3) instead.

The **pthread_attr_setstackaddr**() function sets the stack address attribute of the thread attributes object referred to by *attr* to the value specified in *stackaddr*. This attribute specifies the location of the stack that should be used by a thread that is created using the thread attributes object *attr*.

stackaddr should point to a buffer of at least **PTHREAD_STACK_MIN** bytes that was allocated by the caller. The pages of the allocated buffer should be both readable and writable.

The **pthread_attr_getstackaddr**() function returns the stack address attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *stackaddr*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

No errors are defined (but applications should nevertheless handle a possible error return).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setstackaddr(), **pthread_attr_getstackaddr**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

glibc 2.1. Marked obsolete in POSIX.1-2001. Removed in POSIX.1-2008.

NOTES

Do not use these functions! They cannot be portably used, since they provide no way of specifying the direction of growth or the range of the stack. For example, on architectures with a stack that grows downward, *stackaddr* specifies the next address past the *highest* address of the allocated stack area. However, on architectures with a stack that grows upward, *stackaddr* specifies the *lowest* address in the allocated stack area. By contrast, the *stackaddr* used by *pthread_attr_setstack*(3) and *pthread_attr_getstack*(3), is always a pointer to the lowest address in the allocated stack area (and the *stacksize* argument specifies the range of the stack).

SEE ALSO

pthread_attr_init(3), *pthread_attr_setstack*(3), *pthread_attr_setstacksize*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_attr_setstacksize, pthread_attr_getstacksize − set/get stack size attribute in thread attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);
```

DESCRIPTION

The **pthread_attr_setstacksize**() function sets the stack size attribute of the thread attributes object referred to by *attr* to the value specified in *stacksize*.

The stack size attribute determines the minimum size (in bytes) that will be allocated for threads created using the thread attributes object *attr*.

The **pthread_attr_getstacksize**() function returns the stack size attribute of the thread attributes object referred to by *attr* in the buffer pointed to by *stacksize*.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

pthread_attr_setstacksize() can fail with the following error:

EINVAL
The stack size is less than **PTHREAD_STACK_MIN** (16384) bytes.

On some systems, **pthread_attr_setstacksize**() can fail with the error **EINVAL** if *stacksize* is not a multiple of the system page size.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_attr_setstacksize(), **pthread_attr_getstacksize**()	Thread safety	MT-Safe

VERSIONS

These functions are provided since glibc 2.1.

STANDARDS

POSIX.1-2001, POSIX.1-2008.

NOTES

For details on the default stack size of new threads, see *pthread_create*(3).

A thread's stack size is fixed at the time of thread creation. Only the main thread can dynamically grow its stack.

The *pthread_attr_setstack*(3) function allows an application to set both the size and location of a caller-allocated stack that is to be used by a thread.

BUGS

As at glibc 2.8, if the specified *stacksize* is not a multiple of **STACK_ALIGN** (16 bytes on most architectures), it may be rounded *downward*, in violation of POSIX.1, which says that the allocated stack will be at least *stacksize* bytes.

EXAMPLES

See *pthread_create*(3).

SEE ALSO

getrlimit(2), *pthread_attr_init*(3), *pthread_attr_setguardsize*(3), *pthread_attr_setstack*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_cancel − send a cancelation request to a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_cancel(pthread_t *thread***);**

DESCRIPTION

The **pthread_cancel**() function sends a cancelation request to the thread *thread*. Whether and when the target thread reacts to the cancelation request depends on two attributes that are under the control of that thread: its cancelability *state* and *type*.

A thread's cancelability state, determined by *pthread_setcancelstate*(3), can be *enabled* (the default for new threads) or *disabled*. If a thread has disabled cancelation, then a cancelation request remains queued until the thread enables cancelation. If a thread has enabled cancelation, then its cancelability type determines when cancelation occurs.

A thread's cancelation type, determined by *pthread_setcanceltype*(3), may be either *asynchronous* or *deferred* (the default for new threads). Asynchronous cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this). Deferred cancelability means that cancelation will be delayed until the thread next calls a function that is a *cancelation point*. A list of functions that are or may be cancelation points is provided in *pthreads*(7).

When a cancelation requested is acted on, the following steps occur for *thread* (in this order):

(1) Cancelation clean-up handlers are popped (in the reverse of the order in which they were pushed) and called. (See *pthread_cleanup_push*(3).)

(2) Thread-specific data destructors are called, in an unspecified order. (See *pthread_key_create*(3).)

(3) The thread is terminated. (See *pthread_exit*(3).)

The above steps happen asynchronously with respect to the **pthread_cancel**() call; the return status of **pthread_cancel**() merely informs the caller whether the cancelation request was successfully queued.

After a canceled thread has terminated, a join with that thread using *pthread_join*(3) obtains **PTHREAD_CANCELED** as the thread's exit status. (Joining with a thread is the only way to know that cancelation has completed.)

RETURN VALUE

On success, **pthread_cancel**() returns 0; on error, it returns a nonzero error number.

ERRORS

ESRCH

No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_cancel()	Thread safety	MT-Safe

VERSIONS

On Linux, cancelation is implemented using signals. Under the NPTL threading implementation, the first real-time signal (i.e., signal 32) is used for this purpose. On LinuxThreads, the second real-time signal is used, if real-time signals are available, otherwise **SIGUSR2** is used.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0 POSIX.1-2001.

EXAMPLES

The program below creates a thread and then cancels it. The main thread joins with the canceled thread to check that its exit status was **PTHREAD_CANCELED**. The following shell session shows what happens when we run the program:

```
$ ./a.out
thread_func(): started; cancelation disabled
main(): sending cancelation request
thread_func(): about to enable cancelation
main(): thread was canceled
```

Program source

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void *
thread_func(void *ignored_argument)
{
    int s;

    /* Disable cancelation for a while, so that we don't
       immediately react to a cancelation request. */

    s = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");

    printf("%s(): started; cancelation disabled\n", __func__);
    sleep(5);
    printf("%s(): about to enable cancelation\n", __func__);

    s = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");

    /* sleep() is a cancelation point. */

    sleep(1000);        /* Should get canceled while we sleep */

    /* Should never get here. */

    printf("%s(): not canceled!\n", __func__);
    return NULL;
}

int
main(void)
{
    pthread_t thr;
    void *res;
    int s;

    /* Start a thread and then send it a cancelation request. */

    s = pthread_create(&thr, NULL, &thread_func, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");
```

```
        sleep(2);           /* Give thread a chance to get started */

        printf("%s(): sending cancelation request\n", __func__);
        s = pthread_cancel(thr);
        if (s != 0)
            handle_error_en(s, "pthread_cancel");

        /* Join with thread to see what its exit status was. */

        s = pthread_join(thr, &res);
        if (s != 0)
            handle_error_en(s, "pthread_join");

        if (res == PTHREAD_CANCELED)
            printf("%s(): thread was canceled\n", __func__);
        else
            printf("%s(): thread wasn't canceled (shouldn't happen!)\n",
                    __func__);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

pthread_cleanup_push(3), *pthread_create*(3), *pthread_exit*(3), *pthread_join*(3), *pthread_key_create*(3), *pthread_setcancelstate*(3), *pthread_setcanceltype*(3), *pthread_testcancel*(3), *pthreads*(7)

NAME

pthread_cleanup_push, pthread_cleanup_pop − push and pop thread cancelation clean-up handlers

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

DESCRIPTION

These functions manipulate the calling thread's stack of thread-cancelation clean-up handlers. A clean-up handler is a function that is automatically executed when a thread is canceled (or in various other circumstances described below); it might, for example, unlock a mutex so that it becomes available to other threads in the process.

The **pthread_cleanup_push**() function pushes *routine* onto the top of the stack of clean-up handlers. When *routine* is later invoked, it will be given *arg* as its argument.

The **pthread_cleanup_pop**() function removes the routine at the top of the stack of clean-up handlers, and optionally executes it if *execute* is nonzero.

A cancelation clean-up handler is popped from the stack and executed in the following circumstances:

- When a thread is canceled, all of the stacked clean-up handlers are popped and executed in the reverse of the order in which they were pushed onto the stack.
- When a thread terminates by calling *pthread_exit*(3), all clean-up handlers are executed as described in the preceding point. (Clean-up handlers are *not* called if the thread terminates by performing a *return* from the thread start function.)
- When a thread calls **pthread_cleanup_pop**() with a nonzero *execute* argument, the top-most clean-up handler is popped and executed.

POSIX.1 permits **pthread_cleanup_push**() and **pthread_cleanup_pop**() to be implemented as macros that expand to text containing '**{**' and '**}**', respectively. For this reason, the caller must ensure that calls to these functions are paired within the same function, and at the same lexical nesting level. (In other words, a clean-up handler is established only during the execution of a specified section of code.)

Calling *longjmp*(3) (**siglongjmp**(3)) produces undefined results if any call has been made to **pthread_cleanup_push**() or **pthread_cleanup_pop**() without the matching call of the pair since the jump buffer was filled by *setjmp*(3) (**sigsetjmp**(3)). Likewise, calling *longjmp*(3) (**siglongjmp**(3)) from inside a clean-up handler produces undefined results unless the jump buffer was also filled by *setjmp*(3) (**sigsetjmp**(3)) inside the handler.

RETURN VALUE

These functions do not return a value.

ERRORS

There are no errors.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_cleanup_push(), **pthread_cleanup_pop**()	Thread safety	MT-Safe

VERSIONS

On glibc, the **pthread_cleanup_push**() and **pthread_cleanup_pop**() functions *are* implemented as macros that expand to text containing '**{**' and '**}**', respectively. This means that variables declared within the scope of paired calls to these functions will be visible within only that scope.

POSIX.1 says that the effect of using *return*, *break*, *continue*, or *goto* to prematurely leave a block bracketed **pthread_cleanup_push**() and **pthread_cleanup_pop**() is undefined. Portable applications should avoid doing this.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. glibc 2.0.

EXAMPLES

The program below provides a simple example of the use of the functions described in this page. The program creates a thread that executes a loop bracketed by **pthread_cleanup_push**() and **pthread_cleanup_pop**(). This loop increments a global variable, *cnt*, once each second. Depending on what command-line arguments are supplied, the main thread sends the other thread a cancelation request, or sets a global variable that causes the other thread to exit its loop and terminate normally (by doing a *return*).

In the following shell session, the main thread sends a cancelation request to the other thread:

```
$ ./a.out
New thread started
cnt = 0
cnt = 1
Canceling thread
Called clean-up handler
Thread was canceled; cnt = 0
```

From the above, we see that the thread was canceled, and that the cancelation clean-up handler was called and it reset the value of the global variable *cnt* to 0.

In the next run, the main program sets a global variable that causes other thread to terminate normally:

```
$ ./a.out x
New thread started
cnt = 0
cnt = 1
Thread terminated normally; cnt = 2
```

From the above, we see that the clean-up handler was not executed (because *cleanup_pop_arg* was 0), and therefore the value of *cnt* was not reset.

In the next run, the main program sets a global variable that causes the other thread to terminate normally, and supplies a nonzero value for *cleanup_pop_arg*:

```
$ ./a.out x 1
New thread started
cnt = 0
cnt = 1
Called clean-up handler
Thread terminated normally; cnt = 0
```

In the above, we see that although the thread was not canceled, the clean-up handler was executed, because the argument given to **pthread_cleanup_pop**() was nonzero.

Program source

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static int done = 0;
static int cleanup_pop_arg = 0;
static int cnt = 0;
```

```
static void
cleanup_handler(void *arg)
{
    printf("Called clean-up handler\n");
    cnt = 0;
}

static void *
thread_start(void *arg)
{
    time_t curr;

    printf("New thread started\n");

    pthread_cleanup_push(cleanup_handler, NULL);

    curr = time(NULL);

    while (!done) {
        pthread_testcancel();           /* A cancelation point */
        if (curr < time(NULL)) {
            curr = time(NULL);
            printf("cnt = %d\n", cnt);  /* A cancelation point */
            cnt++;
        }
    }

    pthread_cleanup_pop(cleanup_pop_arg);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;

    s = pthread_create(&thr, NULL, thread_start, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    sleep(2);           /* Allow new thread to run a while */

    if (argc > 1) {
        if (argc > 2)
            cleanup_pop_arg = atoi(argv[2]);
        done = 1;

    } else {
        printf("Canceling thread\n");
        s = pthread_cancel(thr);
        if (s != 0)
            handle_error_en(s, "pthread_cancel");
    }

    s = pthread_join(thr, &res);
    if (s != 0)
```

```
            handle_error_en(s, "pthread_join");

        if (res == PTHREAD_CANCELED)
            printf("Thread was canceled; cnt = %d\n", cnt);
        else
            printf("Thread terminated normally; cnt = %d\n", cnt);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

pthread_cancel(3), *pthread_cleanup_push_defer_np*(3), *pthread_setcancelstate*(3), *pthread_testcancel*(3), *pthreads*(7)

NAME

pthread_cleanup_push_defer_np, pthread_cleanup_pop_restore_np − push and pop thread cancelation clean-up handlers while saving cancelability type

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

void pthread_cleanup_push_defer_np(void (**routine***)(void *), void ****arg***);**
void pthread_cleanup_pop_restore_np(int *execute***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_cleanup_push_defer_np(), **pthread_cleanup_pop_defer_np**():
_GNU_SOURCE

DESCRIPTION

These functions are the same as *pthread_cleanup_push*(3) and *pthread_cleanup_pop*(3), except for the differences noted on this page.

Like *pthread_cleanup_push*(3), **pthread_cleanup_push_defer_np**() pushes *routine* onto the thread's stack of cancelation clean-up handlers. In addition, it also saves the thread's current cancelability type, and sets the cancelability type to "deferred" (see *pthread_setcanceltype*(3)); this ensures that cancelation clean-up will occur even if the thread's cancelability type was "asynchronous" before the call.

Like *pthread_cleanup_pop*(3), **pthread_cleanup_pop_restore_np**() pops the top-most clean-up handler from the thread's stack of cancelation clean-up handlers. In addition, it restores the thread's cancelability type to its value at the time of the matching **pthread_cleanup_push_defer_np**().

The caller must ensure that calls to these functions are paired within the same function, and at the same lexical nesting level. Other restrictions apply, as described in *pthread_cleanup_push*(3).

This sequence of calls:

```
pthread_cleanup_push_defer_np(routine, arg);
pthread_cleanup_pop_restore_np(execute);
```

is equivalent to (but shorter and more efficient than):

```
int oldtype;

pthread_cleanup_push(routine, arg);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
...
pthread_setcanceltype(oldtype, NULL);
pthread_cleanup_pop(execute);
```

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.0

SEE ALSO

pthread_cancel(3), *pthread_cleanup_push*(3), *pthread_setcancelstate*(3), *pthread_testcancel*(3), *pthreads*(7)

NAME

pthread_cond_init, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_destroy − operations on conditions

SYNOPSIS

#include <pthread.h>

pthread_cond_t *cond* **= PTHREAD_COND_INITIALIZER;**

int pthread_cond_init(pthread_cond_t ****cond*, pthread_condattr_t ****cond_attr***); int pthread_cond_signal(pthread_cond_t ****cond***); int pthread_cond_broadcast(pthread_cond_t ****cond***); int pthread_cond_wait(pthread_cond_t ****cond***, pthread_mutex_t ****mutex***); int pthread_cond_timedwait(pthread_cond_t ****cond***, pthread_mutex_t ****mutex***, const struct timespec ****abstime***); int pthread_cond_destroy(pthread_cond_t ****cond***);**

DESCRIPTION

A condition (short for ‘‘condition variable’’) is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

pthread_cond_init initializes the condition variable *cond*, using the condition attributes specified in *cond_attr*, or default attributes if *cond_attr* is **NULL**. The LinuxThreads implementation supports no attributes for conditions, hence the *cond_attr* parameter is actually ignored.

Variables of type **pthread_cond_t** can also be initialized statically, using the constant **PTHREAD_COND_INITIALIZER**.

pthread_cond_signal restarts one of the threads that are waiting on the condition variable *cond*. If no threads are waiting on *cond*, nothing happens. If several threads are waiting on *cond*, exactly one is restarted, but it is not specified which.

pthread_cond_broadcast restarts all the threads that are waiting on the condition variable *cond*. Nothing happens if no threads are waiting on *cond*.

pthread_cond_wait atomically unlocks the *mutex* (as per **pthread_unlock_mutex**) and waits for the condition variable *cond* to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The *mutex* must be locked by the calling thread on entrance to **pthread_cond_wait**. Before returning to the calling thread, **pthread_cond_wait** re-acquires *mutex* (as per **pthread_lock_mutex**).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

pthread_cond_timedwait atomically unlocks *mutex* and waits on *cond*, as **pthread_cond_wait** does, but it also bounds the duration of the wait. If *cond* has not been signaled within the amount of time specified by *abstime*, the mutex *mutex* is re-acquired and **pthread_cond_timedwait** returns the error **ETIMEDOUT**. The *abstime* parameter specifies an absolute time, with the same origin as **time**(2) and **gettimeofday**(2): an *abstime* of 0 corresponds to 00:00:00 GMT, January 1, 1970.

pthread_cond_destroy destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to **pthread_cond_destroy**. In the LinuxThreads implementation, no resources are associated with condition variables, thus **pthread_cond_destroy** actually does nothing except checking that the condition has no waiting threads.

CANCELLATION

pthread_cond_wait and **pthread_cond_timedwait** are cancelation points. If a thread is cancelled while suspended in one of these functions, the thread immediately resumes execution, then locks again the *mutex* argument to **pthread_cond_wait** and **pthread_cond_timedwait**, and finally executes the cancelation. Consequently, cleanup handlers are assured that *mutex* is locked when they are called.

ASYNC-SIGNAL SAFETY

The condition functions are not async-signal safe, and should not be called from a signal handler. In particular, calling **pthread_cond_signal** or **pthread_cond_broadcast** from a signal handler may deadlock the calling thread.

RETURN VALUE

All condition variable functions return 0 on success and a non-zero error code on error.

ERRORS

pthread_cond_init, **pthread_cond_signal**, **pthread_cond_broadcast**, and **pthread_cond_wait** never return an error code.

The **pthread_cond_timedwait** function returns the following error codes on error:

ETIMEDOUT
The condition variable was not signaled until the timeout specified by *abstime*.

EINTR
pthread_cond_timedwait was interrupted by a signal.

The **pthread_cond_destroy** function returns the following error code on error:

EBUSY
Some threads are currently waiting on *cond*.

SEE ALSO

pthread_condattr_init(3), **pthread_mutex_lock**(3), **pthread_mutex_unlock**(3), **gettimeofday**(2), **nanosleep**(2).

EXAMPLE

Consider two shared variables *x* and *y*, protected by the mutex *mut*, and a condition variable *cond* that is to be signaled whenever *x* becomes greater than *y*.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Waiting until *x* is greater than *y* is performed as follows:

```
pthread_mutex_lock(&mut);
while (x <= y) {
    pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

Modifications on *x* and *y* that may cause *x* to become greater than *y* should signal the condition if needed:

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

If it can be proved that at most one waiting thread needs to be waken up (for instance, if there are only two threads communicating through *x* and *y*), **pthread_cond_signal** can be used as a slightly more efficient alternative to **pthread_cond_broadcast**. In doubt, use **pthread_cond_broadcast**.

To wait for *x* to become greater than *y* with a timeout of 5 seconds, do:

```
struct timeval now;
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
```

```
while (x <= y && retcode != ETIMEDOUT) {
        retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT) {
        /* timeout occurred */
} else {
        /* operate on x and y */
}
pthread_mutex_unlock(&mut);
```

NAME

pthread_condattr_init, pthread_condattr_destroy − condition creation attributes

SYNOPSIS

#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *_attr_); int pthread_condattr_destroy(pthread_condattr_t *_attr_);

DESCRIPTION

Condition attributes can be specified at condition creation time, by passing a condition attribute object as second argument to **pthread_cond_init**(3). Passing **NULL** is equivalent to passing a condition attribute object with all attributes set to their default values.

The LinuxThreads implementation supports no attributes for conditions. The functions on condition attributes are included only for compliance with the POSIX standard.

pthread_condattr_init initializes the condition attribute object *attr* and fills it with default values for the attributes. **pthread_condattr_destroy** destroys a condition attribute object, which must not be reused until it is reinitialized. Both functions do nothing in the LinuxThreads implementation.

RETURN VALUE

pthread_condattr_init and **pthread_condattr_destroy** always return 0.

SEE ALSO

pthread_cond_init(3).

NAME

pthread_create − create a new thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

DESCRIPTION

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The new thread terminates in one of the following ways:

- It calls *pthread_exit*(3), specifying an exit status value that is available to another thread in the same process that calls *pthread_join*(3).
- It returns from *start_routine*(). This is equivalent to calling *pthread_exit*(3) with the value supplied in the *return* statement.
- It is canceled (see *pthread_cancel*(3)).
- Any of the threads in the process calls *exit*(3), or the main thread performs a return from *main*(). This causes the termination of all threads in the process.

The *attr* argument points to a *pthread_attr_t* structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using *pthread_attr_init*(3) and related functions. If *attr* is NULL, then the thread is created with default attributes.

Before returning, a successful call to **pthread_create**() stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

The new thread inherits a copy of the creating thread's signal mask (**pthread_sigmask**(3)). The set of pending signals for the new thread is empty (**sigpending**(2)). The new thread does not inherit the creating thread's alternate signal stack (**sigaltstack**(2)).

The new thread inherits the calling thread's floating-point environment (**fenv**(3)).

The initial value of the new thread's CPU-time clock is 0 (see *pthread_getcpuclockid*(3)).

Linux-specific details

The new thread inherits copies of the calling thread's capability sets (see *capabilities*(7)) and CPU affinity mask (see *sched_setaffinity*(2)).

RETURN VALUE

On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

ERRORS

EAGAIN
: Insufficient resources to create another thread.

EAGAIN
: A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error: the **RLIMIT_NPROC** soft resource limit (set via *setrlimit*(2)), which limits the number of processes and threads for a real user ID, was reached; the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads−max*, was reached (see *proc*(5)); or the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see *proc*(5)).

EINVAL
: Invalid settings in *attr*.

EPERM
No permission to set the scheduling policy and parameters specified in *attr*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_create()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

See *pthread_self*(3) for further information on the thread ID returned in **thread* by **pthread_create**(). Unless real-time scheduling policies are being employed, after a call to **pthread_create**(), it is indeterminate which thread—the caller or the new thread—will next execute.

A thread may either be *joinable* or *detached*. If a thread is joinable, then another thread can call *pthread_join*(3) to wait for the thread to terminate and fetch its exit status. Only when a terminated joinable thread has been joined are the last of its resources released back to the system. When a detached thread terminates, its resources are automatically released back to the system: it is not possible to join with the thread in order to obtain its exit status. Making a thread detached is useful for some types of daemon threads whose exit status the application does not need to care about. By default, a new thread is created in a joinable state, unless *attr* was set to create the thread in a detached state (using *pthread_attr_setdetachstate*(3)).

Under the NPTL threading implementation, if the **RLIMIT_STACK** soft resource limit *at the time the program started* has any value other than "unlimited", then it determines the default stack size of new threads. Using *pthread_attr_setstacksize*(3), the stack size attribute can be explicitly set in the *attr* argument used to create a thread, in order to obtain a stack size other than the default. If the **RLIMIT_STACK** resource limit is set to "unlimited", a per-architecture value is used for the stack size: 2 MB on most architectures; 4 MB on POWER and Sparc-64.

BUGS

In the obsolete LinuxThreads implementation, each of the threads in a process has a different process ID. This is in violation of the POSIX threads specification, and is the source of many other nonconformances to the standard; see *pthreads*(7).

EXAMPLES

The program below demonstrates the use of **pthread_create**(), as well as a number of other functions in the pthreads API.

In the following run, on a system providing the NPTL threading implementation, the stack size defaults to the value given by the "stack size" resource limit:

```
$ ulimit -s
8192            # The stack size limit is 8 MB (0x800000 bytes)
$ ./a.out hola salut servus
Thread 1: top of stack near 0xb7dd03b8; argv_string=hola
Thread 2: top of stack near 0xb75cf3b8; argv_string=salut
Thread 3: top of stack near 0xb6dce3b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

In the next run, the program explicitly sets a stack size of 1 MB (using *pthread_attr_setstacksize*(3)) for the created threads:

```
$ ./a.out -s 0x100000 hola salut servus
Thread 1: top of stack near 0xb7d723b8; argv_string=hola
Thread 2: top of stack near 0xb7c713b8; argv_string=salut
Thread 3: top of stack near 0xb7b703b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
```

```
    Joined with thread 3; returned value was SERVUS
```

Program source

```
#include <ctype.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

#define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while (0)

struct thread_info {    /* Used as argument to thread_start() */
    pthread_t thread_id;        /* ID returned by pthread_create() */
    int       thread_num;       /* Application-defined thread # */
    char     *argv_string;      /* From command-line argument */
};

/* Thread start function: display address near top of our stack,
   and return upper-cased copy of argv_string. */

static void *
thread_start(void *arg)
{
    struct thread_info *tinfo = arg;
    char *uargv;

    printf("Thread %d: top of stack near %p; argv_string=%s\n",
           tinfo->thread_num, (void *) &tinfo, tinfo->argv_string);

    uargv = strdup(tinfo->argv_string);
    if (uargv == NULL)
        handle_error("strdup");

    for (char *p = uargv; *p != '\0'; p++)
        *p = toupper(*p);

    return uargv;
}

int
main(int argc, char *argv[])
{
    int                 s, opt;
    void                *res;
    size_t              num_threads;
    ssize_t             stack_size;
    pthread_attr_t      attr;
    struct thread_info  *tinfo;

    /* The "-s" option specifies a stack size for our threads. */

    stack_size = -1;
```

```
        while ((opt = getopt(argc, argv, "s:")) != -1) {
            switch (opt) {
            case 's':
                stack_size = strtoul(optarg, NULL, 0);
                break;

            default:
                fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",
                        argv[0]);
                exit(EXIT_FAILURE);
            }
        }

        num_threads = argc - optind;

        /* Initialize thread creation attributes. */

        s = pthread_attr_init(&attr);
        if (s != 0)
            handle_error_en(s, "pthread_attr_init");

        if (stack_size > 0) {
            s = pthread_attr_setstacksize(&attr, stack_size);
            if (s != 0)
                handle_error_en(s, "pthread_attr_setstacksize");
        }

        /* Allocate memory for pthread_create() arguments. */

        tinfo = calloc(num_threads, sizeof(*tinfo));
        if (tinfo == NULL)
            handle_error("calloc");

        /* Create one thread for each command-line argument. */

        for (size_t tnum = 0; tnum < num_threads; tnum++) {
            tinfo[tnum].thread_num = tnum + 1;
            tinfo[tnum].argv_string = argv[optind + tnum];

            /* The pthread_create() call stores the thread ID into
               corresponding element of tinfo[]. */

            s = pthread_create(&tinfo[tnum].thread_id, &attr,
                               &thread_start, &tinfo[tnum]);
            if (s != 0)
                handle_error_en(s, "pthread_create");
        }

        /* Destroy the thread attributes object, since it is no
           longer needed. */

        s = pthread_attr_destroy(&attr);
        if (s != 0)
            handle_error_en(s, "pthread_attr_destroy");

        /* Now join with each thread, and display its returned value. */

        for (size_t tnum = 0; tnum < num_threads; tnum++) {
            s = pthread_join(tinfo[tnum].thread_id, &res);
```

```
            if (s != 0)
                handle_error_en(s, "pthread_join");

            printf("Joined with thread %d; returned value was %s\n",
                   tinfo[tnum].thread_num, (char *) res);
            free(res);      /* Free memory allocated by thread */
        }

        free(tinfo);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

getrlimit(2), *pthread_attr_init*(3), *pthread_cancel*(3), *pthread_detach*(3), *pthread_equal*(3), *pthread_exit*(3), *pthread_getattr_np*(3), *pthread_join*(3), *pthread_self*(3), *pthread_setattr_default_np*(3), *pthreads*(7)

NAME

pthread_detach − detach a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t *thread***);**

DESCRIPTION

The **pthread_detach**() function marks the thread identified by *thread* as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE

On success, **pthread_detach**() returns 0; on error, it returns an error number.

ERRORS

EINVAL

thread is not a joinable thread.

ESRCH

No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_detach()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Once a thread has been detached, it can't be joined with *pthread_join*(3) or be made joinable again.

A new thread can be created in a detached state using *pthread_attr_setdetachstate*(3) to set the detached attribute of the *attr* argument of *pthread_create*(3).

The detached attribute merely determines the behavior of the system when the thread terminates; it does not prevent the thread from being terminated if the process terminates using *exit*(3) (or equivalently, if the main thread returns).

Either *pthread_join*(3) or **pthread_detach**() should be called for each thread that an application creates, so that system resources for the thread can be released. (But note that the resources of any threads for which one of these actions has not been done will be freed when the process terminates.)

EXAMPLES

The following statement detaches the calling thread:

```
pthread_detach(pthread_self());
```

SEE ALSO

pthread_attr_setdetachstate(3), *pthread_cancel*(3), *pthread_create*(3), *pthread_exit*(3), *pthread_join*(3), *pthreads*(7)

NAME

pthread_equal − compare thread IDs

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_equal(pthread_t *t1*, **pthread_t** *t2*);

DESCRIPTION

The **pthread_equal**() function compares two thread identifiers.

RETURN VALUE

If the two thread IDs are equal, **pthread_equal**() returns a nonzero value; otherwise, it returns 0.

ERRORS

This function always succeeds.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_equal()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The **pthread_equal**() function is necessary because thread IDs should be considered opaque: there is no portable way for applications to directly compare two *pthread_t* values.

SEE ALSO

pthread_create(3), *pthread_self*(3), *pthreads*(7)

NAME

pthread_exit − terminate calling thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

[[noreturn]] void pthread_exit(void **retval*);

DESCRIPTION

The **pthread_exit**() function terminates the calling thread and returns a value via *retval* that (if the thread is joinable) is available to another thread in the same process that calls *pthread_join*(3).

Any clean-up handlers established by *pthread_cleanup_push*(3) that have not yet been popped, are popped (in the reverse of the order in which they were pushed) and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in an unspecified order.

When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released, and functions registered using *atexit*(3) are not called.

After the last thread in a process terminates, the process terminates as by calling *exit*(3) with an exit status of zero; thus, process-shared resources are released and functions registered using *atexit*(3) are called.

RETURN VALUE

This function does not return to the caller.

ERRORS

This function always succeeds.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_exit()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Performing a return from the start function of any thread other than the main thread results in an implicit call to **pthread_exit**(), using the function's return value as the thread's exit status.

To allow other threads to continue execution, the main thread should terminate by calling **pthread_exit**() rather than *exit*(3).

The value pointed to by *retval* should not be located on the calling thread's stack, since the contents of that stack are undefined after the thread terminates.

BUGS

Currently, there are limitations in the kernel implementation logic for *wait*(2)ing on a stopped thread group with a dead thread group leader. This can manifest in problems such as a locked terminal if a stop signal is sent to a foreground process whose thread group leader has already called **pthread_exit**().

SEE ALSO

pthread_create(3), *pthread_join*(3), *pthreads*(7)

NAME

pthread_getattr_default_np, pthread_setattr_default_np, − get or set default thread-creation attributes

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_getattr_default_np(pthread_attr_t *attr);
int pthread_setattr_default_np(const pthread_attr_t *attr);
```

DESCRIPTION

The **pthread_setattr_default_np**() function sets the default attributes used for creation of a new thread—that is, the attributes that are used when *pthread_create*(3) is called with a second argument that is NULL. The default attributes are set using the attributes supplied in **attr*, a previously initialized thread attributes object. Note the following details about the supplied attributes object:

- The attribute settings in the object must be valid.
- The *stack address* attribute must not be set in the object.
- Setting the *stack size* attribute to zero means leave the default stack size unchanged.

The **pthread_getattr_default_np**() function initializes the thread attributes object referred to by *attr* so that it contains the default attributes used for thread creation.

ERRORS

EINVAL
(**pthread_setattr_default_np**()) One of the attribute settings in *attr* is invalid, or the stack address attribute is set in *attr*.

ENOMEM
(**pthread_setattr_default_np**()) Insufficient memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_getattr_default_np(), **pthread_setattr_default_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in their names.

HISTORY

glibc 2.18.

EXAMPLES

The program below uses **pthread_getattr_default_np**() to fetch the default thread-creation attributes and then displays various settings from the returned thread attributes object. When running the program, we see the following output:

```
$ ./a.out
Stack size:          8388608
Guard size:          4096
Scheduling policy:   SCHED_OTHER
Scheduling priority: 0
Detach state:        JOINABLE
Inherit scheduler:   INHERIT
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

```
static void
display_pthread_attr(pthread_attr_t *attr)
{
    int s;
    size_t stacksize;
    size_t guardsize;
    int policy;
    struct sched_param schedparam;
    int detachstate;
    int inheritsched;

    s = pthread_attr_getstacksize(attr, &stacksize);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getstacksize");
    printf("Stack size:          %zu\n", stacksize);

    s = pthread_attr_getguardsize(attr, &guardsize);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getguardsize");
    printf("Guard size:          %zu\n", guardsize);

    s = pthread_attr_getschedpolicy(attr, &policy);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getschedpolicy");
    printf("Scheduling policy:   %s\n",
           (policy == SCHED_FIFO) ? "SCHED_FIFO" :
           (policy == SCHED_RR) ? "SCHED_RR" :
           (policy == SCHED_OTHER) ? "SCHED_OTHER" : "[unknown]");

    s = pthread_attr_getschedparam(attr, &schedparam);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getschedparam");
    printf("Scheduling priority: %d\n", schedparam.sched_priority);

    s = pthread_attr_getdetachstate(attr, &detachstate);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getdetachstate");
    printf("Detach state:        %s\n",
           (detachstate == PTHREAD_CREATE_DETACHED) ? "DETACHED" :
           (detachstate == PTHREAD_CREATE_JOINABLE) ? "JOINABLE" :
           "???");

    s = pthread_attr_getinheritsched(attr, &inheritsched);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getinheritsched");
    printf("Inherit scheduler:   %s\n",
           (inheritsched == PTHREAD_INHERIT_SCHED) ? "INHERIT" :
           (inheritsched == PTHREAD_EXPLICIT_SCHED) ? "EXPLICIT" :
           "???");
}

int
main(void)
{
    int s;
    pthread_attr_t attr;

    s = pthread_getattr_default_np(&attr);
```

```
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_getattr_default_np");

        display_pthread_attr(&attr);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

pthread_attr_getaffinity_np(3), *pthread_attr_getdetachstate*(3), *pthread_attr_getguardsize*(3), *pthread_attr_getinheritsched*(3), *pthread_attr_getschedparam*(3), *pthread_attr_getschedpolicy*(3), *pthread_attr_getscope*(3), *pthread_attr_getstack*(3), *pthread_attr_getstackaddr*(3), *pthread_attr_getstacksize*(3), *pthread_attr_init*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_getattr_np − get attributes of created thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_getattr_np(pthread_t thread, pthread_attr_t *attr);
```

DESCRIPTION

The **pthread_getattr_np**() function initializes the thread attributes object referred to by *attr* so that it contains actual attribute values describing the running thread *thread*.

The returned attribute values may differ from the corresponding attribute values passed in the *attr* object that was used to create the thread using *pthread_create*(3). In particular, the following attributes may differ:

- the detach state, since a joinable thread may have detached itself after creation;
- the stack size, which the implementation may align to a suitable boundary.
- and the guard size, which the implementation may round upward to a multiple of the page size, or ignore (i.e., treat as 0), if the application is allocating its own stack.

Furthermore, if the stack address attribute was not set in the thread attributes object used to create the thread, then the returned thread attributes object will report the actual stack address that the implementation selected for the thread.

When the thread attributes object returned by **pthread_getattr_np**() is no longer required, it should be destroyed using *pthread_attr_destroy*(3).

RETURN VALUE

On success, this function returns 0; on error, it returns a nonzero error number.

ERRORS

ENOMEM
Insufficient memory.

In addition, if *thread* refers to the main thread, then **pthread_getattr_np**() can fail because of errors from various underlying calls: *fopen*(3), if */proc/self/maps* can't be opened; and *getrlimit*(2), if the **RLIMIT_STACK** resource limit is not supported.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_getattr_np()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the name.

HISTORY

glibc 2.2.3.

EXAMPLES

The program below demonstrates the use of **pthread_getattr_np**(). The program creates a thread that then uses **pthread_getattr_np**() to retrieve and display its guard size, stack address, and stack size attributes. Command-line arguments can be used to set these attributes to values other than the default when creating the thread. The shell sessions below demonstrate the use of the program.

In the first run, on an x86-32 system, a thread is created using default attributes:

```
$ ulimit -s      # No stack limit ==> default stack size is 2 MB
unlimited
$ ./a.out
Attributes of created thread:
        Guard size          = 4096 bytes
        Stack address       = 0x40196000 (EOS = 0x40397000)
```

```
        Stack size          = 0x201000 (2101248) bytes
```

In the following run, we see that if a guard size is specified, it is rounded up to the next multiple of the system page size (4096 bytes on x86-32):

```
$ ./a.out -g 4097
Thread attributes object after initializations:
        Guard size          = 4097 bytes
        Stack address       = (nil)
        Stack size          = 0x0 (0) bytes

Attributes of created thread:
        Guard size          = 8192 bytes
        Stack address       = 0x40196000 (EOS = 0x40397000)
        Stack size          = 0x201000 (2101248) bytes
```

In the last run, the program manually allocates a stack for the thread. In this case, the guard size attribute is ignored.

```
$ ./a.out -g 4096 -s 0x8000 -a
Allocated thread stack at 0x804d000

Thread attributes object after initializations:
        Guard size          = 4096 bytes
        Stack address       = 0x804d000 (EOS = 0x8055000)
        Stack size          = 0x8000 (32768) bytes

Attributes of created thread:
        Guard size          = 0 bytes
        Stack address       = 0x804d000 (EOS = 0x8055000)
        Stack size          = 0x8000 (32768) bytes
```

Program source

```
#define _GNU_SOURCE     /* To get pthread_getattr_np() declaration */
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
display_stack_related_attributes(pthread_attr_t *attr, char *prefix)
{
    int s;
    size_t stack_size, guard_size;
    void *stack_addr;

    s = pthread_attr_getguardsize(attr, &guard_size);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getguardsize");
    printf("%sGuard size          = %zu bytes\n", prefix, guard_size);

    s = pthread_attr_getstack(attr, &stack_addr, &stack_size);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_getstack");
    printf("%sStack address       = %p", prefix, stack_addr);
    if (stack_size > 0)
        printf(" (EOS = %p)", (char *) stack_addr + stack_size);
    printf("\n");
    printf("%sStack size          = %#zx (%zu) bytes\n",
```

```
            prefix, stack_size, stack_size);
}

static void
display_thread_attributes(pthread_t thread, char *prefix)
{
    int s;
    pthread_attr_t attr;

    s = pthread_getattr_np(thread, &attr);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_getattr_np");

    display_stack_related_attributes(&attr, prefix);

    s = pthread_attr_destroy(&attr);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_attr_destroy");
}

static void *           /* Start function for thread we create */
thread_start(void *arg)
{
    printf("Attributes of created thread:\n");
    display_thread_attributes(pthread_self(), "\t");

    exit(EXIT_SUCCESS);         /* Terminate all threads */
}

static void
usage(char *pname, char *msg)
{
    if (msg != NULL)
        fputs(msg, stderr);
    fprintf(stderr, "Usage: %s [-s stack-size [-a]]"
            " [-g guard-size]\n", pname);
    fprintf(stderr, "\t\t-a means program should allocate stack\n");
    exit(EXIT_FAILURE);
}

static pthread_attr_t *   /* Get thread attributes from command line */
get_thread_attributes_from_cl(int argc, char *argv[],
                              pthread_attr_t *attrp)
{
    int s, opt, allocate_stack;
    size_t stack_size, guard_size;
    void *stack_addr;
    pthread_attr_t *ret_attrp = NULL;   /* Set to attrp if we initialize
                                           a thread attributes object */
    allocate_stack = 0;
    stack_size = -1;
    guard_size = -1;

    while ((opt = getopt(argc, argv, "ag:s:")) != -1) {
        switch (opt) {
        case 'a':   allocate_stack = 1;                     break;
        case 'g':   guard_size = strtoul(optarg, NULL, 0);  break;
        case 's':   stack_size = strtoul(optarg, NULL, 0);  break;
        default:    usage(argv[0], NULL);
```

```
            }
        }

        if (allocate_stack && stack_size == -1)
            usage(argv[0], "Specifying -a without -s makes no sense\n");

        if (argc > optind)
            usage(argv[0], "Extraneous command-line arguments\n");

        if (stack_size != -1 || guard_size > 0) {
            ret_attrp = attrp;

            s = pthread_attr_init(attrp);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_init");
        }

        if (stack_size != -1) {
            if (!allocate_stack) {
                s = pthread_attr_setstacksize(attrp, stack_size);
                if (s != 0)
                    errc(EXIT_FAILURE, s, "pthread_attr_setstacksize");
            } else {
                s = posix_memalign(&stack_addr, sysconf(_SC_PAGESIZE),
                                   stack_size);
                if (s != 0)
                    errc(EXIT_FAILURE, s, "posix_memalign");
                printf("Allocated thread stack at %p\n\n", stack_addr);

                s = pthread_attr_setstack(attrp, stack_addr, stack_size);
                if (s != 0)
                    errc(EXIT_FAILURE, s, "pthread_attr_setstacksize");
            }
        }

        if (guard_size != -1) {
            s = pthread_attr_setguardsize(attrp, guard_size);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_setstacksize");
        }

        return ret_attrp;
    }

    int
    main(int argc, char *argv[])
    {
        int s;
        pthread_t thr;
        pthread_attr_t attr;
        pthread_attr_t *attrp = NULL;    /* Set to &attr if we initialize
                                            a thread attributes object */

        attrp = get_thread_attributes_from_cl(argc, argv, &attr);

        if (attrp != NULL) {
            printf("Thread attributes object after initializations:\n");
            display_stack_related_attributes(attrp, "\t");
            printf("\n");
```

```
        }

        s = pthread_create(&thr, attrp, &thread_start, NULL);
        if (s != 0)
            errc(EXIT_FAILURE, s, "pthread_create");

        if (attrp != NULL) {
            s = pthread_attr_destroy(attrp);
            if (s != 0)
                errc(EXIT_FAILURE, s, "pthread_attr_destroy");
        }

        pause();    /* Terminates when other thread calls exit() */
    }
```

SEE ALSO

pthread_attr_getaffinity_np(3), *pthread_attr_getdetachstate*(3), *pthread_attr_getguardsize*(3), *pthread_attr_getinheritsched*(3), *pthread_attr_getschedparam*(3), *pthread_attr_getschedpolicy*(3), *pthread_attr_getscope*(3), *pthread_attr_getstack*(3), *pthread_attr_getstackaddr*(3), *pthread_attr_getstacksize*(3), *pthread_attr_init*(3), *pthread_create*(3), *pthreads*(7)

NAME

pthread_getcpuclockid − retrieve ID of a thread's CPU time clock

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread, clockid_t *clockid);
```

DESCRIPTION

The **pthread_getcpuclockid**() function obtains the ID of the CPU-time clock of the thread whose ID is given in *thread*, and returns it in the location pointed to by *clockid*.

RETURN VALUE

On success, this function returns 0; on error, it returns a nonzero error number.

ERRORS

ENOENT
: Per-thread CPU time clocks are not supported by the system.

ESRCH
: No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_getcpuclockid()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

NOTES

When *thread* refers to the calling thread, this function returns an identifier that refers to the same clock manipulated by *clock_gettime*(2) and *clock_settime*(2) when given the clock ID **CLOCK_THREAD_CPUTIME_ID**.

EXAMPLES

The program below creates a thread and then uses *clock_gettime*(2) to retrieve the total process CPU time, and the per-thread CPU time consumed by the two threads. The following shell session shows an example run:

```
$ ./a.out
Main thread sleeping
Subthread starting infinite loop
Main thread consuming some CPU time...
Process total CPU time:    1.368
Main thread CPU time:      0.376
Subthread CPU time:        0.992
```

Program source

```
/* Link with "-lrt" */

#include <errno.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
```

```
#define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while (0)

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void *
thread_start(void *arg)
{
    printf("Subthread starting infinite loop\n");
    for (;;)
        continue;
}

static void
pclock(char *msg, clockid_t cid)
{
    struct timespec ts;

    printf("%s", msg);
    if (clock_gettime(cid, &ts) == -1)
        handle_error("clock_gettime");
    printf("%4jd.%03ld\n", (intmax_t) ts.tv_sec, ts.tv_nsec / 1000000);
}

int
main(void)
{
    pthread_t thread;
    clockid_t cid;
    int s;

    s = pthread_create(&thread, NULL, thread_start, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    printf("Main thread sleeping\n");
    sleep(1);

    printf("Main thread consuming some CPU time...\n");
    for (unsigned int j = 0; j < 2000000; j++)
        getppid();

    pclock("Process total CPU time: ", CLOCK_PROCESS_CPUTIME_ID);

    s = pthread_getcpuclockid(pthread_self(), &cid);
    if (s != 0)
        handle_error_en(s, "pthread_getcpuclockid");
    pclock("Main thread CPU time:   ", cid);

    /* The preceding 4 lines of code could have been replaced by:
       pclock("Main thread CPU time:   ", CLOCK_THREAD_CPUTIME_ID); */

    s = pthread_getcpuclockid(thread, &cid);
    if (s != 0)
        handle_error_en(s, "pthread_getcpuclockid");
    pclock("Subthread CPU time: 1    ", cid);
```

```
        exit(EXIT_SUCCESS);          /* Terminates both threads */
    }
```

SEE ALSO

clock_gettime(2), *clock_settime*(2), *timer_create*(2), *clock_getcpuclockid*(3), *pthread_self*(3), *pthreads*(7), *time*(7)

NAME

pthread_join − join with a terminated thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread***, void ****retval**);**

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to *pthread_exit*(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK
A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL
thread is not a joinable thread.

EINVAL
Another thread is already waiting to join with this thread.

ESRCH
No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_join()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of *waitpid(−1, &status, 0)*, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLES

See *pthread_create*(3).

SEE ALSO

pthread_cancel(3), *pthread_create*(3), *pthread_detach*(3), *pthread_exit*(3), *pthread_tryjoin_np*(3), *pthreads*(7)

NAME

pthread_key_create, pthread_key_delete, pthread_setspecific, pthread_getspecific − management of thread-specific data

SYNOPSIS

#include <pthread.h>

int pthread_key_create(pthread_key_t *****key**, void (*****destr_function****) (void *));** **int pthread_key_delete(pthread_key_t** *key***);** **int pthread_setspecific(pthread_key_t** *key***, const void *** *pointer***);** **void * pthread_getspecific(pthread_key_t** *key***);**

DESCRIPTION

Programs often need global or static variables that have different values in different threads. Since threads share one memory space, this cannot be achieved with regular variables. Thread-specific data is the POSIX threads answer to this need.

Each thread possesses a private memory block, the thread-specific data area, or TSD area for short. This area is indexed by TSD keys. The TSD area associates values of type **void *** to TSD keys. TSD keys are common to all threads, but the value associated with a given TSD key can be different in each thread.

For concreteness, the TSD areas can be viewed as arrays of **void *** pointers, TSD keys as integer indices into these arrays, and the value of a TSD key as the value of the corresponding array element in the calling thread.

When a thread is created, its TSD area initially associates **NULL** with all keys.

pthread_key_create allocates a new TSD key. The key is stored in the location pointed to by *key*. There is a limit of **PTHREAD_KEYS_MAX** on the number of keys allocated at a given time. The value initially associated with the returned key is **NULL** in all currently executing threads.

The *destr_function* argument, if not **NULL**, specifies a destructor function associated with the key. When a thread terminates via **pthread_exit** or by cancelation, *destr_function* is called with arguments the value associated with the key in that thread. The *destr_function* is not called if that value is **NULL**. The order in which destructor functions are called at thread termination time is unspecified.

Before the destructor function is called, the **NULL** value is associated with the key in the current thread. A destructor function might, however, re-associate non-**NULL** values to that key or some other key. To deal with this, if after all the destructors have been called for all non-**NULL** values, there are still some non-**NULL** values with associated destructors, then the process is repeated. The glibc implementation stops the process after **PTHREAD_DESTRUCTOR_ITERATIONS** iterations, even if some non-**NULL** values with associated descriptors remain. Other implementations may loop indefinitely.

pthread_key_delete deallocates a TSD key. It does not check whether non-**NULL** values are associated with that key in the currently executing threads, nor call the destructor function associated with the key.

pthread_setspecific changes the value associated with *key* in the calling thread, storing the given *pointer* instead.

pthread_getspecific returns the value currently associated with *key* in the calling thread.

RETURN VALUE

pthread_key_create, **pthread_key_delete**, and **pthread_setspecific** return 0 on success and a nonzero error code on failure. If successful, **pthread_key_create** stores the newly allocated key in the location pointed to by its *key* argument.

pthread_getspecific returns the value associated with *key* on success, and **NULL** on error.

ERRORS

pthread_key_create returns the following error code on error:

EAGAIN
PTHREAD_KEYS_MAX keys are already allocated.

pthread_key_delete and **pthread_setspecific** return the following error code on error:

EINVAL
key is not a valid, allocated TSD key.

pthread_getspecific returns **NULL** if *key* is not a valid, allocated TSD key.

SEE ALSO

pthread_create(3), pthread_exit(3), pthread_testcancel(3).

EXAMPLE

The following code fragment allocates a thread-specific array of 100 characters, with automatic reclamation at thread exit:

```
/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;

/* Once-only initialisation of the key */
static pthread_once_t buffer_key_once = PTHREAD_ONCE_INIT;

/* Allocate the thread-specific buffer */
void buffer_alloc(void)
{
  pthread_once(&buffer_key_once, buffer_key_alloc);
  pthread_setspecific(buffer_key, malloc(100));
}

/* Return the thread-specific buffer */
char * get_buffer(void)
{
  return (char *) pthread_getspecific(buffer_key);
}

/* Allocate the key */
static void buffer_key_alloc()
{
  pthread_key_create(&buffer_key, buffer_destroy);
}

/* Free the thread-specific buffer */
static void buffer_destroy(void * buf)
{
  free(buf);
}
```

NAME

pthread_kill − send a signal to a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <signal.h>

int pthread_kill(pthread_t *thread***, int** *sig***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_kill():
_POSIX_C_SOURCE >= 199506L || _XOPEN_SOURCE >= 500

DESCRIPTION

The **pthread_kill**() function sends the signal *sig* to *thread*, a thread in the same process as the caller. The signal is asynchronously directed to *thread*.

If *sig* is 0, then no signal is sent, but error checking is still performed.

RETURN VALUE

On success, **pthread_kill**() returns 0; on error, it returns an error number, and no signal is sent.

ERRORS

EINVAL
An invalid signal was specified.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_kill()	Thread safety	MT-Safe

VERSIONS

The glibc implementation of **pthread_kill**() gives an error (**EINVAL**) on attempts to send either of the real-time signals used internally by the NPTL threading implementation. See *nptl*(7) for details.

POSIX.1-2008 recommends that if an implementation detects the use of a thread ID after the end of its lifetime, **pthread_kill**() should return the error **ESRCH**. The glibc implementation returns this error in the cases where an invalid thread ID can be detected. But note also that POSIX says that an attempt to use a thread ID whose lifetime has ended produces undefined behavior, and an attempt to use an invalid thread ID in a call to **pthread_kill**() can, for example, cause a segmentation fault.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Signal dispositions are process-wide: if a signal handler is installed, the handler will be invoked in the thread *thread*, but if the disposition of the signal is "stop", "continue", or "terminate", this action will affect the whole process.

SEE ALSO

kill(2), *sigaction*(2), *sigpending*(2), *pthread_self*(3), *pthread_sigmask*(3), *raise*(3), *pthreads*(7), *signal*(7)

NAME

pthread_kill_other_threads_np − terminate all other threads in process

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

void pthread_kill_other_threads_np(void);

DESCRIPTION

pthread_kill_other_threads_np() has an effect only in the LinuxThreads threading implementation. On that implementation, calling this function causes the immediate termination of all threads in the application, except the calling thread. The cancelation state and cancelation type of the to-be-terminated threads are ignored, and the cleanup handlers are not called in those threads.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_kill_other_threads_np()	Thread safety	MT-Safe

VERSIONS

In the NPTL threading implementation, **pthread_kill_other_threads_np**() exists, but does nothing. (Nothing needs to be done, because the implementation does the right thing during an *execve*(2).)

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the name.

HISTORY

glibc 2.0

NOTES

pthread_kill_other_threads_np() is intended to be called just before a thread calls *execve*(2) or a similar function. This function is designed to address a limitation in the obsolete LinuxThreads implementation whereby the other threads of an application are not automatically terminated (as POSIX.1-2001 requires) during *execve*(2).

SEE ALSO

execve(2), *pthread_cancel*(3), *pthread_setcancelstate*(3), *pthread_setcanceltype*(3), *pthreads*(7)

NAME

pthread_mutex_consistent − make a robust mutex consistent

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t ****mutex*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_mutex_consistent():
_POSIX_C_SOURCE >= 200809L

DESCRIPTION

This function makes a robust mutex consistent if it is in an inconsistent state. A mutex can be left in an inconsistent state if its owner terminates while holding the mutex, in which case the next owner who acquires the mutex will succeed and be notified by a return value of **EOWNERDEAD** from a call to **pthread_mutex_lock**().

RETURN VALUE

On success, *pthread_mutex_consistent*() returns 0. Otherwise, it returns a positive error number to indicate the error.

ERRORS

EINVAL
The mutex is either not robust or is not in an inconsistent state.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.12. POSIX.1-2008.

Before the addition of **pthread_mutex_consistent**() to POSIX, glibc defined the following equivalent nonstandard function if **_GNU_SOURCE** was defined:

[[deprecated]]
int pthread_mutex_consistent_np(const pthread_mutex_t ****mutex*);**

This GNU-specific API, which first appeared in glibc 2.4, is nowadays obsolete and should not be used in new programs; since glibc 2.34 it has been marked as deprecated.

NOTES

pthread_mutex_consistent() simply informs the implementation that the state (shared data) guarded by the mutex has been restored to a consistent state and that normal operations can now be performed with the mutex. It is the application's responsibility to ensure that the shared data has been restored to a consistent state before calling **pthread_mutex_consistent**().

EXAMPLES

See *pthread_mutexattr_setrobust*(3).

SEE ALSO

pthread_mutex_lock(3), *pthread_mutexattr_getrobust*(3), *pthread_mutexattr_init*(3), *pthread_mutexattr_setrobust*(3), *pthreads*(7)

NAME

pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, pthread_mutex_destroy − operations on mutexes

SYNOPSIS

#include <pthread.h>

pthread_mutex_t *fastmutex* **= PTHREAD_MUTEX_INITIALIZER; pthread_mutex_t** *recmutex* **= PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP; pthread_mutex_t** *errchkmutex* **= PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;**

int pthread_mutex_init(pthread_mutex_t ***_mutex_, const pthread_mutexattr_t ***_mutexattr_**); int pthread_mutex_lock(pthread_mutex_t ***_mutex_**); int pthread_mutex_trylock(pthread_mutex_t ***_mutex_**); int pthread_mutex_unlock(pthread_mutex_t ***_mutex_**); int pthread_mutex_destroy(pthread_mutex_t ***_mutex_**);**

DESCRIPTION

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

pthread_mutex_init initializes the mutex object pointed to by *mutex* according to the mutex attributes specified in *mutexattr*. If *mutexattr* is **NULL**, default attributes are used instead.

The LinuxThreads implementation supports only one mutex attributes, the *mutex kind*, which is either ‘‘fast’’, ‘‘recursive’’, or ‘‘error checking’’. The kind of a mutex determines whether it can be locked again by a thread that already owns it. The default kind is ‘‘fast’’. See **pthread_mutexattr_init**(3) for more information on mutex attributes.

Variables of type **pthread_mutex_t** can also be initialized statically, using the constants **PTHREAD_MUTEX_INITIALIZER** (for fast mutexes), **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP** (for recursive mutexes), and **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP** (for error checking mutexes).

pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and **pthread_mutex_lock** returns immediately. If the mutex is already locked by another thread, **pthread_mutex_lock** suspends the calling thread until the mutex is unlocked.

If the mutex is already locked by the calling thread, the behavior of **pthread_mutex_lock** depends on the kind of the mutex. If the mutex is of the ‘‘fast’’ kind, the calling thread is suspended until the mutex is unlocked, thus effectively causing the calling thread to deadlock. If the mutex is of the ‘‘error checking’’ kind, **pthread_mutex_lock** returns immediately with the error code **EDEADLK**. If the mutex is of the ‘‘recursive’’ kind, **pthread_mutex_lock** succeeds and returns immediately, recording the number of times the calling thread has locked the mutex. An equal number of **pthread_mutex_unlock** operations must be performed before the mutex returns to the unlocked state.

pthread_mutex_trylock behaves identically to **pthread_mutex_lock**, except that it does not block the calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a ‘‘fast’’ mutex). Instead, **pthread_mutex_trylock** returns immediately with the error code **EBUSY**.

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to **pthread_mutex_unlock**. If the mutex is of the ‘‘fast’’ kind, **pthread_mutex_unlock** always returns it to the unlocked state. If it is of the ‘‘recursive’’ kind, it decrements the locking count of the mutex (number of **pthread_mutex_lock** operations performed on it by the calling thread), and only when this count reaches zero is the mutex actually unlocked.

On ‘‘error checking’’ and ‘‘recursive’’ mutexes, **pthread_mutex_unlock** actually checks at run-time that the mutex is locked on entrance, and that it was locked by the same thread that is now calling **pthread_mutex_unlock**. If these conditions are not met, an error code is returned and the mutex remains unchanged. ‘‘Fast’’ mutexes perform no such checks, thus allowing a locked mutex to be unlocked by a thread other than its owner. This is non-portable behavior and must not be relied upon.

pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance. In the LinuxThreads implementation, no resources are associated with mutex objects, thus **pthread_mutex_destroy** actually does nothing except checking that the mutex is unlocked.

CANCELLATION

None of the mutex functions is a cancelation point, not even **pthread_mutex_lock**, in spite of the fact that it can suspend a thread for arbitrary durations. This way, the status of mutexes at cancelation points is predictable, allowing cancelation handlers to unlock precisely those mutexes that need to be unlocked before the thread stops executing. Consequently, threads using deferred cancelation should never hold a mutex for extended periods of time.

ASYNC-SIGNAL SAFETY

The mutex functions are not async-signal safe. What this means is that they should not be called from a signal handler. In particular, calling **pthread_mutex_lock** or **pthread_mutex_unlock** from a signal handler may deadlock the calling thread.

RETURN VALUE

pthread_mutex_init always returns 0. The other mutex functions return 0 on success and a non-zero error code on error.

ERRORS

The **pthread_mutex_lock** function returns the following error code on error:

EINVAL
The mutex has not been properly initialized.

EDEADLK
The mutex is already locked by the calling thread (‘‘error checking’’ mutexes only).

The **pthread_mutex_trylock** function returns the following error codes on error:

EBUSY
The mutex could not be acquired because it was currently locked.

EINVAL
The mutex has not been properly initialized.

The **pthread_mutex_unlock** function returns the following error code on error:

EINVAL
The mutex has not been properly initialized.

EPERM
The calling thread does not own the mutex (‘‘error checking’’ mutexes only).

The **pthread_mutex_destroy** function returns the following error code on error:

EBUSY
The mutex is currently locked.

SEE ALSO

pthread_mutexattr_init(3), **pthread_mutexattr_setkind_np**(3), **pthread_cancel**(3).

EXAMPLE

A shared global variable *x* can be protected by a mutex as follows:

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

All accesses and modifications to *x* should be bracketed by calls to **pthread_mutex_lock** and **pthread_mutex_unlock** as follows:

```
pthread_mutex_lock(&mut);
/* operate on x */
pthread_mutex_unlock(&mut);
```

NAME

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared − get/set process-shared mutex attribute

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getpshared(
        const pthread_mutexattr_t *restrict attr,
        int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
        int pshared);
```

DESCRIPTION

These functions get and set the process-shared attribute in a mutex attributes object. This attribute must be appropriately set to ensure correct, efficient operation of a mutex created using this attributes object.

The process-shared attribute can have one of the following values:

PTHREAD_PROCESS_PRIVATE
: Mutexes created with this attributes object are to be shared only among threads in the same process that initialized the mutex. This is the default value for the process-shared mutex attribute.

PTHREAD_PROCESS_SHARED
: Mutexes created with this attributes object can be shared between any threads that have access to the memory containing the object, including threads in different processes.

pthread_mutexattr_getpshared() places the value of the process-shared attribute of the mutex attributes object referred to by *attr* in the location pointed to by *pshared*.

pthread_mutexattr_setpshared() sets the value of the process-shared attribute of the mutex attributes object referred to by *attr* to the value specified in **pshared**.

If *attr* does not refer to an initialized mutex attributes object, the behavior is undefined.

RETURN VALUE

On success, these functions return 0. On error, they return a positive error number.

ERRORS

pthread_mutexattr_setpshared() can fail with the following errors:

EINVAL
: The value specified in *pshared* is invalid.

ENOTSUP
: *pshared is* **PTHREAD_PROCESS_SHARED** but the implementation does not support process-shared mutexes.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

pthread_mutexattr_init(3), *pthreads*(7)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy − initialize and destroy a mutex attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

DESCRIPTION

The **pthread_mutexattr_init**() function initializes the mutex attributes object pointed to by *attr* with default values for all attributes defined by the implementation.

The results of initializing an already initialized mutex attributes object are undefined.

The **pthread_mutexattr_destroy**() function destroys a mutex attribute object (making it uninitialized). Once a mutex attributes object has been destroyed, it can be reinitialized with **pthread_mutexattr_init**().

The results of destroying an uninitialized mutex attributes object are undefined.

RETURN VALUE

On success, these functions return 0. On error, they return a positive error number.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

Subsequent changes to a mutex attributes object do not affect mutex that have already been initialized using that object.

SEE ALSO

pthread_mutex_init(3), *pthread_mutexattr_getpshared*(3), *pthread_mutexattr_getrobust*(3), *pthreads*(7)

NAME

pthread_mutexattr_setkind_np, pthread_mutexattr_getkind_np − deprecated mutex creation attributes

SYNOPSIS

#include <pthread.h>

int pthread_mutexattr_setkind_np(pthread_mutexattr_t **attr*, int *kind*); int pthread_mutexattr_getkind_np(const pthread_mutexattr_t **attr*, int **kind*);

DESCRIPTION

These functions are deprecated, use **pthread_mutexattr_settype**(3) and **pthread_mutexattr_gettype**(3) instead.

RETURN VALUE

pthread_mutexattr_getkind_np always returns 0.

pthread_mutexattr_setkind_np returns 0 on success and a non-zero error code on error.

ERRORS

On error, **pthread_mutexattr_setkind_np** returns the following error code:

EINVAL

kind is neither **PTHREAD_MUTEX_FAST_NP** nor **PTHREAD_MUTEX_RECURSIVE_NP** nor **PTHREAD_MUTEX_ERRORCHECK_NP**.

SEE ALSO

pthread_mutexattr_settype(3), **pthread_mutexattr_gettype**(3).

NAME

pthread_mutexattr_getrobust, pthread_mutexattr_setrobust − get and set the robustness attribute of a mutex attributes object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *attr,
                                int *robustness);
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                int robustness);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_mutexattr_getrobust(), **pthread_mutexattr_setrobust**():
_POSIX_C_SOURCE >= 200809L

DESCRIPTION

The **pthread_mutexattr_getrobust**() function places the value of the robustness attribute of the mutex attributes object referred to by *attr* in **robustness*. The **pthread_mutexattr_setrobust**() function sets the value of the robustness attribute of the mutex attributes object referred to by *attr* to the value specified in **robustness*.

The robustness attribute specifies the behavior of the mutex when the owning thread dies without unlocking the mutex. The following values are valid for *robustness*:

PTHREAD_MUTEX_STALLED
: This is the default value for a mutex attributes object. If a mutex is initialized with the **PTHREAD_MUTEX_STALLED** attribute and its owner dies without unlocking it, the mutex remains locked afterwards and any future attempts to call *pthread_mutex_lock*(3) on the mutex will block indefinitely.

PTHREAD_MUTEX_ROBUST
: If a mutex is initialized with the **PTHREAD_MUTEX_ROBUST** attribute and its owner dies without unlocking it, any future attempts to call *pthread_mutex_lock*(3) on this mutex will succeed and return **EOWNERDEAD** to indicate that the original owner no longer exists and the mutex is in an inconsistent state. Usually after **EOWNERDEAD** is returned, the next owner should call *pthread_mutex_consistent*(3) on the acquired mutex to make it consistent again before using it any further.

  If the next owner unlocks the mutex using *pthread_mutex_unlock*(3) before making it consistent, the mutex will be permanently unusable and any subsequent attempts to lock it using *pthread_mutex_lock*(3) will fail with the error **ENOTRECOVERABLE**. The only permitted operation on such a mutex is *pthread_mutex_destroy*(3)

  If the next owner terminates before calling *pthread_mutex_consistent*(3), further *pthread_mutex_lock*(3) operations on this mutex will still return **EOWNERDEAD**.

Note that the *attr* argument of **pthread_mutexattr_getrobust**() and **pthread_mutexattr_setrobust**() should refer to a mutex attributes object that was initialized by *pthread_mutexattr_init*(3), otherwise the behavior is undefined.

RETURN VALUE

On success, these functions return 0. On error, they return a positive error number.

In the glibc implementation, **pthread_mutexattr_getrobust**() always return zero.

ERRORS

EINVAL
: A value other than **PTHREAD_MUTEX_STALLED** or **PTHREAD_MUTEX_ROBUST** was passed to **pthread_mutexattr_setrobust**().

VERSIONS

In the Linux implementation, when using process-shared robust mutexes, a waiting thread also receives the **EOWNERDEAD** notification if the owner of a robust mutex performs an *execve*(2) without first

unlocking the mutex. POSIX.1 does not specify this detail, but the same behavior also occurs in at least some other implementations.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.12. POSIX.1-2008.

Before the addition of **pthread_mutexattr_getrobust**() and **pthread_mutexattr_setrobust**() to POSIX, glibc defined the following equivalent nonstandard functions if **_GNU_SOURCE** was defined:

[[deprecated]]
int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t ****attr*,**
int ****robustness*);**
[[deprecated]]
int pthread_mutexattr_setrobust_np(const pthread_mutexattr_t ****attr*,**
int *robustness***);**

Correspondingly, the constants **PTHREAD_MUTEX_STALLED_NP** and **PTHREAD_MUTEX_ROBUST_NP** were also defined.

These GNU-specific APIs, which first appeared in glibc 2.4, are nowadays obsolete and should not be used in new programs; since glibc 2.34 these APIs are marked as deprecated.

EXAMPLES

The program below demonstrates the use of the robustness attribute of a mutex attributes object. In this program, a thread holding the mutex dies prematurely without unlocking the mutex. The main thread subsequently acquires the mutex successfully and gets the error **EOWNERDEAD**, after which it makes the mutex consistent.

The following shell session shows what we see when running this program:

```
$ ./a.out
[original owner] Setting lock...
[original owner] Locked. Now exiting without unlocking.
[main] Attempting to lock the robust mutex.
[main] pthread_mutex_lock() returned EOWNERDEAD
[main] Now make the mutex consistent
[main] Mutex is now consistent; unlocking
```

Program source

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static pthread_mutex_t mtx;

static void *
original_owner_thread(void *ptr)
{
    printf("[original owner] Setting lock...\n");
    pthread_mutex_lock(&mtx);
    printf("[original owner] Locked. Now exiting without unlocking.\n");
    pthread_exit(NULL);
}

int
main(void)
{
```

```
        pthread_t thr;
        pthread_mutexattr_t attr;
        int s;

        pthread_mutexattr_init(&attr);

        pthread_mutexattr_setrobust(&attr, PTHREAD_MUTEX_ROBUST);

        pthread_mutex_init(&mtx, &attr);

        pthread_create(&thr, NULL, original_owner_thread, NULL);

        sleep(2);

        /* "original_owner_thread" should have exited by now. */

        printf("[main] Attempting to lock the robust mutex.\n");
        s = pthread_mutex_lock(&mtx);
        if (s == EOWNERDEAD) {
            printf("[main] pthread_mutex_lock() returned EOWNERDEAD\n");
            printf("[main] Now make the mutex consistent\n");
            s = pthread_mutex_consistent(&mtx);
            if (s != 0)
                handle_error_en(s, "pthread_mutex_consistent");
            printf("[main] Mutex is now consistent; unlocking\n");
            s = pthread_mutex_unlock(&mtx);
            if (s != 0)
                handle_error_en(s, "pthread_mutex_unlock");

            exit(EXIT_SUCCESS);
        } else if (s == 0) {
            printf("[main] pthread_mutex_lock() unexpectedly succeeded\n");
            exit(EXIT_FAILURE);
        } else {
            printf("[main] pthread_mutex_lock() unexpectedly failed\n");
            handle_error_en(s, "pthread_mutex_lock");
        }
    }
```

SEE ALSO

get_robust_list(2), *set_robust_list*(2), *pthread_mutex_consistent*(3), *pthread_mutex_init*(3), *pthread_mutex_lock*(3), *pthreads*(7)

NAME

pthread_once − once-only initialization

SYNOPSIS

#include <pthread.h>

pthread_once_t *once_control* **= PTHREAD_ONCE_INIT;**

int pthread_once(pthread_once_t ***_once_control_, void (***_init_routine_**) (void));**

DESCRIPTION

The purpose of **pthread_once** is to ensure that a piece of initialization code is executed at most once. The *once_control* argument points to a static or extern variable statically initialized to **PTHREAD_ONCE_INIT**.

The first time **pthread_once** is called with a given *once_control* argument, it calls *init_routine* with no argument and changes the value of the *once_control* variable to record that initialization has been performed. Subsequent calls to **pthread_once** with the same **once_control** argument do nothing.

RETURN VALUE

pthread_once always returns 0.

ERRORS

None.

NAME

pthread_rwlockattr_setkind_np, pthread_rwlockattr_getkind_np − set/get the read-write lock kind of the thread read-write lock attribute object

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t *attr,
                                  int pref);
int pthread_rwlockattr_getkind_np(
                                  const pthread_rwlockattr_t *restrict attr,
                                  int *restrict pref);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_rwlockattr_setkind_np(), **pthread_rwlockattr_getkind_np**():
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L

DESCRIPTION

The **pthread_rwlockattr_setkind_np**() function sets the "lock kind" attribute of the read-write lock attribute object referred to by *attr* to the value specified in *pref*. The argument *pref* may be set to one of the following:

PTHREAD_RWLOCK_PREFER_READER_NP
: This is the default. A thread may hold multiple read locks; that is, read locks are recursive. According to The Single Unix Specification, the behavior is unspecified when a reader tries to place a lock, and there is no write lock but writers are waiting. Giving preference to the reader, as is set by **PTHREAD_RWLOCK_PREFER_READER_NP**, implies that the reader will receive the requested lock, even if a writer is waiting. As long as there are readers, the writer will be starved.

PTHREAD_RWLOCK_PREFER_WRITER_NP
: This is intended as the write lock analog of **PTHREAD_RWLOCK_PREFER_READER_NP**. This is ignored by glibc because the POSIX requirement to support recursive read locks would cause this option to create trivial deadlocks; instead use **PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP** which ensures the application developer will not take recursive read locks thus avoiding deadlocks.

PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP
: Setting the lock kind to this avoids writer starvation as long as any read locking is not done in a recursive fashion.

The **pthread_rwlockattr_getkind_np**() function returns the value of the lock kind attribute of the read-write lock attribute object referred to by *attr* in the pointer *pref*.

RETURN VALUE

On success, these functions return 0. Given valid pointer arguments, **pthread_rwlockattr_getkind_np**() always succeeds. On error, **pthread_rwlockattr_setkind_np**() returns a nonzero error number.

ERRORS

EINVAL
: *pref* specifies an unsupported value.

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.1.

SEE ALSO

pthreads(7)

NAME

pthread_self − obtain ID of the calling thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

DESCRIPTION

The **pthread_self**() function returns the ID of the calling thread. This is the same value that is returned in **thread* in the *pthread_create*(3) call that created this thread.

RETURN VALUE

This function always succeeds, returning the calling thread's ID.

ERRORS

This function always succeeds.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_self()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID; for example, representation using either an arithmetic type or a structure is permitted. Therefore, variables of type *pthread_t* can't portably be compared using the C equality operator (==); use *pthread_equal*(3) instead.

Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.

Thread IDs are guaranteed to be unique only within a process. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

The thread ID returned by **pthread_self**() is not the same thing as the kernel thread ID returned by a call to *gettid*(2).

SEE ALSO

pthread_create(3), *pthread_equal*(3), *pthreads*(7)

NAME

pthread_setaffinity_np, pthread_getaffinity_np − set/get CPU affinity of a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize,
                           const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize,
                           cpu_set_t *cpuset);
```

DESCRIPTION

The **pthread_setaffinity_np**() function sets the CPU affinity mask of the thread *thread* to the CPU set pointed to by *cpuset*. If the call is successful, and the thread is not currently running on one of the CPUs in *cpuset*, then it is migrated to one of those CPUs.

The **pthread_getaffinity_np**() function returns the CPU affinity mask of the thread *thread* in the buffer pointed to by *cpuset*.

For more details on CPU affinity masks, see *sched_setaffinity*(2). For a description of a set of macros that can be used to manipulate and inspect CPU sets, see *CPU_SET*(3).

The argument *cpusetsize* is the length (in bytes) of the buffer pointed to by *cpuset*. Typically, this argument would be specified as *sizeof(cpu_set_t)*. (It may be some other value, if using the macros described in *CPU_SET*(3) for dynamically allocating a CPU set.)

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

EFAULT
: A supplied memory address was invalid.

EINVAL
: (**pthread_setaffinity_np**()) The affinity bit mask *mask* contains no processors that are currently physically on the system and permitted to the thread according to any restrictions that may be imposed by the "cpuset" mechanism described in *cpuset*(7).

EINVAL
: (**pthread_setaffinity_np**()) *cpuset* specified a CPU that was outside the set supported by the kernel. (The kernel configuration option **CONFIG_NR_CPUS** defines the range of the set supported by the kernel data type used to represent CPU sets.)

EINVAL
: (**pthread_getaffinity_np**()) *cpusetsize* is smaller than the size of the affinity mask used by the kernel.

ESRCH
: No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setaffinity_np(), **pthread_getaffinity_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.3.4.

In glibc 2.3.3 only, versions of these functions were provided that did not have a *cpusetsize* argument. Instead the CPU set size given to the underlying system calls was always *sizeof(cpu_set_t)*.

NOTES

After a call to **pthread_setaffinity_np**(), the set of CPUs on which the thread will actually run is the intersection of the set specified in the *cpuset* argument and the set of CPUs actually present on the system. The system may further restrict the set of CPUs on which the thread runs if the "cpuset" mechanism described in *cpuset*(7) is being used. These restrictions on the actual set of CPUs on which the thread will run are silently imposed by the kernel.

These functions are implemented on top of the *sched_setaffinity*(2) and *sched_getaffinity*(2) system calls.

A new thread created by *pthread_create*(3) inherits a copy of its creator's CPU affinity mask.

EXAMPLES

In the following program, the main thread uses **pthread_setaffinity_np**() to set its CPU affinity mask to include CPUs 0 to 7 (which may not all be available on the system), and then calls **pthread_getaffinity_np**() to check the resulting CPU affinity mask of the thread.

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    int s;
    cpu_set_t cpuset;
    pthread_t thread;

    thread = pthread_self();

    /* Set affinity mask to include CPUs 0 to 7. */

    CPU_ZERO(&cpuset);
    for (size_t j = 0; j < 8; j++)
        CPU_SET(j, &cpuset);

    s = pthread_setaffinity_np(thread, sizeof(cpuset), &cpuset);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_setaffinity_np");

    /* Check the actual affinity mask assigned to the thread. */

    s = pthread_getaffinity_np(thread, sizeof(cpuset), &cpuset);
    if (s != 0)
        errc(EXIT_FAILURE, s, "pthread_getaffinity_np");

    printf("Set returned by pthread_getaffinity_np() contained:\n");
    for (size_t j = 0; j < CPU_SETSIZE; j++)
        if (CPU_ISSET(j, &cpuset))
            printf("    CPU %zu\n", j);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

sched_setaffinity(2), *CPU_SET*(3), *pthread_attr_setaffinity_np*(3), *pthread_self*(3), *sched_getcpu*(3), *cpuset*(7), *pthreads*(7), *sched*(7)

NAME

pthread_setcancelstate, pthread_setcanceltype − set cancelability state and type

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_setcancelstate(int *state***, int ****oldstate***);**
int pthread_setcanceltype(int *type***, int ****oldtype***);**

DESCRIPTION

The **pthread_setcancelstate**() sets the cancelability state of the calling thread to the value given in *state*. The previous cancelability state of the thread is returned in the buffer pointed to by *oldstate*. The *state* argument must have one of the following values:

PTHREAD_CANCEL_ENABLE
: The thread is cancelable. This is the default cancelability state in all new threads, including the initial thread. The thread's cancelability type determines when a cancelable thread will respond to a cancelation request.

PTHREAD_CANCEL_DISABLE
: The thread is not cancelable. If a cancelation request is received, it is blocked until cancelability is enabled.

The **pthread_setcanceltype**() sets the cancelability type of the calling thread to the value given in *type*. The previous cancelability type of the thread is returned in the buffer pointed to by *oldtype*. The *type* argument must have one of the following values:

PTHREAD_CANCEL_DEFERRED
: A cancelation request is deferred until the thread next calls a function that is a cancelation point (see *pthreads*(7)). This is the default cancelability type in all new threads, including the initial thread.

  Even with deferred cancelation, a cancelation point in an asynchronous signal handler may still be acted upon and the effect is as if it was an asynchronous cancelation.

PTHREAD_CANCEL_ASYNCHRONOUS
: The thread can be canceled at any time. (Typically, it will be canceled immediately upon receiving a cancelation request, but the system doesn't guarantee this.)

The set-and-get operation performed by each of these functions is atomic with respect to other threads in the process calling the same function.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

The **pthread_setcancelstate**() can fail with the following error:

EINVAL
: Invalid value for *state*.

The **pthread_setcanceltype**() can fail with the following error:

EINVAL
: Invalid value for *type*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setcancelstate(), **pthread_setcanceltype**()	Thread safety	MT-Safe
pthread_setcancelstate(), **pthread_setcanceltype**()	Async-cancel safety	AC-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0 POSIX.1-2001.

NOTES

For details of what happens when a thread is canceled, see **pthread_cancel**(3).

Briefly disabling cancelability is useful if a thread performs some critical action that must not be interrupted by a cancelation request. Beware of disabling cancelability for long periods, or around operations that may block for long periods, since that will render the thread unresponsive to cancelation requests.

Asynchronous cancelability

Setting the cancelability type to **PTHREAD_CANCEL_ASYNCHRONOUS** is rarely useful. Since the thread could be canceled at *any* time, it cannot safely reserve resources (e.g., allocating memory with *malloc*(3)), acquire mutexes, semaphores, or locks, and so on. Reserving resources is unsafe because the application has no way of knowing what the state of these resources is when the thread is canceled; that is, did cancelation occur before the resources were reserved, while they were reserved, or after they were released? Furthermore, some internal data structures (e.g., the linked list of free blocks managed by the *malloc*(3) family of functions) may be left in an inconsistent state if cancelation occurs in the middle of the function call. Consequently, clean-up handlers cease to be useful.

Functions that can be safely asynchronously canceled are called *async-cancel-safe functions*. POSIX.1-2001 and POSIX.1-2008 require only that *pthread_cancel*(3), **pthread_setcancelstate**(), and **pthread_setcanceltype**() be async-cancel-safe. In general, other library functions can't be safely called from an asynchronously cancelable thread.

One of the few circumstances in which asynchronous cancelability is useful is for cancelation of a thread that is in a pure compute-bound loop.

Portability notes

The Linux threading implementations permit the *oldstate* argument of **pthread_setcancelstate**() to be NULL, in which case the information about the previous cancelability state is not returned to the caller. Many other implementations also permit a NULL *oldstat* argument, but POSIX.1 does not specify this point, so portable applications should always specify a non-NULL value in *oldstate*. A precisely analogous set of statements applies for the *oldtype* argument of **pthread_setcanceltype**().

EXAMPLES

See *pthread_cancel*(3).

SEE ALSO

pthread_cancel(3), *pthread_cleanup_push*(3), *pthread_testcancel*(3), *pthreads*(7)

NAME

pthread_setconcurrency, pthread_getconcurrency − set/get the concurrency level

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_setconcurrency(int *new_level***);**
int pthread_getconcurrency(*void***);**

DESCRIPTION

The **pthread_setconcurrency**() function informs the implementation of the application's desired concurrency level, specified in *new_level*. The implementation takes this only as a hint: POSIX.1 does not specify the level of concurrency that should be provided as a result of calling **pthread_setconcurrency**().

Specifying *new_level* as 0 instructs the implementation to manage the concurrency level as it deems appropriate.

pthread_getconcurrency() returns the current value of the concurrency level for this process.

RETURN VALUE

On success, **pthread_setconcurrency**() returns 0; on error, it returns a nonzero error number.

pthread_getconcurrency() always succeeds, returning the concurrency level set by a previous call to **pthread_setconcurrency**(), or 0, if **pthread_setconcurrency**() has not previously been called.

ERRORS

pthread_setconcurrency() can fail with the following error:

EINVAL
new_level is negative.

POSIX.1 also documents an **EAGAIN** error ("the value specified by *new_level* would cause a system resource to be exceeded").

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setconcurrency(), **pthread_getconcurrency**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

NOTES

The default concurrency level is 0.

Concurrency levels are meaningful only for M:N threading implementations, where at any moment a subset of a process's set of user-level threads may be bound to a smaller number of kernel-scheduling entities. Setting the concurrency level allows the application to give the system a hint as to the number of kernel-scheduling entities that should be provided for efficient execution of the application.

Both LinuxThreads and NPTL are 1:1 threading implementations, so setting the concurrency level has no meaning. In other words, on Linux these functions merely exist for compatibility with other systems, and they have no effect on the execution of a program.

SEE ALSO

pthread_attr_setscope(3), *pthreads*(7)

NAME

pthread_setname_np, pthread_getname_np − set/get the name of a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_setname_np(pthread_t thread, const char *name);
int pthread_getname_np(pthread_t thread, char name[.size], size_t size);
```

DESCRIPTION

By default, all the threads created using **pthread_create**() inherit the program name. The **pthread_setname_np**() function can be used to set a unique name for a thread, which can be useful for debugging multithreaded applications. The thread name is a meaningful C language string, whose length is restricted to 16 characters, including the terminating null byte ('\0'). The *thread* argument specifies the thread whose name is to be changed; *name* specifies the new name.

The **pthread_getname_np**() function can be used to retrieve the name of the thread. The *thread* argument specifies the thread whose name is to be retrieved. The buffer *name* is used to return the thread name; *size* specifies the number of bytes available in *name*. The buffer specified by *name* should be at least 16 characters in length. The returned thread name in the output buffer will be null terminated.

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number.

ERRORS

The **pthread_setname_np**() function can fail with the following error:

ERANGE
: The length of the string specified pointed to by *name* exceeds the allowed limit.

The **pthread_getname_np**() function can fail with the following error:

ERANGE
: The buffer specified by *name* and *size* is too small to hold the thread name.

If either of these functions fails to open */proc/self/task/*tid*/comm*, then the call may fail with one of the errors described in *open*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setname_np(), **pthread_getname_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.12.

NOTES

pthread_setname_np() internally writes to the thread-specific *comm* file under the */proc* filesystem: */proc/self/task/*tid*/comm*. **pthread_getname_np**() retrieves it from the same location.

EXAMPLES

The program below demonstrates the use of **pthread_setname_np**() and **pthread_getname_np**().

The following shell session shows a sample run of the program:

```
$ ./a.out
Created a thread. Default name is: a.out
The thread name after setting it is THREADFOO.
^Z                              # Suspend the program
[1]+  Stopped                 ./a.out
$ ps H -C a.out -o 'pid tid cmd comm'
  PID   TID CMD                         COMMAND
```

```
 5990  5990 ./a.out                    a.out
 5990  5991 ./a.out                    THREADFOO
$ cat /proc/5990/task/5990/comm
a.out
$ cat /proc/5990/task/5991/comm
THREADFOO
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define NAMELEN 16

static void *
threadfunc(void *parm)
{
    sleep(5);          // allow main program to set the thread name
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thread;
    int rc;
    char thread_name[NAMELEN];

    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    if (rc != 0)
        errc(EXIT_FAILURE, rc, "pthread_create");

    rc = pthread_getname_np(thread, thread_name, NAMELEN);
    if (rc != 0)
        errc(EXIT_FAILURE, rc, "pthread_getname_np");

    printf("Created a thread. Default name is: %s\n", thread_name);
    rc = pthread_setname_np(thread, (argc > 1) ? argv[1] : "THREADFOO");
    if (rc != 0)
        errc(EXIT_FAILURE, rc, "pthread_setname_np");

    sleep(2);

    rc = pthread_getname_np(thread, thread_name, NAMELEN);
    if (rc != 0)
        errc(EXIT_FAILURE, rc, "pthread_getname_np");
    printf("The thread name after setting it is %s.\n", thread_name);

    rc = pthread_join(thread, NULL);
    if (rc != 0)
        errc(EXIT_FAILURE, rc, "pthread_join");

    printf("Done\n");
```

```
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

prctl(2), *pthread_create*(3), *pthreads*(7)

NAME

pthread_setschedparam, pthread_getschedparam − set/get scheduling policy and parameters of a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *restrict policy,
                          struct sched_param *restrict param);
```

DESCRIPTION

The **pthread_setschedparam**() function sets the scheduling policy and parameters of the thread *thread*.

policy specifies the new scheduling policy for *thread*. The supported values for *policy*, and their semantics, are described in *sched*(7).

The structure pointed to by *param* specifies the new scheduling parameters for *thread*. Scheduling parameters are maintained in the following structure:

```
struct sched_param {
    int sched_priority;     /* Scheduling priority */
};
```

As can be seen, only one scheduling parameter is supported. For details of the permitted ranges for scheduling priorities in each scheduling policy, see *sched*(7).

The **pthread_getschedparam**() function returns the scheduling policy and parameters of the thread *thread*, in the buffers pointed to by *policy* and *param*, respectively. The returned priority value is that set by the most recent **pthread_setschedparam**(), *pthread_setschedprio*(3), or *pthread_create*(3) call that affected *thread*. The returned priority does not reflect any temporary priority adjustments as a result of calls to any priority inheritance or priority ceiling functions (see, for example, *pthread_mutexattr_setprioceiling*(3) and *pthread_mutexattr_setprotocol*(3)).

RETURN VALUE

On success, these functions return 0; on error, they return a nonzero error number. If **pthread_setschedparam**() fails, the scheduling policy and parameters of *thread* are not changed.

ERRORS

Both of these functions can fail with the following error:

ESRCH
No thread with the ID *thread* could be found.

pthread_setschedparam() may additionally fail with the following errors:

EINVAL
policy is not a recognized policy, or *param* does not make sense for the *policy*.

EPERM
The caller does not have appropriate privileges to set the specified scheduling policy and parameters.

POSIX.1 also documents an **ENOTSUP** ("attempt was made to set the policy or scheduling parameters to an unsupported value") error for **pthread_setschedparam**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setschedparam(), **pthread_getschedparam**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0 POSIX.1-2001.

NOTES

For a description of the permissions required to, and the effect of, changing a thread's scheduling policy and priority, and details of the permitted ranges for priorities in each scheduling policy, see *sched*(7).

EXAMPLES

The program below demonstrates the use of **pthread_setschedparam**() and **pthread_getschedparam**(), as well as the use of a number of other scheduling-related pthreads functions.

In the following run, the main thread sets its scheduling policy to **SCHED_FIFO** with a priority of 10, and initializes a thread attributes object with a scheduling policy attribute of **SCHED_RR** and a scheduling priority attribute of 20. The program then sets (using *pthread_attr_setinheritsched*(3)) the inherit scheduler attribute of the thread attributes object to **PTHREAD_EXPLICIT_SCHED**, meaning that threads created using this attributes object should take their scheduling attributes from the thread attributes object. The program then creates a thread using the thread attributes object, and that thread displays its scheduling policy and priority.

```
$ su      # Need privilege to set real-time scheduling policies
Password:
# ./a.out -mf10 -ar20 -i e
Scheduler settings of main thread
    policy=SCHED_FIFO, priority=10

Scheduler settings in 'attr'
    policy=SCHED_RR, priority=20
    inheritsched is EXPLICIT

Scheduler attributes of new thread
    policy=SCHED_RR, priority=20
```

In the above output, one can see that the scheduling policy and priority were taken from the values specified in the thread attributes object.

The next run is the same as the previous, except that the inherit scheduler attribute is set to **PTHREAD_INHERIT_SCHED**, meaning that threads created using the thread attributes object should ignore the scheduling attributes specified in the attributes object and instead take their scheduling attributes from the creating thread.

```
# ./a.out -mf10 -ar20 -i i
Scheduler settings of main thread
    policy=SCHED_FIFO, priority=10

Scheduler settings in 'attr'
    policy=SCHED_RR, priority=20
    inheritsched is INHERIT

Scheduler attributes of new thread
    policy=SCHED_FIFO, priority=10
```

In the above output, one can see that the scheduling policy and priority were taken from the creating thread, rather than the thread attributes object.

Note that if we had omitted the *-i i* option, the output would have been the same, since **PTHREAD_INHERIT_SCHED** is the default for the inherit scheduler attribute.

Program source

```
/* pthreads_sched_test.c */

#include <errno.h>
#include <pthread.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

[[noreturn]]
static void
usage(char *prog_name, char *msg)
{
    if (msg != NULL)
        fputs(msg, stderr);

    fprintf(stderr, "Usage: %s [options]\n", prog_name);
    fprintf(stderr, "Options are:\n");
#define fpe(msg) fprintf(stderr, "\t%s", msg)          /* Shorter */
    fpe("-a<policy><prio> Set scheduling policy and priority in\n");
    fpe("                 thread attributes object\n");
    fpe("                 <policy> can be\n");
    fpe("                     f  SCHED_FIFO\n");
    fpe("                     r  SCHED_RR\n");
    fpe("                     o  SCHED_OTHER\n");
    fpe("-A               Use default thread attributes object\n");
    fpe("-i {e|i}         Set inherit scheduler attribute to\n");
    fpe("                 'explicit' or 'inherit'\n");
    fpe("-m<policy><prio> Set scheduling policy and priority on\n");
    fpe("                 main thread before pthread_create() call\n");
    exit(EXIT_FAILURE);
}

static int
get_policy(char p, int *policy)
{
    switch (p) {
    case 'f': *policy = SCHED_FIFO;     return 1;
    case 'r': *policy = SCHED_RR;       return 1;
    case 'o': *policy = SCHED_OTHER;    return 1;
    default:  return 0;
    }
}

static void
display_sched_attr(int policy, const struct sched_param *param)
{
    printf("    policy=%s, priority=%d\n",
           (policy == SCHED_FIFO)  ? "SCHED_FIFO" :
           (policy == SCHED_RR)    ? "SCHED_RR" :
           (policy == SCHED_OTHER) ? "SCHED_OTHER" :
           "???",
           param->sched_priority);
}

static void
display_thread_sched_attr(char *msg)
{
    int policy, s;
    struct sched_param param;

    s = pthread_getschedparam(pthread_self(), &policy, &param);
```

```
        if (s != 0)
            handle_error_en(s, "pthread_getschedparam");

        printf("%s\n", msg);
        display_sched_attr(policy, &param);
    }

    static void *
    thread_start(void *arg)
    {
        display_thread_sched_attr("Scheduler attributes of new thread");

        return NULL;
    }

    int
    main(int argc, char *argv[])
    {
        int s, opt, inheritsched, use_null_attrib, policy;
        pthread_t thread;
        pthread_attr_t attr;
        pthread_attr_t *attrp;
        char *attr_sched_str, *main_sched_str, *inheritsched_str;
        struct sched_param param;

        /* Process command-line options. */

        use_null_attrib = 0;
        attr_sched_str = NULL;
        main_sched_str = NULL;
        inheritsched_str = NULL;

        while ((opt = getopt(argc, argv, "a:Ai:m:")) != -1) {
            switch (opt) {
            case 'a': attr_sched_str = optarg;      break;
            case 'A': use_null_attrib = 1;          break;
            case 'i': inheritsched_str = optarg;    break;
            case 'm': main_sched_str = optarg;      break;
            default:  usage(argv[0], "Unrecognized option\n");
            }
        }

        if (use_null_attrib
            && (inheritsched_str != NULL || attr_sched_str != NULL))
        {
            usage(argv[0], "Can't specify -A with -i or -a\n");
        }

        /* Optionally set scheduling attributes of main thread,
           and display the attributes. */

        if (main_sched_str != NULL) {
            if (!get_policy(main_sched_str[0], &policy))
                usage(argv[0], "Bad policy for main thread (-m)\n");
            param.sched_priority = strtol(&main_sched_str[1], NULL, 0);

            s = pthread_setschedparam(pthread_self(), policy, &param);
            if (s != 0)
                handle_error_en(s, "pthread_setschedparam");
```

```
        }

        display_thread_sched_attr("Scheduler settings of main thread");
        printf("\n");

        /* Initialize thread attributes object according to options. */

        attrp = NULL;

        if (!use_null_attrib) {
            s = pthread_attr_init(&attr);
            if (s != 0)
                handle_error_en(s, "pthread_attr_init");
            attrp = &attr;
        }

        if (inheritsched_str != NULL) {
            if (inheritsched_str[0] == 'e')
                inheritsched = PTHREAD_EXPLICIT_SCHED;
            else if (inheritsched_str[0] == 'i')
                inheritsched = PTHREAD_INHERIT_SCHED;
            else
                usage(argv[0], "Value for −i must be 'e' or 'i'\n");

            s = pthread_attr_setinheritsched(&attr, inheritsched);
            if (s != 0)
                handle_error_en(s, "pthread_attr_setinheritsched");
        }

        if (attr_sched_str != NULL) {
            if (!get_policy(attr_sched_str[0], &policy))
                usage(argv[0], "Bad policy for 'attr' (−a)\n");
            param.sched_priority = strtol(&attr_sched_str[1], NULL, 0);

            s = pthread_attr_setschedpolicy(&attr, policy);
            if (s != 0)
                handle_error_en(s, "pthread_attr_setschedpolicy");
            s = pthread_attr_setschedparam(&attr, &param);
            if (s != 0)
                handle_error_en(s, "pthread_attr_setschedparam");
        }

        /* If we initialized a thread attributes object, display
           the scheduling attributes that were set in the object. */

        if (attrp != NULL) {
            s = pthread_attr_getschedparam(&attr, &param);
            if (s != 0)
                handle_error_en(s, "pthread_attr_getschedparam");
            s = pthread_attr_getschedpolicy(&attr, &policy);
            if (s != 0)
                handle_error_en(s, "pthread_attr_getschedpolicy");

            printf("Scheduler settings in 'attr'\n");
            display_sched_attr(policy, &param);

            pthread_attr_getinheritsched(&attr, &inheritsched);
            printf("    inheritsched is %s\n",
                   (inheritsched == PTHREAD_INHERIT_SCHED)  ? "INHERIT" :
```

```
                (inheritsched == PTHREAD_EXPLICIT_SCHED) ? "EXPLICIT" :
                "???");
        printf("\n");
    }

    /* Create a thread that will display its scheduling attributes. */

    s = pthread_create(&thread, attrp, &thread_start, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    /* Destroy unneeded thread attributes object. */

    if (!use_null_attrib) {
      s = pthread_attr_destroy(&attr);
      if (s != 0)
          handle_error_en(s, "pthread_attr_destroy");
    }

    s = pthread_join(thread, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_join");

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getrlimit(2), *sched_get_priority_min*(2), *pthread_attr_init*(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedparam*(3), *pthread_attr_setschedpolicy*(3), *pthread_create*(3), *pthread_self*(3), *pthread_setschedprio*(3), *pthreads*(7), *sched*(7)

NAME

pthread_setschedprio − set scheduling priority of a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_setschedprio(pthread_t *thread***, int** *prio***);**

DESCRIPTION

The **pthread_setschedprio**() function sets the scheduling priority of the thread *thread* to the value specified in *prio*. (By contrast *pthread_setschedparam*(3) changes both the scheduling policy and priority of a thread.)

RETURN VALUE

On success, this function returns 0; on error, it returns a nonzero error number. If **pthread_setschedprio**() fails, the scheduling priority of *thread* is not changed.

ERRORS

EINVAL

prio is not valid for the scheduling policy of the specified thread.

EPERM

The caller does not have appropriate privileges to set the specified priority.

ESRCH

No thread with the ID *thread* could be found.

POSIX.1 also documents an **ENOTSUP** ("attempt was made to set the priority to an unsupported value") error for *pthread_setschedparam*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_setschedprio()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.3.4. POSIX.1-2001.

NOTES

For a description of the permissions required to, and the effect of, changing a thread's scheduling priority, and details of the permitted ranges for priorities in each scheduling policy, see *sched*(7).

SEE ALSO

getrlimit(2), *sched_get_priority_min*(2), *pthread_attr_init*(3), *pthread_attr_setinheritsched*(3), *pthread_attr_setschedparam*(3), *pthread_attr_setschedpolicy*(3), *pthread_create*(3), *pthread_self*(3), *pthread_setschedparam*(3), *pthreads*(7), *sched*(7)

NAME

pthread_sigmask − examine and change mask of blocked signals

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <signal.h>

int pthread_sigmask(int *how***, const sigset_t ****set***, sigset_t ****oldset***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_sigmask():
_POSIX_C_SOURCE >= 199506L || _XOPEN_SOURCE >= 500

DESCRIPTION

The **pthread_sigmask**() function is just like *sigprocmask*(2), with the difference that its use in multi-threaded programs is explicitly specified by POSIX.1. Other differences are noted in this page.

For a description of the arguments and operation of this function, see *sigprocmask*(2).

RETURN VALUE

On success, **pthread_sigmask**() returns 0; on error, it returns an error number.

ERRORS

See *sigprocmask*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_sigmask()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

A new thread inherits a copy of its creator's signal mask.

The glibc **pthread_sigmask**() function silently ignores attempts to block the two real-time signals that are used internally by the NPTL threading implementation. See *nptl*(7) for details.

EXAMPLES

The program below blocks some signals in the main thread, and then creates a dedicated thread to fetch those signals via *sigwait*(3). The following shell session demonstrates its use:

```
$ ./a.out &
[1] 5423
$ kill -QUIT %1
Signal handling thread got signal 3
$ kill -USR1 %1
Signal handling thread got signal 10
$ kill -TERM %1
[1]+  Terminated              ./a.out
```

Program source

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* Simple error handling functions */
```

```
#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void *
sig_thread(void *arg)
{
    sigset_t *set = arg;
    int s, sig;

    for (;;) {
        s = sigwait(set, &sig);
        if (s != 0)
            handle_error_en(s, "sigwait");
        printf("Signal handling thread got signal %d\n", sig);
    }
}

int
main(void)
{
    pthread_t thread;
    sigset_t set;
    int s;

    /* Block SIGQUIT and SIGUSR1; other threads created by main()
       will inherit a copy of the signal mask. */

    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigaddset(&set, SIGUSR1);
    s = pthread_sigmask(SIG_BLOCK, &set, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_sigmask");

    s = pthread_create(&thread, NULL, &sig_thread, &set);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    /* Main thread carries on to create other threads and/or do
       other work. */

    pause();            /* Dummy pause so we can test program */
}
```

SEE ALSO

sigaction(2), *sigpending*(2), *sigprocmask*(2), *pthread_attr_setsigmask_np*(3), *pthread_create*(3), *pthread_kill*(3), *sigsetops*(3), *pthreads*(7), *signal*(7)

NAME

pthread_sigqueue − queue a signal and data to a thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <signal.h>
#include <pthread.h>

int pthread_sigqueue(pthread_t thread, int sig,
                     const union sigval value);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_sigqueue():
_GNU_SOURCE

DESCRIPTION

The **pthread_sigqueue**() function performs a similar task to *sigqueue*(3), but, rather than sending a signal to a process, it sends a signal to a thread in the same process as the calling thread.

The *thread* argument is the ID of a thread in the same process as the caller. The *sig* argument specifies the signal to be sent. The *value* argument specifies data to accompany the signal; see *sigqueue*(3) for details.

RETURN VALUE

On success, **pthread_sigqueue**() returns 0; on error, it returns an error number.

ERRORS

EAGAIN
The limit of signals which may be queued has been reached. (See *signal*(7) for further information.)

EINVAL
sig was invalid.

ENOSYS
pthread_sigqueue() is not supported on this system.

ESRCH
thread is not valid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_sigqueue()	Thread safety	MT-Safe

VERSIONS

The glibc implementation of **pthread_sigqueue**() gives an error (**EINVAL**) on attempts to send either of the real-time signals used internally by the NPTL threading implementation. See *nptl*(7) for details.

STANDARDS

GNU.

HISTORY

glibc 2.11.

SEE ALSO

rt_tgsigqueueinfo(2), *sigaction*(2), *pthread_sigmask*(3), *sigqueue*(3), *sigwait*(3), *pthreads*(7), *signal*(7)

NAME

pthread_spin_init, pthread_spin_destroy − initialize or destroy a spin lock

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_spin_init(), **pthread_spin_destroy**():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

General note: Most programs should use mutexes instead of spin locks. Spin locks are primarily useful in conjunction with real-time scheduling policies. See NOTES.

The **pthread_spin_init**() function allocates any resources required for the use of the spin lock referred to by *lock* and initializes the lock to be in the unlocked state. The *pshared* argument must have one of the following values:

PTHREAD_PROCESS_PRIVATE
: The spin lock is to be operated on only by threads in the same process as the thread that calls **pthread_spin_init**(). (Attempting to share the spin lock between processes results in undefined behavior.)

PTHREAD_PROCESS_SHARED
: The spin lock may be operated on by any thread in any process that has access to the memory containing the lock (i.e., the lock may be in a shared memory object that is shared among multiple processes).

Calling **pthread_spin_init**() on a spin lock that has already been initialized results in undefined behavior.

The **pthread_spin_destroy**() function destroys a previously initialized spin lock, freeing any resources that were allocated for that lock. Destroying a spin lock that has not been previously been initialized or destroying a spin lock while another thread holds the lock results in undefined behavior.

Once a spin lock has been destroyed, performing any operation on the lock other than once more initializing it with **pthread_spin_init**() results in undefined behavior.

The result of performing operations such as *pthread_spin_lock*(3), *pthread_spin_unlock*(3), and **pthread_spin_destroy**() on *copies* of the object referred to by *lock* is undefined.

RETURN VALUE

On success, there functions return zero. On failure, they return an error number. In the event that **pthread_spin_init**() fails, the lock is not initialized.

ERRORS

pthread_spin_init() may fail with the following errors:

EAGAIN
: The system has insufficient resources to initialize a new spin lock.

ENOMEM
: Insufficient memory to initialize the spin lock.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

Support for process-shared spin locks is a POSIX option. The option is supported in the glibc implementation.

NOTES

Spin locks should be employed in conjunction with real-time scheduling policies (**SCHED_FIFO**, or possibly **SCHED_RR**). Use of spin locks with nondeterministic scheduling policies such as **SCHED_OTHER** probably indicates a design mistake. The problem is that if a thread operating under such a policy is scheduled off the CPU while it holds a spin lock, then other threads will waste time spinning on the lock until the lock holder is once more rescheduled and releases the lock.

If threads create a deadlock situation while employing spin locks, those threads will spin forever consuming CPU time.

User-space spin locks are *not* applicable as a general locking solution. They are, by definition, prone to priority inversion and unbounded spin times. A programmer using spin locks must be exceptionally careful not only in the code, but also in terms of system configuration, thread placement, and priority assignment.

SEE ALSO

pthread_mutex_init(3), *pthread_mutex_lock*(3), *pthread_spin_lock*(3), *pthread_spin_unlock*(3), *pthreads*(7)

NAME

pthread_spin_lock, pthread_spin_trylock, pthread_spin_unlock − lock and unlock a spin lock

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t ****lock*);**
int pthread_spin_trylock(pthread_spinlock_t ****lock*);**
int pthread_spin_unlock(pthread_spinlock_t ****lock*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

pthread_spin_lock(), **pthread_spin_trylock**():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **pthread_spin_lock**() function locks the spin lock referred to by *lock*. If the spin lock is currently unlocked, the calling thread acquires the lock immediately. If the spin lock is currently locked by another thread, the calling thread spins, testing the lock until it becomes available, at which point the calling thread acquires the lock.

Calling **pthread_spin_lock**() on a lock that is already held by the caller or a lock that has not been initialized with *pthread_spin_init*(3) results in undefined behavior.

The **pthread_spin_trylock**() function is like **pthread_spin_lock**(), except that if the spin lock referred to by *lock* is currently locked, then, instead of spinning, the call returns immediately with the error **EBUSY**.

The **pthread_spin_unlock**() function unlocks the spin lock referred to *lock*. If any threads are spinning on the lock, one of those threads will then acquire the lock.

Calling **pthread_spin_unlock**() on a lock that is not held by the caller results in undefined behavior.

RETURN VALUE

On success, these functions return zero. On failure, they return an error number.

ERRORS

pthread_spin_lock() may fail with the following errors:

EDEADLOCK
The system detected a deadlock condition.

pthread_spin_trylock() fails with the following errors:

EBUSY
The spin lock is currently locked by another thread.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

CAVEATS

Applying any of the functions described on this page to an uninitialized spin lock results in undefined behavior.

Carefully read NOTES in *pthread_spin_init*(3).

SEE ALSO

pthread_spin_destroy(3), *pthread_spin_init*(3), *pthreads*(7)

NAME

pthread_testcancel − request delivery of any pending cancelation request

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <pthread.h>

void pthread_testcancel(void);

DESCRIPTION

Calling **pthread_testcancel**() creates a cancelation point within the calling thread, so that a thread that is otherwise executing code that contains no cancelation points will respond to a cancelation request.

If cancelability is disabled (using *pthread_setcancelstate*(3)), or no cancelation request is pending, then a call to **pthread_testcancel**() has no effect.

RETURN VALUE

This function does not return a value. If the calling thread is canceled as a consequence of a call to this function, then the function does not return.

ERRORS

This function always succeeds.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_testcancel()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.0. POSIX.1-2001.

EXAMPLES

See *pthread_cleanup_push*(3).

SEE ALSO

pthread_cancel(3), *pthread_cleanup_push*(3), *pthread_setcancelstate*(3), *pthreads*(7)

NAME

pthread_tryjoin_np, pthread_timedjoin_np − try to join with a terminated thread

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_tryjoin_np(pthread_t thread, void **retval);
int pthread_timedjoin_np(pthread_t thread, void **retval,
                         const struct timespec *abstime);
```

DESCRIPTION

These functions operate in the same way as *pthread_join*(3), except for the differences described on this page.

The **pthread_tryjoin_np**() function performs a nonblocking join with the thread *thread*, returning the exit status of the thread in **retval*. If *thread* has not yet terminated, then instead of blocking, as is done by *pthread_join*(3), the call returns an error.

The **pthread_timedjoin_np**() function performs a join-with-timeout. If *thread* has not yet terminated, then the call blocks until a maximum time, specified in *abstime*, measured against the **CLOCK_REALTIME** clock. If the timeout expires before *thread* terminates, the call returns an error. The *abstime* argument is a *timespec*(3) structure, specifying an absolute time measured since the Epoch (see *time*(2)).

RETURN VALUE

On success, these functions return 0; on error, they return an error number.

ERRORS

These functions can fail with the same errors as *pthread_join*(3). **pthread_tryjoin_np**() can in addition fail with the following error:

EBUSY
thread had not yet terminated at the time of the call.

pthread_timedjoin_np() can in addition fail with the following errors:

EINVAL
abstime value is invalid (*tv_sec* is less than 0 or *tv_nsec* is greater than 1e9).

ETIMEDOUT
The call timed out before *thread* terminated.

pthread_timedjoin_np() never returns the error **EINTR**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_tryjoin_np(), **pthread_timedjoin_np**()	Thread safety	MT-Safe

STANDARDS

GNU; hence the suffix "_np" (nonportable) in the names.

HISTORY

glibc 2.3.3.

BUGS

The **pthread_timedjoin_np**() function measures time by internally calculating a relative sleep interval that is then measured against the **CLOCK_MONOTONIC** clock instead of the **CLOCK_REALTIME** clock. Consequently, the timeout is unaffected by discontinuous changes to the **CLOCK_REALTIME** clock.

EXAMPLES

The following code waits to join for up to 5 seconds:

```
struct timespec ts;
int s;
```

```
        ...

        if (clock_gettime(CLOCK_REALTIME, &ts) == -1) {
            /* Handle error */
        }

        ts.tv_sec += 5;

        s = pthread_timedjoin_np(thread, NULL, &ts);
        if (s != 0) {
            /* Handle error */
        }
```

SEE ALSO

clock_gettime(2), *pthread_exit*(3), *pthread_join*(3), *timespec*(3), *pthreads*(7)

NAME

pthread_yield − yield the processor

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <pthread.h>

[[deprecated]] int pthread_yield(void);
```

DESCRIPTION

Note: This function is deprecated; see below.

pthread_yield() causes the calling thread to relinquish the CPU. The thread is placed at the end of the run queue for its static priority and another thread is scheduled to run. For further details, see *sched_yield*(2)

RETURN VALUE

On success, **pthread_yield**() returns 0; on error, it returns an error number.

ERRORS

On Linux, this call always succeeds (but portable and future-proof applications should nevertheless handle a possible error return).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
pthread_yield()	Thread safety	MT-Safe

VERSIONS

On Linux, this function is implemented as a call to *sched_yield*(2).

STANDARDS

None.

HISTORY

Deprecated since glibc 2.34. Use the standardized *sched_yield*(2) instead.

NOTES

pthread_yield() is intended for use with real-time scheduling policies (i.e., **SCHED_FIFO** or **SCHED_RR**). Use of **pthread_yield**() with nondeterministic scheduling policies such as **SCHED_OTHER** is unspecified and very likely means your application design is broken.

SEE ALSO

sched_yield(2), *pthreads*(7), *sched*(7)

NAME

ptsname, ptsname_r − get the name of the slave pseudoterminal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

char *ptsname(int fd);
int ptsname_r(int fd, char buf[.buflen], size_t buflen);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ptsname():
  Since glibc 2.24:
    _XOPEN_SOURCE >= 500
  glibc 2.23 and earlier:
    _XOPEN_SOURCE

ptsname_r():
  _GNU_SOURCE

DESCRIPTION

The **ptsname**() function returns the name of the slave pseudoterminal device corresponding to the master referred to by the file descriptor *fd*.

The **ptsname_r**() function is the reentrant equivalent of **ptsname**(). It returns the name of the slave pseudoterminal device as a null-terminated string in the buffer pointed to by *buf*. The *buflen* argument specifies the number of bytes available in *buf*.

RETURN VALUE

On success, **ptsname**() returns a pointer to a string in static storage which will be overwritten by subsequent calls. This pointer must not be freed. On failure, NULL is returned.

On success, **ptsname_r**() returns 0. On failure, an error number is returned to indicate the error.

ERRORS

EINVAL
: (**ptsname_r**() only) *buf* is NULL. (This error is returned only for glibc 2.25 and earlier.)

ENOTTY
: *fd* does not refer to a pseudoterminal master device.

ERANGE
: (**ptsname_r**() only) *buf* is too small.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ptsname()	Thread safety	MT-Unsafe race:ptsname
ptsname_r()	Thread safety	MT-Safe

VERSIONS

A version of **ptsname_r**() is documented on Tru64 and HP-UX, but on those implementations, −1 is returned on error, with *errno* set to indicate the error. Avoid using this function in portable programs.

STANDARDS

ptsname():
: POSIX.1-2008.

ptsname_r() is a Linux extension, that is proposed for inclusion in the next major revision of POSIX.1 (Issue 8).

HISTORY

ptsname():
: POSIX.1-2001. glibc 2.1.

ptsname() is part of the UNIX 98 pseudoterminal support (see *pts*(4)).

SEE ALSO

grantpt(3), *posix_openpt*(3), *ttyname*(3), *unlockpt*(3), *pts*(4), *pty*(7)

NAME

putenv − change or add an environment variable

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int putenv(char *string);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

putenv():

```
_XOPEN_SOURCE
    || /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE
```

DESCRIPTION

The **putenv**() function adds or changes the value of environment variables. The argument *string* is of the form *name=value*. If *name* does not already exist in the environment, then *string* is added to the environment. If *name* does exist, then the value of *name* in the environment is changed to *value*. The string pointed to by *string* becomes part of the environment, so altering the string changes the environment.

RETURN VALUE

The **putenv**() function returns zero on success. On failure, it returns a nonzero value, and *errno* is set to indicate the error.

ERRORS

ENOMEM
: Insufficient space to allocate new environment.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
putenv()	Thread safety	MT-Unsafe const:env

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr2, 4.3BSD-Reno.

The **putenv**() function is not required to be reentrant, and the one in glibc 2.0 is not, but the glibc 2.1 version is.

Since glibc 2.1.2, the glibc implementation conforms to SUSv2: the pointer *string* given to **putenv**() is used. In particular, this string becomes part of the environment; changing it later will change the environment. (Thus, it is an error to call **putenv**() with an automatic variable as the argument, then return from the calling function while *string* is still part of the environment.) However, from glibc 2.0 to glibc 2.1.1, it differs: a copy of the string is used. On the one hand this causes a memory leak, and on the other hand it violates SUSv2.

The 4.3BSD-Reno version, like glibc 2.0, uses a copy; this is fixed in all modern BSDs.

SUSv2 removes the *const* from the prototype, and so does glibc 2.1.3.

The GNU C library implementation provides a nonstandard extension. If *string* does not include an equal sign:

```
putenv("NAME");
```

then the named variable is removed from the caller's environment.

SEE ALSO

clearenv(3), *getenv*(3), *setenv*(3), *unsetenv*(3), *environ*(7)

NAME

putgrent − write a group database entry to a file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <grp.h>

int putgrent(const struct group *restrict *grp***, FILE *restrict** *stream***);**

DESCRIPTION

The **putgrent**() function is the counterpart for *fgetgrent*(3). The function writes the content of the provided *struct group* into the *stream*. The list of group members must be NULL-terminated or NULL-initialized.

The *struct group* is defined as follows:

```
struct group {
    char   *gr_name;      /* group name */
    char   *gr_passwd;    /* group password */
    gid_t   gr_gid;       /* group ID */
    char  **gr_mem;       /* group members */
};
```

RETURN VALUE

The function returns zero on success, and a nonzero value on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
putgrent()	Thread safety	MT-Safe

STANDARDS

GNU.

SEE ALSO

fgetgrent(3), *getgrent*(3), *group*(5)

NAME

putpwent − write a password file entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>

int putpwent(const struct passwd *restrict p, FILE *restrict stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

putpwent():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

The **putpwent**() function writes a password entry from the structure *p* in the file associated with *stream*.

The *passwd* structure is defined in <*pwd.h*> as follows:

```
struct passwd {
    char    *pw_name;       /* username */
    char    *pw_passwd;     /* user password */
    uid_t    pw_uid;        /* user ID */
    gid_t    pw_gid;        /* group ID */
    char    *pw_gecos;      /* real name */
    char    *pw_dir;        /* home directory */
    char    *pw_shell;      /* shell program */
};
```

RETURN VALUE

The **putpwent**() function returns 0 on success. On failure, it returns −1, and *errno* is set to indicate the error.

ERRORS

EINVAL
Invalid (NULL) argument given.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
putpwent()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

SVr4.

SEE ALSO

endpwent(3), *fgetpwent*(3), *getpw*(3), *getpwent*(3), *getpwnam*(3), *getpwuid*(3), *setpwent*(3)

NAME

fputc, fputs, putc, putchar, puts − output of characters and strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);

int fputs(const char *restrict s, FILE *restrict stream);
int puts(const char *s);
```

DESCRIPTION

fputc() writes the character *c*, cast to an *unsigned char*, to *stream*.

putc() is equivalent to **fputc**() except that it may be implemented as a macro which evaluates *stream* more than once.

putchar(*c***)** is equivalent to **putc(***c***,** *stdout***)**.

fputs() writes the string *s* to *stream*, without its terminating null byte ('\0').

puts() writes the string *s* and a trailing newline to *stdout*.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the *stdio* library for the same output stream.

For nonlocking counterparts, see *unlocked_stdio*(3).

RETURN VALUE

fputc(), **putc**(), and **putchar**() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

puts() and **fputs**() return a nonnegative number on success, or **EOF** on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
fputc(), **fputs**(), **putc**(), **putchar**(), **puts**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, C99.

BUGS

It is not advisable to mix calls to output functions from the *stdio* library with low-level calls to *write*(2) for the file descriptor associated with the same output stream; the results will be undefined and very probably not what you want.

SEE ALSO

write(2), *ferror*(3), *fgets*(3), *fopen*(3), *fputwc*(3), *fputws*(3), *fseek*(3), *fwrite*(3), *putwchar*(3), *scanf*(3), *unlocked_stdio*(3)

NAME

putwchar − write a wide character to standard output

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wint_t putwchar(wchar_t *wc***);**

DESCRIPTION

The **putwchar**() function is the wide-character equivalent of the *putchar*(3) function. It writes the wide character *wc* to *stdout*. If *ferror(stdout)* becomes true, it returns **WEOF**. If a wide character conversion error occurs, it sets *errno* to **EILSEQ** and returns **WEOF**. Otherwise, it returns *wc*.

For a nonlocking counterpart, see *unlocked_stdio*(3).

RETURN VALUE

The **putwchar**() function returns *wc* if no error occurred, or **WEOF** to indicate an error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
putwchar()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **putwchar**() depends on the **LC_CTYPE** category of the current locale.

It is reasonable to expect that **putwchar**() will actually write the multibyte sequence corresponding to the wide character *wc*.

SEE ALSO

fputwc(3), *unlocked_stdio*(3)

NAME

qecvt, qfcvt, qgcvt − convert a floating-point number to a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

[[deprecated]] char *qecvt(long double number, int ndigits,
                           int *restrict decpt, int *restrict sign);
[[deprecated]] char *qfcvt(long double number, int ndigits,
                           int *restrict decpt, int *restrict sign);
[[deprecated]] char *qgcvt(long double number, int ndigit, char *buf);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

qecvt(), **qfcvt**(), **qgcvt**():
Since glibc 2.19:
_DEFAULT_SOURCE
In glibc up to and including 2.19:
_SVID_SOURCE

DESCRIPTION

The functions **qecvt**(), **qfcvt**(), and **qgcvt**() are identical to *ecvt*(3), *fcvt*(3), and *gcvt*(3) respectively, except that they use a *long double* argument *number*. See *ecvt*(3) and *gcvt*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
qecvt()	Thread safety	MT-Unsafe race:qecvt
qfcvt()	Thread safety	MT-Unsafe race:qfcvt
qgcvt()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

SVr4, SunOS, GNU.

These functions are obsolete. Instead, *snprintf*(3) is recommended.

SEE ALSO

ecvt(3), *ecvt_r*(3), *gcvt*(3), *sprintf*(3)

NAME

qsort, qsort_r − sort an array

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

void qsort(void base[.size * .nmemb], size_t nmemb, size_t size,
      int (*compar)(const void [.size], const void [.size]));
void qsort_r(void base[.size * .nmemb], size_t nmemb, size_t size,
      int (*compar)(const void [.size], const void [.size], void *),
      void *arg);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

qsort_r():
_GNU_SOURCE

DESCRIPTION

The **qsort**() function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

The **qsort_r**() function is identical to **qsort**() except that the comparison function *compar* takes a third argument. A pointer is passed to the comparison function via *arg*. In this way, the comparison function does not need to use global variables to pass through arbitrary arguments, and is therefore reentrant and safe to use in threads.

RETURN VALUE

The **qsort**() and **qsort_r**() functions return no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
qsort(), **qsort_r**()	Thread safety	MT-Safe

STANDARDS

qsort() C11, POSIX.1-2008.

HISTORY

qsort() POSIX.1-2001, C89, SVr4, 4.3BSD.

qsort_r()
glibc 2.8.

NOTES

To compare C strings, the comparison function can call *strcmp*(3), as shown in the example below.

EXAMPLES

For one example of use, see the example under *bsearch*(3).

Another example is the following program, which sorts the strings given in its command-line arguments:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int
cmpstringp(const void *p1, const void *p2)
{
```

```
        /* The actual arguments to this function are "pointers to
           pointers to char", but strcmp(3) arguments are "pointers
           to char", hence the following cast plus dereference. */

        return strcmp(*(const char **) p1, *(const char **) p2);
    }

    int
    main(int argc, char *argv[])
    {
        if (argc < 2) {
            fprintf(stderr, "Usage: %s <string>...\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);

        for (size_t j = 1; j < argc; j++)
            puts(argv[j]);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

sort(1), *alphasort*(3), *strcmp*(3), *versionsort*(3)

NAME

raise − send a signal to the caller

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

int raise(int *sig***);**

DESCRIPTION

The **raise**() function sends a signal to the calling process or thread. In a single-threaded program it is equivalent to

```
kill(getpid(), sig);
```

In a multithreaded program it is equivalent to

```
pthread_kill(pthread_self(), sig);
```

If the signal causes a handler to be called, **raise**() will return only after the signal handler has returned.

RETURN VALUE

raise() returns 0 on success, and nonzero for failure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
raise()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

Since glibc 2.3.3, **raise**() is implemented by calling *tgkill*(2), if the kernel supports that system call. Older glibc versions implemented **raise**() using *kill*(2).

SEE ALSO

getpid(2), *kill*(2), *sigaction*(2), *signal*(2), *pthread_kill*(3), *signal*(7)

NAME

rand, rand_r, srand − pseudo-random number generator

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);

[[deprecated]] int rand_r(unsigned int *seedp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

rand_r():
Since glibc 2.24:
_POSIX_C_SOURCE >= 199506L
glibc 2.23 and earlier
_POSIX_C_SOURCE

DESCRIPTION

The **rand**() function returns a pseudo-random integer in the range 0 to **RAND_MAX** inclusive (i.e., the mathematical range [0, **RAND_MAX**]).

The **srand**() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by **rand**(). These sequences are repeatable by calling **srand**() with the same seed value.

If no seed value is provided, the **rand**() function is automatically seeded with a value of 1.

The function **rand**() is not reentrant, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. In order to get reproducible behavior in a threaded application, this state must be made explicit; this can be done using the reentrant function **rand_r**().

Like **rand**(), **rand_r**() returns a pseudo-random integer in the range [0, **RAND_MAX**]. The *seedp* argument is a pointer to an *unsigned int* that is used to store state between calls. If **rand_r**() is called with the same initial value for the integer pointed to by *seedp*, and that value is not modified between calls, then the same pseudo-random sequence will result.

The value pointed to by the *seedp* argument of **rand_r**() provides only a very small amount of state, so this function will be a weak pseudo-random generator. Try *drand48_r*(3) instead.

RETURN VALUE

The **rand**() and **rand_r**() functions return a value between 0 and **RAND_MAX** (inclusive). The **srand**() function returns no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rand(), **rand_r**(), **srand**()	Thread safety	MT-Safe

VERSIONS

The versions of **rand**() and **srand**() in the Linux C Library use the same random number generator as *random*(3) and *srandom*(3), so the lower-order bits should be as random as the higher-order bits. However, on older **rand**() implementations, and on current implementations on different systems, the lower-order bits are much less random than the higher-order bits. Do not use this function in applications intended to be portable when good randomness is needed. (Use *random*(3) instead.)

STANDARDS

rand()
srand()
C11, POSIX.1-2008.

rand_r()
POSIX.1-2008.

HISTORY

rand()
srand()
SVr4, 4.3BSD, C89, POSIX.1-2001.

rand_r()
POSIX.1-2001. Obsolete in POSIX.1-2008.

EXAMPLES

POSIX.1-2001 gives the following example of an implementation of **rand**() and **srand**(), possibly useful when one needs the same sequence on two different machines.

```
static unsigned long next = 1;

/* RAND_MAX assumed to be 32767 */
int myrand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned int seed) {
    next = seed;
}
```

The following program can be used to display the pseudo-random sequence produced by **rand**() when given a particular seed. When the seed is *−1*, the program uses a random seed.

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int           r;
    unsigned int  seed, nloops;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    if (seed == -1) {
        seed = arc4random();
        printf("seed: %u\n", seed);
    }

    srand(seed);
    for (unsigned int j = 0; j < nloops; j++) {
        r =  rand();
        printf("%d\n", r);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

drand48(3), *random*(3)

NAME

random, srandom, initstate, setstate − random number generator

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

long random(void);
void srandom(unsigned int seed);

char *initstate(unsigned int seed, char state[.n], size_t n);
char *setstate(char *state);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

random(), **srandom**(), **initstate**(), **setstate**():

```
_XOPEN_SOURCE >= 500
    || /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

DESCRIPTION

The **random**() function uses a nonlinear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to 2^31 − 1. The period of this random number generator is very large, approximately *16 * ((2^31) − 1)*.

The **srandom**() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by **random**(). These sequences are repeatable by calling **srandom**() with the same seed value. If no seed value is provided, the **random**() function is automatically seeded with a value of 1.

The **initstate**() function allows a state array *state* to be initialized for use by **random**(). The size of the state array *n* is used by **initstate**() to decide how sophisticated a random number generator it should use—the larger the state array, the better the random numbers will be. Current "optimal" values for the size of the state array *n* are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes results in an error. *seed* is the seed for the initialization, which specifies a starting point for the random number sequence, and provides for restarting at the same point.

The **setstate**() function changes the state array used by the **random**() function. The state array *state* is used for random number generation until the next call to **initstate**() or **setstate**(). *state* must first have been initialized using **initstate**() or be the result of a previous call of **setstate**().

RETURN VALUE

The **random**() function returns a value between 0 and *(2^31) − 1*. The **srandom**() function returns no value.

The **initstate**() function returns a pointer to the previous state array. On failure, it returns NULL, and *errno* is set to indicate the error.

On success, **setstate**() returns a pointer to the previous state array. On failure, it returns NULL, and *errno* is set to indicate the error.

ERRORS

EINVAL
: The *state* argument given to **setstate**() was NULL.

EINVAL
: A state array of less than 8 bytes was specified to **initstate**().

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
random(), **srandom**(), **initstate**(), **setstate**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

NOTES

Random-number generation is a complex topic. *Numerical Recipes in C: The Art of Scientific Computing* (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New York: Cambridge University Press, 2007, 3rd ed.) provides an excellent discussion of practical random-number generation issues in Chapter 7 (Random Numbers).

For a more theoretical discussion which also covers many practical issues in depth, see Chapter 3 (Random Numbers) in Donald E. Knuth's *The Art of Computer Programming*, volume 2 (Seminumerical Algorithms), 2nd ed.; Reading, Massachusetts: Addison-Wesley Publishing Company, 1981.

CAVEATS

The **random**() function should not be used in multithreaded programs where reproducible behavior is required. Use *random_r*(3) for that purpose.

BUGS

According to POSIX, **initstate**() should return NULL on error. In the glibc implementation, *errno* is (as specified) set on error, but the function does not return NULL.

SEE ALSO

getrandom(2), *drand48*(3), *rand*(3), *random_r*(3), *srand*(3)

NAME

random_r, srandom_r, initstate_r, setstate_r − reentrant random number generator

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int random_r(struct random_data *restrict buf,
        int32_t *restrict result);
int srandom_r(unsigned int seed, struct random_data *buf);

int initstate_r(unsigned int seed, char statebuf[restrict .statelen],
        size_t statelen, struct random_data *restrict buf);
int setstate_r(char *restrict statebuf,
        struct random_data *restrict buf);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

random_r(), **srandom_r**(), **initstate_r**(), **setstate_r**():

```
/* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

DESCRIPTION

These functions are the reentrant equivalents of the functions described in *random*(3). They are suitable for use in multithreaded programs where each thread needs to obtain an independent, reproducible sequence of random numbers.

The **random_r**() function is like *random*(3), except that instead of using state information maintained in a global variable, it uses the state information in the argument pointed to by *buf*, which must have been previously initialized by **initstate_r**(). The generated random number is returned in the argument *result*.

The **srandom_r**() function is like *srandom*(3), except that it initializes the seed for the random number generator whose state is maintained in the object pointed to by *buf*, which must have been previously initialized by **initstate_r**(), instead of the seed associated with the global state variable.

The **initstate_r**() function is like *initstate*(3) except that it initializes the state in the object pointed to by *buf*, rather than initializing the global state variable. Before calling this function, the *buf.state* field must be initialized to NULL. The **initstate_r**() function records a pointer to the *statebuf* argument inside the structure pointed to by *buf*. Thus, *statebuf* should not be deallocated so long as *buf* is still in use. (So, *statebuf* should typically be allocated as a static variable, or allocated on the heap using *malloc*(3) or similar.)

The **setstate_r**() function is like *setstate*(3) except that it modifies the state in the object pointed to by *buf*, rather than modifying the global state variable. *state* must first have been initialized using **initstate_r**() or be the result of a previous call of **setstate_r**().

RETURN VALUE

All of these functions return 0 on success. On error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EINVAL
: A state array of less than 8 bytes was specified to **initstate_r**().

EINVAL
: The *statebuf* or *buf* argument to **setstate_r**() was NULL.

EINVAL
: The *buf* or *result* argument to **random_r**() was NULL.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
random_r(), **srandom_r**(), **initstate_r**(), **setstate_r**()	Thread safety	MT-Safe race:buf

STANDARDS

GNU.

BUGS

The **initstate_r**() interface is confusing. It appears that the *random_data* type is intended to be opaque, but the implementation requires the user to either initialize the *buf.state* field to NULL or zero out the entire structure before the call.

SEE ALSO

drand48(3), *rand*(3), *random*(3)

NAME

rcmd, rresvport, iruserok, ruserok, rcmd_af, rresvport_af, iruserok_af, ruserok_af − routines for returning a stream to a remote command

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>    /* Or <unistd.h> on some systems */

int rcmd(char **restrict ahost, unsigned short inport,
         const char *restrict locuser,
         const char *restrict remuser,
         const char *restrict cmd, int *restrict  fd2p);

int rresvport(int *port);

int iruserok(uint32_t raddr, int superuser,
         const char *ruser, const char *luser);
int ruserok(const char *rhost, int superuser,
         const char *ruser, const char *luser);

int rcmd_af(char **restrict ahost, unsigned short inport,
         const char *restrict locuser,
         const char *restrict remuser,
         const char *restrict cmd, int *restrict  fd2p,
         sa_family_t af);

int rresvport_af(int *port, sa_family_t af);

int iruserok_af(const void *restrict raddr, int superuser,
         const char *restrict ruser, const char *restrict luser,
         sa_family_t af);
int ruserok_af(const char *rhost, int superuser,
         const char *ruser, const char *luser,
         sa_family_t af);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

rcmd(), **rcmd_af**(), **rresvport**(), **rresvport_af**(), **iruserok**(), **iruserok_af**(), **ruserok**(), **ruserok_af**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **rcmd**() function is used by the superuser to execute a command on a remote machine using an authentication scheme based on privileged port numbers. The **rresvport**() function returns a file descriptor to a socket with an address in the privileged port space. The **iruserok**() and **ruserok**() functions are used by servers to authenticate clients requesting service with **rcmd**(). All four functions are used by the *rshd*(8) server (among others).

rcmd()

The **rcmd**() function looks up the host **ahost* using *gethostbyname*(3), returning −1 if the host does not exist. Otherwise, **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the caller, and given to the remote command as *stdin* and *stdout*. If *fd2p* is nonzero, then an auxiliary channel to a control process will be set up, and a file descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd*(8)

rresvport()
The **rresvport**() function is used to obtain a socket with a privileged port bound to it. This socket is suitable for use by **rcmd**() and several other functions. Privileged ports are those in the range 0 to 1023. Only a privileged process (on Linux, a process that has the **CAP_NET_BIND_SERVICE** capability in the user namespace governing its network namespace) is allowed to bind to a privileged port. In the glibc implementation, this function restricts its search to the ports from 512 to 1023. The *port* argument is value-result: the value it supplies to the call is used as the starting point for a circular search of the port range; on (successful) return, it contains the port number that was bound to.

iruserok() and ruserok()
The **iruserok**() and **ruserok**() functions take a remote host's IP address or name, respectively, two usernames and a flag indicating whether the local user's name is that of the superuser. Then, if the user is *not* the superuser, it checks the */etc/hosts.equiv* file. If that lookup is not done, or is unsuccessful, the *.rhosts* in the local user's home directory is checked to see if the request for service is allowed.

If this file does not exist, is not a regular file, is owned by anyone other than the user or the superuser, is writable by anyone other than the owner, or is hardlinked anywhere, the check automatically fails. Zero is returned if the machine name is listed in the *hosts.equiv* file, or the host and remote username are found in the *.rhosts* file; otherwise **iruserok**() and **ruserok**() return −1. If the local domain (as obtained from *gethostname*(2)) is the same as the remote domain, only the machine name need be specified.

If the IP address of the remote host is known, **iruserok**() should be used in preference to **ruserok**(), as it does not require trusting the DNS server for the remote host's domain.

***_af() variants**
All of the functions described above work with IPv4 (**AF_INET**) sockets. The "_af" variants take an extra argument that allows the socket address family to be specified. For these functions, the *af* argument can be specified as **AF_INET** or **AF_INET6**. In addition, **rcmd_af**() supports the use of **AF_UNSPEC**.

RETURN VALUE
The **rcmd**() function returns a valid socket descriptor on success. It returns −1 on error and prints a diagnostic message on the standard error.

The **rresvport**() function returns a valid, bound socket descriptor on success. On failure, it returns −1 and sets *errno* to indicate the error. The error code **EAGAIN** is overloaded to mean: "All network ports in use".

For information on the return from **ruserok**() and **iruserok**(), see above.

ATTRIBUTES
For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rcmd(), **rcmd_af**()	Thread safety	MT-Unsafe
rresvport(), **rresvport_af**()	Thread safety	MT-Safe
iruserok(), **ruserok**(), **iruserok_af**(), **ruserok_af**()	Thread safety	MT-Safe locale

STANDARDS
BSD.

HISTORY
iruserok_af()
rcmd_af()
rresvport_af()
ruserok_af()
glibc 2.2.

Solaris, 4.2BSD. The "_af" variants are more recent additions, and are not present on as wide a range of systems.

BUGS
iruserok() and **iruserok_af**() are declared in glibc headers only since glibc 2.12.

SEE ALSO

rlogin(1), *rsh*(1), *rexec*(3), *rexecd*(8), *rlogind*(8), *rshd*(8)

NAME

re_comp, re_exec − BSD regex functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _REGEX_RE_COMP
#include <sys/types.h>
#include <regex.h>

[[deprecated]] char *re_comp(const char *regex);
[[deprecated]] int re_exec(const char *string);
```

DESCRIPTION

re_comp() is used to compile the null-terminated regular expression pointed to by *regex*. The compiled pattern occupies a static area, the pattern buffer, which is overwritten by subsequent use of **re_comp**(). If *regex* is NULL, no operation is performed and the pattern buffer's contents are not altered.

re_exec() is used to assess whether the null-terminated string pointed to by *string* matches the previously compiled *regex*.

RETURN VALUE

re_comp() returns NULL on successful compilation of *regex* otherwise it returns a pointer to an appropriate error message.

re_exec() returns 1 for a successful match, zero for failure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
re_comp(), **re_exec**()	Thread safety	MT-Unsafe

STANDARDS

None.

HISTORY

4.3BSD.

These functions are obsolete; the functions documented in *regcomp*(3) should be used instead.

SEE ALSO

regcomp(3), *regex*(7), GNU regex manual

NAME

readdir − read a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR ***dirp*);**

DESCRIPTION

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are *d_name* and *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.

The fields of the *dirent* structure are as follows:

d_ino This is the inode number of the file.

d_off The value returned in *d_off* is the same as would be returned by calling *telldir*(3) at the current position in the directory stream. Be aware that despite its type and name, the *d_off* field is seldom any kind of directory offset on modern filesystems. Applications should treat this field as an opaque value, making no assumptions about its contents; see also *telldir*(3).

d_reclen
This is the size (in bytes) of the returned record. This may not match the size of the structure definition shown above; see NOTES.

d_type This field contains a value indicating the file type, making it possible to avoid the expense of calling *lstat*(2) if further actions depend on the type of the file.

When a suitable feature test macro is defined (**_DEFAULT_SOURCE** since glibc 2.19, or **_BSD_SOURCE** on glibc 2.19 and earlier), glibc defines the following macro constants for the value returned in *d_type*:

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

DT_REG This is a regular file.

DT_SOCK This is a UNIX domain socket.

DT_UNKNOWN
The file type could not be determined.

Currently, only some filesystems (among them: Btrfs, ext2, ext3, and ext4) have full support for returning the file type in *d_type*. All applications must properly handle a return of **DT_UNKNOWN**.

d_name
This field contains the null terminated filename. *See NOTES*.

The data returned by **readdir**() may be overwritten by subsequent calls to **readdir**() for the same directory stream.

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to *free*(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set to indicate the error. To distinguish end of stream from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EBADF
Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
readdir()	Thread safety	MT-Unsafe race:dirstream

In the current POSIX.1 specification (POSIX.1-2008), **readdir**() is not required to be thread-safe. However, in modern implementations (including the glibc implementation), concurrent calls to **readdir**() that specify different directory streams are thread-safe. In cases where multiple threads must read from the same directory stream, using **readdir**() with external synchronization is still preferable to the use of the deprecated *readdir_r*(3) function. It is expected that a future version of POSIX.1 will require that **readdir**() be thread-safe when concurrently employed on different directory streams.

VERSIONS

Only the fields *d_name* and (as an XSI extension) *d_ino* are specified in POSIX.1. Other than Linux, the *d_type* field is available mainly only on BSD systems. The remaining fields are available on many, but not all systems. Under glibc, programs can check for the availability of the fields not defined in POSIX.1 by testing whether the macros **_DIRENT_HAVE_D_NAMLEN**, **_DIRENT_HAVE_D_RECLEN**, **_DIRENT_HAVE_D_OFF**, or **_DIRENT_HAVE_D_TYPE** are defined.

The d_name field

The *dirent* structure definition shown above is taken from the glibc headers, and shows the *d_name* field with a fixed size.

Warning: applications should avoid any dependence on the size of the *d_name* field. POSIX defines it as *char d_name[]*, a character array of unspecified size, with at most **NAME_MAX** characters preceding the terminating null byte ('\0').

POSIX.1 explicitly notes that this field should not be used as an lvalue. The standard also notes that the use of *sizeof(d_name)* is incorrect; use *strlen(d_name)* instead. (On some systems, this field is defined as *char d_name[1]*!) By implication, the use *sizeof(struct dirent)* to capture the size of the record including the size of *d_name* is also incorrect.

Note that while the call

```
fpathconf(fd, _PC_NAME_MAX)
```

returns the value 255 for most filesystems, on some filesystems (e.g., CIFS, Windows SMB servers), the null-terminated filename that is (correctly) returned in *d_name* can actually exceed this size. In such cases, the *d_reclen* field will contain a value that exceeds the size of the glibc *dirent* structure shown above.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

NOTES

A directory stream is opened using *opendir*(3).

The order in which filenames are read by successive calls to **readdir**() depends on the filesystem

implementation; it is unlikely that the names will be sorted in any fashion.

SEE ALSO

getdents(2), *read*(2), *closedir*(3), *dirfd*(3), *ftw*(3), *offsetof*(3), *opendir*(3), *readdir_r*(3), *rewinddir*(3), *scandir*(3), *seekdir*(3), *telldir*(3)

NAME

readdir_r − read a directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <dirent.h>

[[deprecated]] int readdir_r(DIR *restrict dirp,
                             struct dirent *restrict entry,
                             struct dirent **restrict result);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

readdir_r():
_POSIX_C_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

This function is deprecated; use *readdir*(3) instead.

The **readdir_r**() function was invented as a reentrant version of *readdir*(3). It reads the next directory entry from the directory stream *dirp*, and returns it in the caller-allocated buffer pointed to by *entry*. For details of the *dirent* structure, see *readdir*(3).

A pointer to the returned buffer is placed in **result*; if the end of the directory stream was encountered, then NULL is instead returned in **result*.

It is recommended that applications use *readdir*(3) instead of **readdir_r**(). Furthermore, since glibc 2.24, glibc deprecates **readdir_r**(). The reasons are as follows:

- On systems where **NAME_MAX** is undefined, calling **readdir_r**() may be unsafe because the interface does not allow the caller to specify the length of the buffer used for the returned directory entry.
- On some systems, **readdir_r**() can't read directory entries with very long names. When the glibc implementation encounters such a name, **readdir_r**() fails with the error **ENAMETOOLONG** *after the final directory entry has been read*. On some other systems, **readdir_r**() may return a success status, but the returned *d_name* field may not be null terminated or may be truncated.
- In the current POSIX.1 specification (POSIX.1-2008), *readdir*(3) is not required to be thread-safe. However, in modern implementations (including the glibc implementation), concurrent calls to *readdir*(3) that specify different directory streams are thread-safe. Therefore, the use of **readdir_r**() is generally unnecessary in multithreaded programs. In cases where multiple threads must read from the same directory stream, using *readdir*(3) with external synchronization is still preferable to the use of **readdir_r**(), for the reasons given in the points above.
- It is expected that a future version of POSIX.1 will make **readdir_r**() obsolete, and require that *readdir*(3) be thread-safe when concurrently employed on different directory streams.

RETURN VALUE

The **readdir_r**() function returns 0 on success. On error, it returns a positive error number (listed under ERRORS). If the end of the directory stream is reached, **readdir_r**() returns 0, and returns NULL in **result*.

ERRORS

EBADF
Invalid directory stream descriptor *dirp*.

ENAMETOOLONG
A directory entry whose name was too long to be read was encountered.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
readdir_r()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

readdir(3)

NAME

realpath − return the canonicalized absolute pathname

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <limits.h>
#include <stdlib.h>

char *realpath(const char *restrict path,
               char *restrict resolved_path);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

realpath():

```
_XOPEN_SOURCE >= 500
    || /* glibc >= 2.19: */ _DEFAULT_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION

realpath() expands all symbolic links and resolves references to */./*, */../* and extra '/' characters in the null-terminated string named by *path* to produce a canonicalized absolute pathname. The resulting pathname is stored as a null-terminated string, up to a maximum of **PATH_MAX** bytes, in the buffer pointed to by *resolved_path*. The resulting path will have no symbolic link, */./* or */../* components.

If *resolved_path* is specified as NULL, then **realpath**() uses *malloc*(3) to allocate a buffer of up to **PATH_MAX** bytes to hold the resolved pathname, and returns a pointer to this buffer. The caller should deallocate this buffer using *free*(3).

RETURN VALUE

If there is no error, **realpath**() returns a pointer to the *resolved_path*.

Otherwise, it returns NULL, the contents of the array *resolved_path* are undefined, and *errno* is set to indicate the error.

ERRORS

EACCES
: Read or search permission was denied for a component of the path prefix.

EINVAL
: *path* is NULL. (Before glibc 2.3, this error is also returned if *resolved_path* is NULL.)

EIO An I/O error occurred while reading from the filesystem.

ELOOP
: Too many symbolic links were encountered in translating the pathname.

ENAMETOOLONG
: A component of a pathname exceeded **NAME_MAX** characters, or an entire pathname exceeded **PATH_MAX** characters.

ENOENT
: The named file does not exist.

ENOMEM
: Out of memory.

ENOTDIR
: A component of the path prefix is not a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
realpath()	Thread safety	MT-Safe

VERSIONS

GNU extensions

If the call fails with either **EACCES** or **ENOENT** and *resolved_path* is not NULL, then the prefix of *path* that is not readable or does not exist is returned in *resolved_path*.

STANDARDS

POSIX.1-2008.

HISTORY

4.4BSD, POSIX.1-2001, Solaris.

POSIX.1-2001 says that the behavior if *resolved_path* is NULL is implementation-defined. POSIX.1-2008 specifies the behavior described in this page.

In 4.4BSD and Solaris, the limit on the pathname length is **MAXPATHLEN** (found in <*sys/param.h*>). SUSv2 prescribes **PATH_MAX** and **NAME_MAX**, as found in <*limits.h*> or provided by the *pathconf*(3) function. A typical source fragment would be

```
#ifdef PATH_MAX
  path_max = PATH_MAX;
#else
  path_max = pathconf(path, _PC_PATH_MAX);
  if (path_max <= 0)
    path_max = 4096;
#endif
```

(But see the BUGS section.)

BUGS

The POSIX.1-2001 standard version of this function is broken by design, since it is impossible to determine a suitable size for the output buffer, *resolved_path*. According to POSIX.1-2001 a buffer of size **PATH_MAX** suffices, but **PATH_MAX** need not be a defined constant, and may have to be obtained using *pathconf*(3). And asking *pathconf*(3) does not really help, since, on the one hand POSIX warns that the result of *pathconf*(3) may be huge and unsuitable for mallocing memory, and on the other hand *pathconf*(3) may return −1 to signify that **PATH_MAX** is not bounded. The *resolved_path* == *NULL* feature, not standardized in POSIX.1-2001, but standardized in POSIX.1-2008, allows this design problem to be avoided.

SEE ALSO

realpath(1), *readlink*(2), *canonicalize_file_name*(3), *getcwd*(3), *pathconf*(3), *sysconf*(3)

NAME

recno − record number database access method

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/types.h>
#include <db.h>

DESCRIPTION

Note well: This page documents interfaces provided up until glibc 2.1. Since glibc 2.2, glibc no longer provides these interfaces. Probably, you are looking for the APIs provided by the *libdb* library instead.

The routine *dbopen*(3) is the library interface to database files. One of the supported file formats is record number files. The general description of the database access methods is in *dbopen*(3), this manual page describes only the recno-specific information.

The record number data structure is either variable or fixed-length records stored in a flat-file format, accessed by the logical record number. The existence of record number five implies the existence of records one through four, and the deletion of record number one causes record number five to be renumbered to record number four, as well as the cursor, if positioned after record number one, to shift down one record.

The recno access-method-specific data structure provided to *dbopen*(3) is defined in the <*db.h*> include file as follows:

```
typedef struct {
    unsigned long flags;
    unsigned int  cachesize;
    unsigned int  psize;
    int           lorder;
    size_t        reclen;
    unsigned char bval;
    char         *bfname;
} RECNOINFO;
```

The elements of this structure are defined as follows:

flags The flag value is specified by ORing any of the following values:

R_FIXEDLEN
The records are fixed-length, not byte delimited. The structure element *reclen* specifies the length of the record, and the structure element *bval* is used as the pad character. Any records, inserted into the database, that are less than *reclen* bytes long are automatically padded.

R_NOKEY
In the interface specified by *dbopen*(3), the sequential record retrieval fills in both the caller's key and data structures. If the **R_NOKEY** flag is specified, the *cursor* routines are not required to fill in the key structure. This permits applications to retrieve records at the end of files without reading all of the intervening records.

R_SNAPSHOT
This flag requires that a snapshot of the file be taken when *dbopen*(3) is called, instead of permitting any unmodified records to be read from the original file.

cachesize
A suggested maximum size, in bytes, of the memory cache. This value is **only** advisory, and the access method will allocate more memory rather than fail. If *cachesize* is 0 (no size is specified), a default cache is used.

psize The recno access method stores the in-memory copies of its records in a btree. This value is the size (in bytes) of the pages used for nodes in that tree. If *psize* is 0 (no page size is specified), a page size is chosen based on the underlying filesystem I/O block size. See *btree*(3) for more information.

lorder The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified), the current host order is used.

reclen The length of a fixed-length record.

bval The delimiting byte to be used to mark the end of a record for variable-length records, and the pad character for fixed-length records. If no value is specified, newlines ("\n") are used to mark the end of variable-length records and fixed-length records are padded with spaces.

bfname The recno access method stores the in-memory copies of its records in a btree. If *bfname* is non-NULL, it specifies the name of the btree file, as if specified as the filename for a *dbopen*(3) of a btree file.

The data part of the key/data pair used by the *recno* access method is the same as other access methods. The key is different. The *data* field of the key should be a pointer to a memory location of type *recno_t*, as defined in the <*db.h*> include file. This type is normally the largest unsigned integral type available to the implementation. The *size* field of the key should be the size of that type.

Because there can be no metadata associated with the underlying recno access method files, any changes made to the default values (e.g., fixed record length or byte separator value) must be explicitly specified each time the file is opened.

In the interface specified by *dbopen*(3), using the *put* interface to create a new record will cause the creation of multiple, empty records if the record number is more than one greater than the largest record currently in the database.

ERRORS

The *recno* access method routines may fail and set *errno* for any of the errors specified for the library routine *dbopen*(3) or the following:

EINVAL
An attempt was made to add a record to a fixed-length database that was too large to fit.

BUGS

Only big and little endian byte order is supported.

SEE ALSO

btree(3), *dbopen*(3), *hash*(3), *mpool*(3)

Document Processing in a Relational Database System, Michael Stonebraker, Heidi Stettner, Joseph Kalash, Antonin Guttman, Nadene Lynn, Memorandum No. UCB/ERL M82/32, May 1982.

NAME

regcomp, regexec, regerror, regfree − POSIX regex functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict regex,
        int cflags);
int regexec(const regex_t *restrict preg, const char *restrict string,
        size_t nmatch, regmatch_t pmatch[_Nullable restrict .nmatch],
        int eflags);

size_t regerror(int errcode, const regex_t *_Nullable restrict preg,
        char errbuf[_Nullable restrict .errbuf_size],
        size_t errbuf_size);
void regfree(regex_t *preg);

typedef struct {
   size_t    re_nsub;
} regex_t;

typedef struct {
   regoff_t  rm_so;
   regoff_t  rm_eo;
} regmatch_t;

typedef /* ... */  regoff_t;
```

DESCRIPTION

Compilation

regcomp() is used to compile a regular expression into a form that is suitable for subsequent **regexec**() searches.

On success, the pattern buffer at **preg* is initialized. *regex* is a null-terminated string. The locale must be the same when running **regexec**().

After **regcomp**() succeeds, *preg->re_nsub* holds the number of subexpressions in *regex*. Thus, a value of *preg->re_nsub* + 1 passed as *nmatch* to **regexec**() is sufficient to capture all matches.

cflags is the bitwise OR of zero or more of the following:

REG_EXTENDED
: Use POSIX Extended Regular Expression syntax when interpreting *regex*. If not set, POSIX Basic Regular Expression syntax is used.

REG_ICASE
: Do not differentiate case. Subsequent **regexec**() searches using this pattern buffer will be case insensitive.

REG_NOSUB
: Report only overall success. **regexec**() will use only *pmatch* for **REG_STARTEND**, ignoring *nmatch*.

REG_NEWLINE
: Match-any-character operators don't match a newline.

  A nonmatching list (**[^...]**) not containing a newline does not match a newline.

  Match-beginning-of-line operator (^) matches the empty string immediately after a newline, regardless of whether *eflags*, the execution flags of **regexec**(), contains **REG_NOTBOL**.

  Match-end-of-line operator (**$**) matches the empty string immediately before a newline, regardless of whether *eflags* contains **REG_NOTEOL**.

Matching

regexec() is used to match a null-terminated string against the compiled pattern buffer in **preg*, which must have been initialised with **regexec**(). *eflags* is the bitwise OR of zero or more of the following

flags:

REG_NOTBOL
The match-beginning-of-line operator always fails to match (but see the compilation flag **REG_NEWLINE** above). This flag may be used when different portions of a string are passed to **regexec**() and the beginning of the string should not be interpreted as the beginning of the line.

REG_NOTEOL
The match-end-of-line operator always fails to match (but see the compilation flag **REG_NEWLINE** above).

REG_STARTEND
Match [*string* + *pmatch[0].rm_so*, *string* + *pmatch[0].rm_eo*) instead of [*string*, *string* + *strlen(string)*). This allows matching embedded NUL bytes and avoids a *strlen*(3) on known-length strings. If any matches are returned (**REG_NOSUB** wasn't passed to **regcomp**(), the match succeeded, and *nmatch* > 0), they overwrite *pmatch* as usual, and the match offsets remain relative to *string* (not *string* + *pmatch[0].rm_so*). This flag is a BSD extension, not present in POSIX.

Match offsets
Unless **REG_NOSUB** was passed to **regcomp**(), it is possible to obtain the locations of matches within *string*: **regexec**() fills *nmatch* elements of *pmatch* with results: *pmatch[0]* corresponds to the entire match, *pmatch[1]* to the first subexpression, etc. If there were more matches than *nmatch*, they are discarded; if fewer, unused elements of *pmatch* are filled with **-1**s.

Each returned valid (non-**−1**) match corresponds to the range [*string* + *rm_so*, *string* + *rm_eo*).

regoff_t is a signed integer type capable of storing the largest value that can be stored in either an *ptrdiff_t* type or a *ssize_t* type.

Error reporting
regerror() is used to turn the error codes that can be returned by both **regcomp**() and **regexec**() into error message strings.

If *preg* isn't a null pointer, *errcode* must be the latest error returned from an operation on *preg*.

If *errbuf_size* isn't 0, up to *errbuf_size* bytes are copied to *errbuf*; the error string is always null-terminated, and truncated to fit.

Freeing
regfree() deinitializes the pattern buffer at **preg*, freeing any associated memory; **preg* must have been initialized via **regcomp**().

RETURN VALUE

regcomp() returns zero for a successful compilation or an error code for failure.

regexec() returns zero for a successful match or **REG_NOMATCH** for failure.

regerror() returns the size of the buffer required to hold the string.

ERRORS

The following errors can be returned by **regcomp**():

REG_BADBR
Invalid use of back reference operator.

REG_BADPAT
Invalid use of pattern operators such as group or list.

REG_BADRPT
Invalid use of repetition operators such as using '*' as the first character.

REG_EBRACE
Un-matched brace interval operators.

REG_EBRACK
Un-matched bracket list operators.

REG_ECOLLATE
Invalid collating element.

REG_ECTYPE
Unknown character class name.

REG_EEND
Nonspecific error. This is not defined by POSIX.

REG_EESCAPE
Trailing backslash.

REG_EPAREN
Un-matched parenthesis group operators.

REG_ERANGE
Invalid use of the range operator; for example, the ending point of the range occurs prior to the starting point.

REG_ESIZE
Compiled regular expression requires a pattern buffer larger than 64 kB. This is not defined by POSIX.

REG_ESPACE
The regex routines ran out of memory.

REG_ESUBREG
Invalid back reference to a subexpression.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
regcomp(), **regexec**()	Thread safety	MT-Safe locale
regerror()	Thread safety	MT-Safe env
regfree()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Prior to POSIX.1-2008, *regoff_t* was required to be capable of storing the largest value that can be stored in either an *off_t* type or a *ssize_t* type.

CAVEATS

re_nsub is only required to be initialized if **REG_NOSUB** wasn't specified, but all known implementations initialize it regardless.

Both *regex_t* and *regmatch_t* may (and do) have more members, in any order. Always reference them by name.

EXAMPLES

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>

#define ARRAY_SIZE(arr) (sizeof((arr)) / sizeof((arr)[0]))

static const char *const str =
        "1) John Driverhacker;\n2) John Doe;\n3) John Foo;\n";
static const char *const re = "John.*o";

int main(void)
{
    static const char *s = str;
```

```
        regex_t     regex;
        regmatch_t  pmatch[1];
        regoff_t    off, len;

        if (regcomp(&regex, re, REG_NEWLINE))
            exit(EXIT_FAILURE);

        printf("String = \"%s\"\n", str);
        printf("Matches:\n");

        for (unsigned int i = 0; ; i++) {
            if (regexec(&regex, s, ARRAY_SIZE(pmatch), pmatch, 0))
                break;

            off = pmatch[0].rm_so + (s - str);
            len = pmatch[0].rm_eo - pmatch[0].rm_so;
            printf("#%zu:\n", i);
            printf("offset = %jd; length = %jd\n", (intmax_t) off,
                    (intmax_t) len);
            printf("substring = \"%.*s\"\n", len, s + pmatch[0].rm_so);

            s += pmatch[0].rm_eo;
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

grep(1), *regex*(7)

The glibc manual section, *Regular Expressions*

NAME

drem, dremf, dreml, remainder, remainderf, remainderl − floating-point remainder function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

/* Obsolete synonyms */
[[deprecated]] double drem(double x, double y);
[[deprecated]] float dremf(float x, float y);
[[deprecated]] long double dreml(long double x, long double y);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

remainder():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

remainderf(), **remainderl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

drem(), **dremf**(), **dreml**():
/* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions compute the remainder of dividing *x* by *y*. The return value is *x*−*n***y*, where *n* is the value *x* / *y*, rounded to the nearest integer. If the absolute value of *x*−*n***y* is 0.5, *n* is chosen to be even.

These functions are unaffected by the current rounding mode (see *fenv*(3)).

The **drem**() function does precisely the same thing.

RETURN VALUE

On success, these functions return the floating-point remainder, *x*−*n***y*. If the return value is 0, it has the sign of *x*.

If *x* or *y* is a NaN, a NaN is returned.

If *x* is an infinity, and *y* is not a NaN, a domain error occurs, and a NaN is returned.

If *y* is zero, and *x* is not a NaN, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity and *y* is not a NaN
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

  These functions do not set *errno* for this case.

Domain error: *y* is zero
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
drem(), **dremf**(), **dreml**(), **remainder**(), **remainderf**(), **remainderl**()	Thread safety	MT-Safe

STANDARDS

remainder()
remainderf()
remainderl()
C11, POSIX.1-2008.

drem()
dremf()
dreml()
None.

HISTORY

remainder()
remainderf()
remainderl()
C99, POSIX.1-2001.

drem() 4.3BSD.

dremf()
dreml()
Tru64, glibc2.

BUGS

Before glibc 2.15, the call

```
remainder(nan(""), 0);
```

returned a NaN, as expected, but wrongly caused a domain error. Since glibc 2.15, a silent NaN (i.e., no domain error) is returned.

Before glibc 2.15, *errno* was not set to **EDOM** for the domain error that occurs when *x* is an infinity and *y* is not a NaN.

EXAMPLES

The call "remainder(29.0, 3.0)" returns −1.

SEE ALSO

div(3), *fmod*(3), *remquo*(3)

NAME

remove − remove a file or directory

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

int remove(const char ****pathname*);**

DESCRIPTION

remove() deletes a name from the filesystem. It calls *unlink*(2) for files, and *rmdir*(2) for directories.

If the removed name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file, but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name is removed, but processes which have the object open may continue to use it.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

The errors that occur are those for *unlink*(2) and *rmdir*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
remove()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, 4.3BSD.

BUGS

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of files which are still being used.

SEE ALSO

rm(1), *unlink*(1), *link*(2), *mknod*(2), *open*(2), *rename*(2), *rmdir*(2), *unlink*(2), *mkfifo*(3), *symlink*(7)

NAME

remquo, remquof, remquol − remainder and part of quotient

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

remquo(), **remquof**(), **remquol**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions compute the remainder and part of the quotient upon division of *x* by *y*. A few bits of the quotient are stored via the *quo* pointer. The remainder is returned as the function result.

The value of the remainder is the same as that computed by the *remainder*(3) function.

The value stored via the *quo* pointer has the sign of *x / y* and agrees with the quotient in at least the low order 3 bits.

For example, *remquo(29.0, 3.0)* returns −1.0 and might store 2. Note that the actual quotient might not fit in an integer.

RETURN VALUE

On success, these functions return the same value as the analogous functions described in *remainder*(3).

If *x* or *y* is a NaN, a NaN is returned.

If *x* is an infinity, and *y* is not a NaN, a domain error occurs, and a NaN is returned.

If *y* is zero, and *x* is not a NaN, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity or *y* is 0, and the other argument is not a NaN
An invalid floating-point exception (**FE_INVALID**) is raised.

These functions do not set *errno*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
remquo(), **remquof**(), **remquol**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

SEE ALSO

fmod(3), *logb*(3), *remainder*(3)

NAME

res_ninit, res_nquery, res_nsearch, res_nquerydomain, res_nmkquery, res_nsend, res_nclose, res_init, res_query, res_search, res_querydomain, res_mkquery, res_send, dn_comp, dn_expand − resolver routines

LIBRARY

Resolver library (*libresolv*, *−lresolv*)

SYNOPSIS

```
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

struct __res_state;
typedef struct __res_state *res_state;

int res_ninit(res_state statep);

void res_nclose(res_state statep);

int res_nquery(res_state statep,
        const char *dname, int class, int type,
        unsigned char answer[.anslen], int anslen);

int res_nsearch(res_state statep,
        const char *dname, int class, int type,
        unsigned char answer[.anslen], int anslen);

int res_nquerydomain(res_state statep,
        const char *name, const char *domain,
        int class, int type, unsigned char answer[.anslen],
        int anslen);

int res_nmkquery(res_state statep,
        int op, const char *dname, int class,
        int type, const unsigned char data[.datalen], int datalen,
        const unsigned char *newrr,
        unsigned char buf [.buflen], int buflen);

int res_nsend(res_state statep,
        const unsigned char msg[.msglen], int msglen,
        unsigned char answer[.anslen], int anslen);

int dn_comp(const char *exp_dn, unsigned char comp_dn[.length],
        int length, unsigned char **dnptrs,
        unsigned char **lastdnptr);

int dn_expand(const unsigned char *msg,
        const unsigned char *eomorig,
        const unsigned char *comp_dn, char exp_dn[.length],
        int length);

[[deprecated]] extern struct __res_state _res;

[[deprecated]] int res_init(void);

[[deprecated]]
int res_query(const char *dname, int class, int type,
        unsigned char answer[.anslen], int anslen);

[[deprecated]]
int res_search(const char *dname, int class, int type,
        unsigned char answer[.anslen], int anslen);

[[deprecated]]
int res_querydomain(const char *name, const char *domain,
        int class, int type, unsigned char answer[.anslen],
        int anslen);
```

[[deprecated]]
int res_mkquery(int *op***, const char ****dname***, int** *class***,**
int *type***, const unsigned char** *data***[.***datalen***], int** *datalen***,**
const unsigned char ****newrr*,**
unsigned char *buf* **[.***buflen***], int** *buflen***);**

[[deprecated]]
int res_send(const unsigned char *msg***[.***msglen***], int** *msglen***,**
unsigned char *answer***[.***anslen***], int** *anslen***);**

DESCRIPTION

Note: This page is incomplete (various resolver functions provided by glibc are not described) and likely out of date.

The functions described below make queries to and interpret the responses from Internet domain name servers.

The API consists of a set of more modern, reentrant functions and an older set of nonreentrant functions that have been superseded. The traditional resolver interfaces such as **res_init**() and **res_query**() use some static (global) state stored in the *_res* structure, rendering these functions non-thread-safe. BIND 8.2 introduced a set of new interfaces **res_ninit**(), **res_nquery**(), and so on, which take a *res_state* as their first argument, so you can use a per-thread resolver state.

The **res_ninit**() and **res_init**() functions read the configuration files (see *resolv.conf*(5)) to get the default domain name and name server address(es). If no server is given, the local host is tried. If no domain is given, that associated with the local host is used. It can be overridden with the environment variable **LOCALDOMAIN**. **res_ninit**() or **res_init**() is normally executed by the first call to one of the other functions. Every call to **res_ninit**() requires a corresponding call to **res_nclose**() to free memory allocated by **res_ninit**() and subsequent calls to **res_nquery**().

The **res_nquery**() and **res_query**() functions query the name server for the fully qualified domain name *name* of specified *type* and *class*. The reply is left in the buffer *answer* of length *anslen* supplied by the caller.

The **res_nsearch**() and **res_search**() functions make a query and waits for the response like **res_nquery**() and **res_query**(), but in addition they implement the default and search rules controlled by **RES_DEFNAMES** and **RES_DNSRCH** (see description of *_res* options below).

The **res_nquerydomain**() and **res_querydomain**() functions make a query using *res_nquery*()/*res_query*() on the concatenation of *name* and *domain*.

The following functions are lower-level routines used by *res_nquery*()/*res_query*()

The **res_nmkquery**() and **res_mkquery**() functions construct a query message in *buf* of length *buflen* for the domain name *dname*. The query type *op* is one of the following (typically **QUERY**):

QUERY
Standard query.

IQUERY
Inverse query. This option was removed in glibc 2.26, since it has not been supported by DNS servers for a very long time.

NS_NOTIFY_OP
Notify secondary of SOA (Start of Authority) change.

newrr is currently unused.

The **res_nsend**() and **res_send**() function send a preformatted query given in *msg* of length *msglen* and returns the answer in *answer* which is of length *anslen*. They will call *res_ninit*()/*res_init*() if it has not already been called.

The **dn_comp**() function compresses the domain name *exp_dn* and stores it in the buffer *comp_dn* of length *length*. The compression uses an array of pointers *dnptrs* to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit of the array is specified by *lastdnptr*. If *dnptr* is NULL, domain names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

The **dn_expand**() function expands the compressed domain name *comp_dn* to a full domain name,

which is placed in the buffer *exp_dn* of size *length*. The compressed name is contained in a query or reply message, and *msg* points to the beginning of the message.

The resolver routines use configuration and state information contained in a *__res_state* structure (either passed as the *statep* argument, or in the global variable *_res*, in the case of the older nonreentrant functions). The only field of this structure that is normally manipulated by the user is the *options* field. This field can contain the bitwise "OR" of the following options:

RES_INIT
: True if **res_ninit**() or **res_init**() has been called.

RES_DEBUG
: Print debugging messages. This option is available only if glibc was built with debugging enabled, which is not the default.

RES_AAONLY (unimplemented; deprecated in glibc 2.25)
: Accept authoritative answers only. **res_send**() continues until it finds an authoritative answer or returns an error. This option was present but unimplemented until glibc 2.24; since glibc 2.25, it is deprecated, and its usage produces a warning.

RES_USEVC
: Use TCP connections for queries rather than UDP datagrams.

RES_PRIMARY (unimplemented; deprecated in glibc 2.25)
: Query primary domain name server only. This option was present but unimplemented until glibc 2.24; since glibc 2.25, it is deprecated, and its usage produces a warning.

RES_IGNTC
: Ignore truncation errors. Don't retry with TCP.

RES_RECURSE
: Set the recursion desired bit in queries. Recursion is carried out by the domain name server, not by **res_send**(). [Enabled by default].

RES_DEFNAMES
: If set, **res_search**() will append the default domain name to single component names—that is, those that do not contain a dot. [Enabled by default].

RES_STAYOPEN
: Used with **RES_USEVC** to keep the TCP connection open between queries.

RES_DNSRCH
: If set, **res_search**() will search for hostnames in the current domain and in parent domains. This option is used by *gethostbyname*(3). [Enabled by default].

RES_INSECURE1
: Accept a response from a wrong server. This can be used to detect potential security hazards, but you need to compile glibc with debugging enabled and use **RES_DEBUG** option (for debug purpose only).

RES_INSECURE2
: Accept a response which contains a wrong query. This can be used to detect potential security hazards, but you need to compile glibc with debugging enabled and use **RES_DEBUG** option (for debug purpose only).

RES_NOALIASES
: Disable usage of **HOSTALIASES** environment variable.

RES_USE_INET6
: Try an AAAA query before an A query inside the *gethostbyname*(3) function, and map IPv4 responses in IPv6 "tunneled form" if no AAAA records are found but an A record set exists. Since glibc 2.25, this option is deprecated, and its usage produces a warning; applications should use *getaddrinfo*(3), rather than *gethostbyname*(3).

RES_ROTATE
: Causes round-robin selection of name servers from among those listed. This has the effect of spreading the query load among all listed servers, rather than having all clients try the first listed server first every time.

RES_NOCHECKNAME (unimplemented; deprecated in glibc 2.25)
Disable the modern BIND checking of incoming hostnames and mail names for invalid characters such as underscore (_), non-ASCII, or control characters. This option was present until glibc 2.24; since glibc 2.25, it is deprecated, and its usage produces a warning.

RES_KEEPTSIG (unimplemented; deprecated in glibc 2.25)
Do not strip TSIG records. This option was present but unimplemented until glibc 2.24; since glibc 2.25, it is deprecated, and its usage produces a warning.

RES_BLAST (unimplemented; deprecated in glibc 2.25)
Send each query simultaneously and recursively to all servers. This option was present but unimplemented until glibc 2.24; since glibc 2.25, it is deprecated, and its usage produces a warning.

RES_USEBSTRING (glibc 2.3.4 to glibc 2.24)
Make reverse IPv6 lookups using the bit-label format described in RFC 2673; if this option is not set (which is the default), then nibble format is used. This option was removed in glibc 2.25, since it relied on a backward-incompatible DNS extension that was never deployed on the Internet.

RES_NOIP6DOTINT (glibc 2.24 and earlier)
Use *ip6.arpa* zone in IPv6 reverse lookup instead of *ip6.int*, which is deprecated since glibc 2.3.4. This option is present up to and including glibc 2.24, where it is enabled by default. In glibc 2.25, this option was removed.

RES_USE_EDNS0 (since glibc 2.6)
Enables support for the DNS extensions (EDNS0) described in RFC 2671.

RES_SNGLKUP (since glibc 2.10)
By default, glibc performs IPv4 and IPv6 lookups in parallel since glibc 2.9. Some appliance DNS servers cannot handle these queries properly and make the requests time out. This option disables the behavior and makes glibc perform the IPv6 and IPv4 requests sequentially (at the cost of some slowdown of the resolving process).

RES_SNGLKUPREOP
When **RES_SNGLKUP** option is enabled, opens a new socket for the each request.

RES_USE_DNSSEC
Use DNSSEC with OK bit in OPT record. This option implies **RES_USE_EDNS0**.

RES_NOTLDQUERY
Do not look up unqualified name as a top-level domain (TLD).

RES_DEFAULT
Default option which implies: **RES_RECURSE**, **RES_DEFNAMES**, **RES_DNSRCH**, and **RES_NOIP6DOTINT**.

RETURN VALUE

The **res_ninit**() and **res_init**() functions return 0 on success, or −1 if an error occurs.

The **res_nquery**(), **res_query**(), **res_nsearch**(), **res_search**(), **res_nquerydomain**(), **res_querydomain**(), **res_nmkquery**(), **res_mkquery**(), **res_nsend**(), and **res_send**() functions return the length of the response, or −1 if an error occurs.

The **dn_comp**() and **dn_expand**() functions return the length of the compressed name, or −1 if an error occurs.

In the case of an error return from **res_nquery**(), **res_query**(), **res_nsearch**(), **res_search**(), **res_nquerydomain**(), or **res_querydomain**(), the global variable *h_errno* (see *gethostbyname*(3)) can be consulted to determine the cause of the error.

FILES

/etc/resolv.conf
resolver configuration file

/etc/host.conf
resolver configuration file

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
res_ninit(), **res_nclose**(), **res_nquery**(), **res_nsearch**(), **res_nquerydomain**(), **res_nsend**()	Thread safety	MT-Safe locale
res_nmkquery(), **dn_comp**(), **dn_expand**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD.

SEE ALSO

gethostbyname(3), *resolv.conf*(5), *resolver*(5), *hostname*(7), *named*(8)

The GNU C library source file *resolv/README*.

NAME

rewinddir − reset directory stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir(DIR *dirp);
```

DESCRIPTION

The **rewinddir**() function resets the position of the directory stream *dirp* to the beginning of the directory.

RETURN VALUE

The **rewinddir**() function returns no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rewinddir()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

closedir(3), *opendir*(3), *readdir*(3), *scandir*(3), *seekdir*(3), *telldir*(3)

NAME

rexec, rexec_af − return stream to a remote command

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

[[deprecated]]
int rexec(char **restrict ahost, int inport,
          const char *restrict user, const char *restrict passwd,
          const char *restrict cmd, int *restrict fd2p);

[[deprecated]]
int rexec_af(char **restrict ahost, int inport,
          const char *restrict user, const char *restrict passwd,
          const char *restrict cmd, int *restrict fd2p,
          sa_family_t af);
```

rexec(), **rexec_af**():

Since glibc 2.19:

_DEFAULT_SOURCE

In glibc up to and including 2.19:

_BSD_SOURCE

DESCRIPTION

This interface is obsoleted by *rcmd*(3).

The **rexec**() function looks up the host **ahost* using *gethostbyname*(3), returning −1 if the host does not exist. Otherwise, **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the *.netrc* file in user's home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call *getservbyname("exec", "tcp")* (see *getservent*(3)) will return a pointer to a structure that contains the necessary port. The protocol for connection is described in detail in *rexecd*(8)

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the caller, and given to the remote command as *stdin* and *stdout*. If *fd2p* is nonzero, then an auxiliary channel to a control process will be setup, and a file descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

rexec_af()

The **rexec**() function works over IPv4 (**AF_INET**). By contrast, the **rexec_af**() function provides an extra argument, *af*, that allows the caller to select the protocol. This argument can be specified as **AF_INET**, **AF_INET6**, or **AF_UNSPEC** (to allow the implementation to select the protocol).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rexec(), **rexec_af**()	Thread safety	MT-Unsafe

STANDARDS

None.

HISTORY

rexec() 4.2BSD, BSD, Solaris.

rexec_af()
glibc 2.2.

BUGS

The **rexec**() function sends the unencrypted password across the network.

The underlying service is considered a big security hole and therefore not enabled on many sites; see *rexecd*(8) for explanations.

SEE ALSO

rcmd(3), *rexecd*(8)

NAME

nearbyint, nearbyintf, nearbyintl, rint, rintf, rintl − round to nearest integer

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);

double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

nearbyint(), **nearbyintf**(), **nearbyintl**():
_POSIX_C_SOURCE >= 200112L || _ISOC99_SOURCE

rint():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

rintf(), **rintl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **nearbyint**(), **nearbyintf**(), and **nearbyintl**() functions round their argument to an integer value in floating-point format, using the current rounding direction (see *fesetround*(3)) and without raising the *inexact* exception. When the current rounding direction is to nearest, these functions round halfway cases to the even integer in accordance with IEEE-754.

The **rint**(), **rintf**(), and **rintl**() functions do the same, but will raise the *inexact* exception (**FE_INEXACT**, checkable via *fetestexcept*(3)) when the result differs in value from the argument.

RETURN VALUE

These functions return the rounded integer value.

If *x* is integral, +0, −0, NaN, or infinite, *x* itself is returned.

ERRORS

No errors occur. POSIX.1-2001 documents a range error for overflows, but see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
nearbyint(), **nearbyintf**(), **nearbyintl**(), **rint**(), **rintf**(), **rintl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

SUSv2 and POSIX.1-2001 contain text about overflow (which might set *errno* to **ERANGE**, or raise an **FE_OVERFLOW** exception). In practice, the result cannot overflow on any current machine, so this error-handling stuff is just nonsense. (More precisely, overflow can happen only when the maximum value of the exponent is smaller than the number of mantissa bits. For the IEEE-754 standard 32-bit and 64-bit floating-point numbers the maximum value of the exponent is 127 (respectively, 1023), and the number of mantissa bits including the implicit bit is 24 (respectively, 53).)

If you want to store the rounded value in an integer type, you probably want to use one of the functions described in *lrint*(3) instead.

SEE ALSO

ceil(3), *floor*(3), *lrint*(3), *round*(3), *trunc*(3)

NAME

round, roundf, roundl − round to nearest integer, away from zero

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

round(), **roundf**(), **roundl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions round *x* to the nearest integer, but round halfway cases away from zero (regardless of the current rounding direction, see *fenv*(3)), instead of to the nearest even integer like *rint*(3).

For example, *round(0.5)* is 1.0, and *round(−0.5)* is −1.0.

RETURN VALUE

These functions return the rounded integer value.

If *x* is integral, +0, −0, NaN, or infinite, *x* itself is returned.

ERRORS

No errors occur. POSIX.1-2001 documents a range error for overflows, but see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
round(), **roundf**(), **roundl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

NOTES

POSIX.1-2001 contains text about overflow (which might set *errno* to **ERANGE**, or raise an **FE_OVERFLOW** exception). In practice, the result cannot overflow on any current machine, so this error-handling stuff is just nonsense. (More precisely, overflow can happen only when the maximum value of the exponent is smaller than the number of mantissa bits. For the IEEE-754 standard 32-bit and 64-bit floating-point numbers the maximum value of the exponent is 127 (respectively, 1023), and the number of mantissa bits including the implicit bit is 24 (respectively, 53).)

If you want to store the rounded value in an integer type, you probably want to use one of the functions described in *lround*(3) instead.

SEE ALSO

ceil(3), *floor*(3), *lround*(3), *nearbyint*(3), *rint*(3), *trunc*(3)

NAME

roundup − round up in steps

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <sys/param.h>

roundup(*x***,** *step***);**

DESCRIPTION

This macro rounds *x* to the nearest multiple of *step* that is not less than *x*.

It is typically used for rounding up a pointer to align it or increasing a buffer to be allocated.

This API is not designed to be generic, and doesn't work in some cases that are not important for the typical use cases described above. See CAVEATS.

RETURN VALUE

This macro returns the rounded value.

STANDARDS

None.

CAVEATS

The arguments may be evaluated more than once.

x should be nonnegative, and *step* should be positive.

If *x* + *step* would overflow or wrap around, the behavior is undefined.

SEE ALSO

ceil(3), *floor*(3), *lrint*(3), *rint*(3), *lround*(3), *round*(3)

NAME

rpc − library routines for remote procedure calls

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS AND DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

To take use of these routines, include the header file *<rpc/rpc.h>*.

The prototypes below make use of the following types:

```
typedef int bool_t;

typedef bool_t (*xdrproc_t)(XDR *, void *, ...);

typedef bool_t (*resultproc_t)(caddr_t resp,
                               struct sockaddr_in *raddr);
```

See the header files for the declarations of the *AUTH*, *CLIENT*, *SVCXPRT*, and *XDR* types.

void auth_destroy(AUTH *****auth***);**

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy**().

AUTH *authnone_create(void);

Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

AUTH *authunix_create(char ***host*, uid_t** *uid***, gid_t** *gid***,**
int *len***, gid_t** *aup_gids***[.**len**]);**

Create and return an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

AUTH *authunix_create_default(void);

Calls **authunix_create**() with the appropriate parameters.

int callrpc(char ***host*, unsigned long** *prognum***,**
unsigned long *versnum***, unsigned long** *procnum***,**
xdrproc_t *inproc***, const char ***in***,**
xdrproc_t *outproc***, char ***out***);**

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of **enum clnt_stat** cast to an integer if it fails. The routine **clnt_perrno**() is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create**() for restrictions. You do not have control of timeouts or authentication using this routine.

enum clnt_stat clnt_broadcast(unsigned long *prognum***,**
unsigned long *versnum***, unsigned long** *procnum***,**
xdrproc_t *inproc***, char ***in***,**
xdrproc_t *outproc***, char ***out***,**
resultproc_t *eachresult***);**

Like **callrpc**(), except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls **eachresult**(), whose form is:

```
eachresult(char *out, struct sockaddr_in *addr);
```

where *out* is the same as *out* passed to **clnt_broadcast**(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult**() returns zero, **clnt_broadcast**() waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

enum clnt_stat clnt_call(CLIENT ****clnt*, unsigned long** *procnum***,**
xdrproc_t *inproc***, char ****in***,**
xdrproc_t *outproc***, char ****out***,**
struct timeval *tout***);**

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**(). The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

clnt_destroy(CLIENT ****clnt*);**

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**(). If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

CLIENT *clnt_create(const char ****host*, unsigned long** *prog***,**
unsigned long *vers***, const char ****proto***);**

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using **clnt_control**().

Warning: using UDP has its shortcomings. Since UDP-based RPC messages can hold only up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

bool_t clnt_control(CLIENT ****cl*, int** *req***, char ****info***);**

A macro used to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *req* and their argument types and what they do are:

```
CLSET_TIMEOUT  struct timeval // set total timeout
CLGET_TIMEOUT  struct timeval // get total timeout
```

Note: if you set the timeout using **clnt_control**(), the timeout parameter passed to **clnt_call**() will be ignored in all future calls.

```
CLGET_SERVER_ADDR  struct sockaddr_in
                  // get server's address
```

The following operations are valid for UDP only:

```
CLSET_RETRY_TIMEOUT  struct timeval // set the retry timeout
CLGET_RETRY_TIMEOUT  struct timeval // get the retry timeout
```

The retry timeout is the time that "UDP RPC" waits for the server to reply before retransmitting the request.

clnt_freeres(CLIENT * *clnt***, xdrproc_t** *outproc***, char ****out***);**

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns one if the results were successfully freed, and zero

otherwise.

void clnt_geterr(CLIENT ****clnt*, struct rpc_err ****errp***);**

A macro that copies the error structure out of the client handle to the structure at address *errp*.

void clnt_pcreateerror(const char ****s*);**

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. Used when a **clnt_create**(), **clntraw_create**(), **clnttcp_create**(), or **clntudp_create**() call fails.

void clnt_perrno(enum clnt_stat *stat***);**

Print a message to standard error corresponding to the condition indicated by *stat*. Used after **callrpc**().

clnt_perror(CLIENT ****clnt*, const char ****s***);**

Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. Used after **clnt_call**().

char *clnt_spcreateerror(const char ****s*);**

Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each call.

char *clnt_sperrno(enum clnt_stat *stat***);**

Take the same arguments as **clnt_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message. The string ends with a NEWLINE.

clnt_sperrno() is used instead of **clnt_perrno**() if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with *printf*(3), or if a message format different than that supported by **clnt_perrno**() is to be used. Note: unlike **clnt_sperror**() and **clnt_spcreateerror**(), **clnt_sperrno**() returns pointer to static data, but the result will not get overwritten on each call.

char *clnt_sperror(CLIENT ****rpch*, const char ****s***);**

Like **clnt_perror**(), except that (like *clnt_sperrno*()) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

CLIENT *clntraw_create(unsigned long *prognum***, unsigned long** *versnum***);**

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create**(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

CLIENT *clnttcp_create(struct sockaddr_in ****addr*,**
unsigned long *prognum***, unsigned long** *versnum***,**
int ****sockp*, unsigned int** *sendsz***, unsigned int** *recvsz***);**

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address **addr*. If *addr*−>*sin_port* is zero, then it is set to the actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of zero choose suitable defaults. This routine returns NULL if it fails.

CLIENT *clntudp_create(struct sockaddr_in ****addr*,**
unsigned long *prognum***, unsigned long** *versnum***,**

struct timeval *wait***, int ***sockp**);**

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses use UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr−>sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**().

Warning: since UDP-based RPC messages can hold only up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

CLIENT *clntudp_bufcreate(struct sockaddr_in ***_addr_,**
unsigned long *prognum***, unsigned long** *versnum***,**
struct timeval *wait***, int ***_sockp_**,**
unsigned int *sendsize***, unsigned int** *recosize***);**

This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses use UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr−>sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**().

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

void get_myaddress(struct sockaddr_in ***_addr_);**

Stuff the machine's IP address into **addr*, without consulting the library routines that deal with */etc/hosts*. The port number is always set to **htons(PMAPPORT)**.

struct pmaplist *pmap_getmaps(struct sockaddr_in ***_addr_);**

A user interface to the **portmap** service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL. The command *rpcinfo −p* uses this routine.

unsigned short pmap_getport(struct sockaddr_in ***_addr_,**
unsigned long *prognum***, unsigned long** *versnum***,**
unsigned int *protocol***);**

A user interface to the **portmap** service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote **portmap** service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

enum clnt_stat pmap_rmtcall(struct sockaddr_in ***_addr_,**
unsigned long *prognum***, unsigned long** *versnum***,**
unsigned long *procnum***,**
xdrproc_t *inproc***, char ***_in_**,**
xdrproc_t *outproc***, char ***_out_**,**
struct timeval *tout***, unsigned long ***_portp_**);**

A user interface to the **portmap** service, which instructs **portmap** on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc**() and **clnt_call**(). This procedure should be used for a "ping" and nothing else. See also **clnt_broadcast**().

bool_t pmap_set(unsigned long *prognum***, unsigned long** *versnum***,**
int *protocol***, unsigned short** *port***);**

A user interface to the **portmap** service, which establishes a mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the machine's **portmap** service. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. This routine returns one if it succeeds, zero otherwise. Automatically done by **svc_register**().

bool_t pmap_unset(unsigned long *prognum***, unsigned long** *versnum***);**

A user interface to the **portmap** service, which destroys all mapping between the triple [*prognum*,*versnum*,*] and **ports** on the machine's **portmap** service. This routine returns one if it succeeds, zero otherwise.

int registerrpc(unsigned long *prognum***, unsigned long** *versnum***,**
unsigned long *procnum***, char *(*******procname***)(char *),**
xdrproc_t *inproc***, xdrproc_t** *outproc***);**

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *procname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create**() for restrictions.

struct rpc_createerr *rpc_createerr***;**

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror**() to print the reason why.

void svc_destroy(SVCXPRT * *xprt***);**

A macro that destroys the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

fd_set *svc_fdset***;**

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the *select*(2) system call. This is of interest only if a service implementor does their own asynchronous event processing, instead of calling **svc_run**(). This variable is read-only (do not pass its address to *select*(2)!), yet it may change after calls to **svc_getreqset**() or any creation routines.

int *svc_fds***;**

Similar to **svc_fdset**, but limited to 32 file descriptors. This interface is obsoleted by **svc_fdset**.

svc_freeargs(SVCXPRT * *xprt***, xdrproc_t** *inproc***, char *** *in***);**

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**(). This routine returns 1 if the results were successfully freed, and zero otherwise.

svc_getargs(SVCXPRT * *xprt***, xdrproc_t** *inproc***, char *** *in***);**

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

struct sockaddr_in *svc_getcaller(SVCXPRT * *xprt***);**

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

void svc_getreqset(fd_set * *rdfds***);**

This routine is of interest only if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when the *select*(2) system call has determined that an RPC request has arrived on some RPC socket(s); *rdfds* is the resultant

read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

void svc_getreq(int *rdfds***);**

Similar to **svc_getreqset**(), but limited to 32 file descriptors. This interface is obsoleted by **svc_getreqset**().

bool_t svc_register(SVCXPRT * *xprt***, unsigned long** *prognum***,**
unsigned long *versnum***,**
void (* *dispatch***)(struct svc_req *, SVCXPRT *),**
unsigned long *protocol***);**

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the **portmap** service. If *protocol* is nonzero, then a mapping of the triple [*prognum*,*versnum*,*protocol*] to *xprt*−>*xp_port* is established with the local **portmap** service (generally *protocol* is zero, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure *dispatch* has the following form:

```
dispatch(struct svc_req *request, SVCXPRT *xprt);
```

The **svc_register**() routine returns one if it succeeds, and zero otherwise.

void svc_run(void);

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq**() when one arrives. This procedure is usually waiting for a *select*(2) system call to return.

bool_t svc_sendreply(SVCXPRT * *xprt***, xdrproc_t** *outproc***, char *** *out***);**

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

void svc_unregister(unsigned long *prognum***, unsigned long** *versnum***);**

Remove all mapping of the double [*prognum*,*versnum*] to dispatch routines, and of the triple [*prognum*,*versnum*,*] to port number.

void svcerr_auth(SVCXPRT * *xprt***, enum auth_stat** *why***);**

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void svcerr_decode(SVCXPRT * *xprt***);**

Called by a service dispatch routine that cannot successfully decode its parameters. See also **svc_getargs**().

void svcerr_noproc(SVCXPRT * *xprt***);**

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void svcerr_noprog(SVCXPRT * *xprt***);**

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_progvers(SVCXPRT * *xprt***, unsigned long** *low_vers***,**
unsigned long *high_vers***);**

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_systemerr(SVCXPRT * *xprt***);**

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void svcerr_weakauth(SVCXPRT * *xprt***);**

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK)**.

SVCXPRT *svcfd_create(int *fd***, unsigned int** *sendsize***,**
unsigned int *recvsize***);**

Create a service on top of any open file descriptor. Typically, this file descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

SVCXPRT *svcraw_create(void);

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create**(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

SVCXPRT *svctcp_create(int *sock***, unsigned int** *send_buf_size***,**
unsigned int *recv_buf_size***);**

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xprt−>xp_sock* is the transport's socket descriptor, and *xprt−>xp_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

SVCXPRT *svcudp_bufcreate(int *sock***, unsigned int** *sendsize***,**
unsigned int *recosize***);**

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt−>xp_sock* is the transport's socket descriptor, and *xprt−>xp_port* is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

SVCXPRT *svcudp_create(int *sock***);**

This call is equivalent to *svcudp_bufcreate(sock,SZ,SZ)* for some default size *SZ*.

bool_t xdr_accepted_reply(XDR * *xdrs***, struct accepted_reply *** *ar***);**

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_authunix_parms(XDR * *xdrs***, struct authunix_parms *** *aupp***);**

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

void xdr_callhdr(XDR * *xdrs***, struct rpc_msg *** *chdr***);**

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_callmsg(XDR * *xdrs***, struct rpc_msg *** *cmsg***);**

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_opaque_auth(XDR * *xdrs***, struct opaque_auth *** *ap***);**

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_pmap(XDR * *xdrs***, struct pmap *** *regs***);**

Used for describing parameters to various **portmap** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

bool_t xdr_pmaplist(XDR ****xdrs*, struct pmaplist *****rp***);**

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

bool_t xdr_rejected_reply(XDR ****xdrs*, struct rejected_reply ****rr***);**

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t xdr_replymsg(XDR ****xdrs*, struct rpc_msg ****rmsg***);**

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

void xprt_register(SVCXPRT ****xprt*);**

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

void xprt_unregister(SVCXPRT ****xprt*);**

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
auth_destroy(), **authnone_create**(), **authunix_create**(), **authunix_create_default**(), **callrpc**(), **clnt_broadcast**(), **clnt_call**(), **clnt_destroy**(), **clnt_create**(), **clnt_control**(), **clnt_freeres**(), **clnt_geterr**(), **clnt_pcreateerror**(), **clnt_perrno**(), **clnt_perror**(), **clnt_spcreateerror**(), **clnt_sperrno**(), **clnt_sperror**(), **clntraw_create**(), **clnttcp_create**(), **clntudp_create**(), **clntudp_bufcreate**(), **get_myaddress**(), **pmap_getmaps**(), **pmap_getport**(), **pmap_rmtcall**(), **pmap_set**(), **pmap_unset**(), **registerrpc**(), **svc_destroy**(), **svc_freeargs**(), **svc_getargs**(), **svc_getcaller**(), **svc_getreqset**(), **svc_getreq**(), **svc_register**(), **svc_run**(), **svc_sendreply**(), **svc_unregister**(), **svcerr_auth**(), **svcerr_decode**(), **svcerr_noproc**(), **svcerr_noprog**(), **svcerr_progvers**(), **svcerr_systemerr**(), **svcerr_weakauth**(), **svcfd_create**(), **svcraw_create**(), **svctcp_create**(), **svcudp_bufcreate**(), **svcudp_create**(), **xdr_accepted_reply**(), **xdr_authunix_parms**(), **xdr_callhdr**(), **xdr_callmsg**(), **xdr_opaque_auth**(), **xdr_pmap**(), **xdr_pmaplist**(), **xdr_rejected_reply**(), **xdr_replymsg**(), **xprt_register**(), **xprt_unregister**()	Thread safety	MT-Safe

SEE ALSO

xdr(3)

The following manuals:

Remote Procedure Calls: Protocol Specification
Remote Procedure Call Programming Guide
rpcgen Programming Guide

RPC: Remote Procedure Call Protocol Specification, RFC 1050, Sun Microsystems, Inc., USC-ISI.

NAME

rpmatch − determine if the answer to a question is affirmative or negative

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int rpmatch(const char ****response*);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

rpmatch():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_SVID_SOURCE

DESCRIPTION

rpmatch() handles a user response to yes or no questions, with support for internationalization.

response should be a null-terminated string containing a user-supplied response, perhaps obtained with *fgets*(3) or *getline*(3).

The user's language preference is taken into account per the environment variables **LANG**, **LC_MESSAGES**, and **LC_ALL**, if the program has called *setlocale*(3) to effect their changes.

Regardless of the locale, responses matching **^[Yy]** are always accepted as affirmative, and those matching **^[Nn]** are always accepted as negative.

RETURN VALUE

After examining *response*, **rpmatch**() returns 0 for a recognized negative response ("no"), 1 for a recognized positive response ("yes"), and −1 when the value of *response* is unrecognized.

ERRORS

A return value of −1 may indicate either an invalid input, or some other error. It is incorrect to only test if the return value is nonzero.

rpmatch() can fail for any of the reasons that *regcomp*(3) or *regexec*(3) can fail; the cause of the error is not available from *errno* or anywhere else, but indicates a failure of the regex engine (but this case is indistinguishable from that of an unrecognized value of *response*).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rpmatch()	Thread safety	MT-Safe locale

STANDARDS

None.

HISTORY

GNU, FreeBSD, AIX.

BUGS

The **YESEXPR** and **NOEXPR** of some locales (including "C") only inspect the first character of the *response*. This can mean that "yno" et al. resolve to **1**. This is an unfortunate historical side-effect which should be fixed in time with proper localisation, and should not deter from **rpmatch**() being the proper way to distinguish between binary answers.

EXAMPLES

The following program displays the results when **rpmatch**() is applied to the string given in the program's command-line argument.

```
#define _DEFAULT_SOURCE
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0) {
        fprintf(stderr, "%s response\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    setlocale(LC_ALL, "");
    printf("rpmatch() returns: %d\n", rpmatch(argv[1]));
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fgets(3), *getline*(3), *nl_langinfo*(3), *regcomp*(3), *setlocale*(3)

NAME

rtime − get time from a remote machine

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <rpc/auth_des.h>

int rtime(struct sockaddr_in ****addrp*, struct rpc_timeval ****timep***,**
struct rpc_timeval ****timeout*);**

DESCRIPTION

This function uses the Time Server Protocol as described in RFC 868 to obtain the time from a remote machine.

The Time Server Protocol gives the time in seconds since 00:00:00 UTC, 1 Jan 1900, and this function subtracts the appropriate constant in order to convert the result to seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

When *timeout* is non-NULL, the udp/time socket (port 37) is used. Otherwise, the tcp/time socket (port 37) is used.

RETURN VALUE

On success, 0 is returned, and the obtained 32-bit time value is stored in *timep−>tv_sec*. In case of error −1 is returned, and *errno* is set to indicate the error.

ERRORS

All errors for underlying functions (**sendto**(2), *poll*(2), *recvfrom*(2), *connect*(2), *read*(2)) can occur. Moreover:

EIO The number of returned bytes is not 4.

ETIMEDOUT
The waiting time as defined in timeout has expired.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
rtime()	Thread safety	MT-Safe

NOTES

Only IPv4 is supported.

Some *in.timed* versions support only TCP. Try the example program with *use_tcp* set to 1.

BUGS

rtime() in glibc 2.2.5 and earlier does not work properly on 64-bit machines.

EXAMPLES

This example requires that port 37 is up and open. You may check that the time entry within */etc/inetd.conf* is not commented out.

The program connects to a computer called "linux". Using "localhost" does not work. The result is the localtime of the computer "linux".

```
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include <rpc/auth_des.h>

static int use_tcp = 0;
static const char servername[] = "linux";

int
```

```
main(void)
{
    int                 ret;
    time_t              t;
    struct hostent      *hent;
    struct rpc_timeval  time1 = {0, 0};
    struct rpc_timeval  timeout = {1, 0};
    struct sockaddr_in  name;

    memset(&name, 0, sizeof(name));
    sethostent(1);
    hent = gethostbyname(servername);
    memcpy(&name.sin_addr, hent->h_addr, hent->h_length);

    ret = rtime(&name, &time1, use_tcp ? NULL : &timeout);
    if (ret < 0)
        perror("rtime error");
    else {
        t = time1.tv_sec;
        printf("%s\n", ctime(&t));
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

ntpdate(1), *inetd*(8)

NAME

rtnetlink − macros to manipulate rtnetlink messages

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>

rtnetlink_socket = socket(AF_NETLINK, int *socket_type***, NETLINK_ROUTE);**

int RTA_OK(struct rtattr ***_rta_, int** *rtabuflen***);**

void *RTA_DATA(struct rtattr ***_rta_);**
unsigned int RTA_PAYLOAD(struct rtattr ***_rta_);**

struct rtattr *RTA_NEXT(struct rtattr ***_rta_, unsigned int** *rtabuflen***);**

unsigned int RTA_LENGTH(unsigned int *length***);**
unsigned int RTA_SPACE(unsigned int *length***);**

DESCRIPTION

All *rtnetlink*(7) messages consist of a *netlink*(7) message header and appended attributes. The attributes should be manipulated only using the macros provided here.

RTA_OK(*rta***,** *attrlen***)** returns true if *rta* points to a valid routing attribute; *attrlen* is the running length of the attribute buffer. When not true then you must assume there are no more attributes in the message, even if *attrlen* is nonzero.

RTA_DATA(*rta***)** returns a pointer to the start of this attribute's data.

RTA_PAYLOAD(*rta***)** returns the length of this attribute's data.

RTA_NEXT(*rta***,** *attrlen***)** gets the next attribute after *rta*. Calling this macro will update *attrlen*. You should use **RTA_OK** to check the validity of the returned pointer.

RTA_LENGTH(*len***)** returns the length which is required for *len* bytes of data plus the header.

RTA_SPACE(*len***)** returns the amount of space which will be needed in a message with *len* bytes of data.

STANDARDS

Linux.

BUGS

This manual page is incomplete.

EXAMPLES

Creating a rtnetlink message to set the MTU of a device:

```
#include <linux/rtnetlink.h>

...

struct {
    struct nlmsghdr  nh;
    struct ifinfomsg if;
    char             attrbuf[512];
} req;

struct rtattr *rta;
unsigned int mtu = 1000;

int rtnetlink_sk = socket(AF_NETLINK, SOCK_DGRAM, NETLINK_ROUTE);

memset(&req, 0, sizeof(req));
```

```
        req.nh.nlmsg_len = NLMSG_LENGTH(sizeof(req.if));
        req.nh.nlmsg_flags = NLM_F_REQUEST;
        req.nh.nlmsg_type = RTM_NEWLINK;
        req.if.ifi_family = AF_UNSPEC;
        req.if.ifi_index = INTERFACE_INDEX;
        req.if.ifi_change = 0xffffffff; /* ??? */
        rta = (struct rtattr *)(((char *) &req) +
                                NLMSG_ALIGN(req.nh.nlmsg_len));
        rta->rta_type = IFLA_MTU;
        rta->rta_len = RTA_LENGTH(sizeof(mtu));
        req.nh.nlmsg_len = NLMSG_ALIGN(req.nh.nlmsg_len) +
                                       RTA_LENGTH(sizeof(mtu));
        memcpy(RTA_DATA(rta), &mtu, sizeof(mtu));
        send(rtnetlink_sk, &req, req.nh.nlmsg_len, 0);
```

SEE ALSO

netlink(3), *netlink*(7), *rtnetlink*(7)

NAME

scalb, scalbf, scalbl − multiply floating-point number by integral power of radix (OBSOLETE)

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

[[deprecated]] double scalb(double x, double exp);
[[deprecated]] float scalbf(float x, float exp);
[[deprecated]] long double scalbl(long double x, long double exp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

scalb():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

scalbf(), **scalbl**():
_XOPEN_SOURCE >= 600
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions multiply their first argument *x* by **FLT_RADIX** (probably 2) to the power of *exp*, that is:

```
x * FLT_RADIX ** exp
```

The definition of **FLT_RADIX** can be obtained by including <*float.h*>.

RETURN VALUE

On success, these functions return *x* * **FLT_RADIX** ** *exp*.

If *x* or *exp* is a NaN, a NaN is returned.

If *x* is positive infinity (negative infinity), and *exp* is not negative infinity, positive infinity (negative infinity) is returned.

If *x* is +0 (−0), and *exp* is not positive infinity, +0 (−0) is returned.

If *x* is zero, and *exp* is positive infinity, a domain error occurs, and a NaN is returned.

If *x* is an infinity, and *exp* is negative infinity, a domain error occurs, and a NaN is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with a sign the same as *x*.

If the result underflows, a range error occurs, and the functions return zero, with a sign the same as *x*.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is 0, and *exp* is positive infinity, or *x* is positive infinity and *exp* is negative infinity and the other argument is not a NaN
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

Range error, overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error, underflow
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
scalb(), **scalbf**(), **scalbl**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

scalb() 4.3BSD. Obsolescent in POSIX.1-2001; Removed in POSIX.1-2008, recommending the use of *scalbln*(3), *scalblnf*(3), or *scalblnl*(3) instead.

BUGS

Before glibc 2.20, these functions did not set *errno* for domain and range errors.

SEE ALSO

ldexp(3), *scalbln*(3)

NAME

scalbn, scalbnf, scalbnl, scalbln, scalblnf, scalblnl − multiply floating-point number by integral power of radix

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double scalbln(double x, long exp);
float scalblnf(float x, long exp);
long double scalblnl(long double x, long exp);

double scalbn(double x, int exp);
float scalbnf(float x, int exp);
long double scalbnl(long double x, int exp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

scalbln(), **scalblnf**(), **scalblnl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE

scalbn(), **scalbnf**(), **scalbnl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions multiply their first argument *x* by **FLT_RADIX** (probably 2) to the power of *exp*, that is:

```
x * FLT_RADIX ** exp
```

The definition of **FLT_RADIX** can be obtained by including <*float.h*>.

RETURN VALUE

On success, these functions return *x* * **FLT_RADIX** ** *exp*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity (negative infinity), positive infinity (negative infinity) is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with a sign the same as *x*.

If the result underflows, a range error occurs, and the functions return zero, with a sign the same as *x*.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error, overflow
: An overflow floating-point exception (**FE_OVERFLOW**) is raised.

Range error, underflow
: *errno* is set to **ERANGE**. An underflow floating-point exception (**FE_UNDERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
scalbn(), **scalbnf**(), **scalbnl**(), **scalbln**(), **scalblnf**(), **scalblnl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

HISTORY

These functions differ from the obsolete functions described in *scalb*(3) in the type of their second argument. The functions described on this page have a second argument of an integral type, while those in *scalb*(3) have a second argument of type *double*.

NOTES

If **FLT_RADIX** equals 2 (which is usual), then **scalbn**() is equivalent to *ldexp*(3).

BUGS

Before glibc 2.20, these functions did not set *errno* for range errors.

SEE ALSO

ldexp(3), *scalb*(3)

NAME

scandir, scandirat, alphasort, versionsort − scan a directory for matching entries

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <dirent.h>

int scandir(const char *restrict dirp,
        struct dirent ***restrict namelist,
        int (* filter)(const struct dirent *),
        int (*compar)(const struct dirent **,
                const struct dirent **));

int alphasort(const struct dirent **a, const struct dirent **b);
int versionsort(const struct dirent **a, const struct dirent **b);

#include <fcntl.h>          /* Definition of AT_* constants */
#include <dirent.h>

int scandirat(int dirfd, const char *restrict dirp,
        struct dirent ***restrict namelist,
        int (* filter)(const struct dirent *),
        int (*compar)(const struct dirent **,
                const struct dirent **));
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

scandir(), **alphasort**():

```
/* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200809L
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

versionsort():

```
_GNU_SOURCE
```

scandirat():

```
_GNU_SOURCE
```

DESCRIPTION

The **scandir**() function scans the directory *dirp*, calling *filter*() on each directory entry. Entries for which *filter*() returns nonzero are stored in strings allocated via *malloc*(3), sorted using *qsort*(3) with the comparison function *compar*(), and collected in array *namelist* which is allocated via *malloc*(3). If *filter* is NULL, all entries are selected.

The **alphasort**() and **versionsort**() functions can be used as the comparison function *compar*(). The former sorts directory entries using *strcoll*(3), the latter using *strverscmp*(3) on the strings *(*a)−>d_name* and *(*b)−>d_name*.

scandirat()

The **scandirat**() function operates in exactly the same way as **scandir**(), except for the differences described here.

If the pathname given in *dirp* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **scandir**() for a relative pathname).

If *dirp* is relative and *dirfd* is the special value **AT_FDCWD**, then *dirp* is interpreted relative to the current working directory of the calling process (like *scandir*())

If *dirp* is absolute, then *dirfd* is ignored.

See *openat*(2) for an explanation of the need for **scandirat**().

RETURN VALUE

The **scandir**() function returns the number of directory entries selected. On error, −1 is returned, with *errno* set to indicate the error.

The **alphasort**() and **versionsort**() functions return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

ERRORS

EBADF
(**scandirat**()) *dirp* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

ENOENT
The path in *dirp* does not exist.

ENOMEM
Insufficient memory to complete the operation.

ENOTDIR
The path in *dirp* is not a directory.

ENOTDIR
(**scandirat**()) *dirp* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
scandir(), **scandirat**()	Thread safety	MT-Safe
alphasort(), **versionsort**()	Thread safety	MT-Safe locale

STANDARDS

alphasort()
scandir()
POSIX.1-2008.

versionsort()
scandirat()
GNU.

HISTORY

alphasort()
scandir()
4.3BSD, POSIX.1-2008.

versionsort()
glibc 2.1.

scandirat()
glibc 2.15.

NOTES

Since glibc 2.1, **alphasort**() calls *strcoll*(3); earlier it used *strcmp*(3).

Before glibc 2.10, the two arguments of **alphasort**() and **versionsort**() were typed as *const void* *. When **alphasort**() was standardized in POSIX.1-2008, the argument type was specified as the type-safe *const struct dirent* **, and glibc 2.10 changed the definition of **alphasort**() (and the nonstandard *versionsort*()) to match the standard.

EXAMPLES

The program below prints a list of the files in the current directory in reverse order.

Program source

```
#define _DEFAULT_SOURCE
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    struct dirent **namelist;
    int n;
```

```
        n = scandir(".", &namelist, NULL, alphasort);
        if (n == -1) {
            perror("scandir");
            exit(EXIT_FAILURE);
        }

        while (n--) {
            printf("%s\n", namelist[n]->d_name);
            free(namelist[n]);
        }
        free(namelist);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

closedir(3), *fnmatch*(3), *opendir*(3), *readdir*(3), *rewinddir*(3), *seekdir*(3), *strcmp*(3), *strcoll*(3), *strverscmp*(3), *telldir*(3)

NAME

scanf, fscanf, vscanf, vfscanf − input FILE format conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *restrict format, ...);
int fscanf(FILE *restrict stream,
        const char *restrict format, ...);

#include <stdarg.h>

int vscanf(const char *restrict format, va_list ap);
int vfscanf(FILE *restrict stream,
        const char *restrict format, va_list ap);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

vscanf(), **vfscanf**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **scanf**() family of functions scans formatted input like *sscanf*(3), but read from a *FILE*. It is very difficult to use these functions correctly, and it is preferable to read entire lines with *fgets*(3) or *getline*(3) and parse them later with *sscanf*(3) or more specialized functions such as *strtol*(3).

The **scanf**() function reads input from the standard input stream *stdin* and **fscanf**() reads input from the stream pointer *stream*.

The **vfscanf**() function is analogous to *vfprintf*(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see *stdarg*(3). The **vscanf**() function is analogous to *vprintf*(3) and reads from the standard input.

RETURN VALUE

On success, these functions return the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure.

The value **EOF** is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. **EOF** is also returned if a read error occurs, in which case the error indicator for the stream (see *ferror*(3)) is set, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: The file descriptor underlying *stream* is marked nonblocking, and the read operation would block.

EBADF
: The file descriptor underlying *stream* is invalid, or not open for reading.

EILSEQ
: Input byte sequence does not form a valid character.

EINTR
: The read operation was interrupted by a signal; see *signal*(7).

EINVAL
: Not enough arguments; or *format* is NULL.

ENOMEM
: Out of memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
scanf(), **fscanf**(), **vscanf**(), **vfscanf**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

CAVEATS

These functions make it difficult to distinguish newlines from other white space, This is especially problematic with line-buffered input, like the standard input stream.

These functions can’t report errors after the last non-suppressed conversion specification.

BUGS

It is impossible to accurately know how many characters these functions have consumed from the input stream, since they only report the number of successful conversions. For example, if the input is "123\n a", *scanf("%d %d", &a, &b)* will consume the digits, the newline, and the space, but not the letter a. This makes it difficult to recover from invalid input.

SEE ALSO

fgets(3), *getline*(3), *sscanf*(3)

NAME

sched_getcpu − determine CPU on which the calling thread is running

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sched.h>

int sched_getcpu(void);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sched_getcpu():
Since glibc 2.14:
_GNU_SOURCE
Before glibc 2.14:
_BSD_SOURCE || _SVID_SOURCE
/* _GNU_SOURCE also suffices */

DESCRIPTION

sched_getcpu() returns the number of the CPU on which the calling thread is currently executing.

RETURN VALUE

On success, **sched_getcpu**() returns a nonnegative CPU number. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

ENOSYS
This kernel does not implement *getcpu*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sched_getcpu()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.6.

NOTES

The call

```
cpu = sched_getcpu();
```

is equivalent to the following *getcpu*(2) call:

```
int c, s;
s = getcpu(&c, NULL);
cpu = (s == -1) ? s : c;
```

SEE ALSO

getcpu(2), *sched*(7)

NAME

seekdir − set the position of the next readdir() call in the directory stream.

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <dirent.h>

void seekdir(DIR *****dirp*, long** *loc***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

seekdir():
_XOPEN_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **seekdir**() function sets the location in the directory stream from which the next *readdir*(2) call will start. The *loc* argument should be a value returned by a previous call to *telldir*(3).

RETURN VALUE

The **seekdir**() function returns no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
seekdir()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

CAVEATS

Up to glibc 2.1.1, the type of the *loc* argument was *off_t*. POSIX.1-2001 specifies *long*, and this is the type used since glibc 2.1.2. See *telldir*(3) for information on why you should be careful in making any assumptions about the value in this argument.

SEE ALSO

lseek(2), *closedir*(3), *opendir*(3), *readdir*(3), *rewinddir*(3), *scandir*(3), *telldir*(3)

NAME

sem_close − close a named semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_close(sem_t ****sem*);**

DESCRIPTION

sem_close() closes the named semaphore referred to by *sem*, allowing any resources that the system has allocated to the calling process for this semaphore to be freed.

RETURN VALUE

On success **sem_close**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EINVAL

sem is not a valid semaphore.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_close()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

All open named semaphores are automatically closed on process termination, or upon *execve*(2).

SEE ALSO

sem_getvalue(3), *sem_open*(3), *sem_post*(3), *sem_unlink*(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_destroy − destroy an unnamed semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_destroy(sem_t *_sem_);

DESCRIPTION

sem_destroy() destroys the unnamed semaphore at the address pointed to by *sem*.

Only a semaphore that has been initialized by *sem_init*(3) should be destroyed using **sem_destroy**().

Destroying a semaphore that other processes or threads are currently blocked on (in *sem_wait*(3)) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using *sem_init*(3).

RETURN VALUE

sem_destroy() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL

sem is not a valid semaphore.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_destroy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

An unnamed semaphore should be destroyed with **sem_destroy**() before the memory in which it is located is deallocated. Failure to do this can result in resource leaks on some implementations.

SEE ALSO

sem_init(3), *sem_post*(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_getvalue − get the value of a semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_getvalue(sem_t *restrict *sem***, int *restrict** *sval***);**

DESCRIPTION

sem_getvalue() places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*.

If one or more processes or threads are blocked waiting to lock the semaphore with *sem_wait*(3), POSIX.1 permits two possibilities for the value returned in *sval*: either 0 is returned; or a negative number whose absolute value is the count of the number of processes and threads currently blocked in *sem_wait*(3). Linux adopts the former behavior.

RETURN VALUE

sem_getvalue() returns 0 on success; on error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

sem is not a valid semaphore. (The glibc implementation currently does not check whether *sem* is valid.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_getvalue()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The value of the semaphore may already have changed by the time **sem_getvalue**() returns.

SEE ALSO

sem_post(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_init − initialize an unnamed semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_init(sem_t ****sem*, int** *pshared***, unsigned int** *value***);**

DESCRIPTION

sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

If *pshared* is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory (see *shm_open*(3), *mmap*(2), and *shmget*(2)). (Since a child created by *fork*(2) inherits its parent's memory mappings, it can also access the semaphore.) Any process that can access the shared memory region can operate on the semaphore using *sem_post*(3), *sem_wait*(3), and so on.

Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE

sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL

value exceeds **SEM_VALUE_MAX**.

ENOSYS

pshared is nonzero, but the system does not support process-shared semaphores (see *sem_overview*(7)).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_init()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Bizarrely, POSIX.1-2001 does not specify the value that should be returned by a successful call to **sem_init**(). POSIX.1-2008 rectifies this, specifying the zero return on success.

EXAMPLES

See *shm_open*(3) and *sem_wait*(3).

SEE ALSO

sem_destroy(3), *sem_post*(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_open − initialize and open a named semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>       /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

DESCRIPTION

sem_open() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by *name*. For details of the construction of *name*, see *sem_overview*(7).

The *oflag* argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including <*fcntl.h*>.) If **O_CREAT** is specified in *oflag*, then the semaphore is created if it does not already exist. The owner (user ID) of the semaphore is set to the effective user ID of the calling process. The group ownership (group ID) is set to the effective group ID of the calling process. If both **O_CREAT** and **O_EXCL** are specified in *oflag*, then an error is returned if a semaphore with the given *name* already exists.

If **O_CREAT** is specified in *oflag*, then two additional arguments must be supplied. The *mode* argument specifies the permissions to be placed on the new semaphore, as for *open*(2). (Symbolic definitions for the permissions bits can be obtained by including <*sys/stat.h*>.) The permissions settings are masked against the process umask. Both read and write permission should be granted to each class of user that will access the semaphore. The *value* argument specifies the initial value for the new semaphore. If **O_CREAT** is specified, and a semaphore with the given *name* already exists, then *mode* and *value* are ignored.

RETURN VALUE

On success, **sem_open**() returns the address of the new semaphore; this address is used when calling other semaphore-related functions. On error, **sem_open**() returns **SEM_FAILED**, with *errno* set to indicate the error.

ERRORS

EACCES
: The semaphore exists, but the caller does not have permission to open it.

EEXIST
: Both **O_CREAT** and **O_EXCL** were specified in *oflag*, but a semaphore with this *name* already exists.

EINVAL
: *value* was greater than **SEM_VALUE_MAX**.

EINVAL
: *name* consists of just "/", followed by no other characters.

EMFILE
: The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG
: *name* was too long.

ENFILE
: The system-wide limit on the total number of open files has been reached.

ENOENT
: The **O_CREAT** flag was not specified in *oflag* and no semaphore with this *name* exists; or, **O_CREAT** was specified, but *name* wasn't well formed.

ENOMEM
Insufficient memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_open()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

sem_close(3), *sem_getvalue*(3), *sem_post*(3), *sem_unlink*(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_post − unlock a semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_post(sem_t *****sem*);**

DESCRIPTION

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a *sem_wait*(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE

sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL

sem is not a valid semaphore.

EOVERFLOW

The maximum allowable value for a semaphore would be exceeded.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_post()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

sem_post() is async-signal-safe: it may be safely called within a signal handler.

EXAMPLES

See *sem_wait*(3) and *shm_open*(3).

SEE ALSO

sem_getvalue(3), *sem_wait*(3), *sem_overview*(7), *signal-safety*(7)

NAME

sem_unlink − remove a named semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

#include <semaphore.h>

int sem_unlink(const char *_name_);

DESCRIPTION

sem_unlink() removes the named semaphore referred to by *name*. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

RETURN VALUE

On success **sem_unlink**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EACCES
The caller does not have permission to unlink this semaphore.

ENAMETOOLONG
name was too long.

ENOENT
There is no semaphore with the given *name*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_unlink()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

sem_getvalue(3), *sem_open*(3), *sem_post*(3), *sem_wait*(3), *sem_overview*(7)

NAME

sem_wait, sem_timedwait, sem_trywait − lock a semaphore

LIBRARY

POSIX threads library (*libpthread*, *−lpthread*)

SYNOPSIS

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict abs_timeout);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sem_timedwait():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

sem_timedwait() is the same as **sem_wait**(), except that *abs_timeout* specifies a limit on the amount of time that the call should block if the decrement cannot be immediately performed. The *abs_timeout* argument points to a *timespec*(3) structure that specifies an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

If the timeout has already expired by the time of the call, and the semaphore could not be locked immediately, then **sem_timedwait**() fails with a timeout error (*errno* set to **ETIMEDOUT**).

If the operation can be performed immediately, then **sem_timedwait**() never fails with a timeout error, regardless of the value of *abs_timeout*. Furthermore, the validity of *abs_timeout* is not checked in this case.

RETURN VALUE

All of these functions return 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
(**sem_trywait**()) The operation could not be performed without blocking (i.e., the semaphore currently has the value zero).

EINTR
The call was interrupted by a signal handler; see *signal*(7).

EINVAL
sem is not a valid semaphore.

EINVAL
(**sem_timedwait**()) The value of *abs_timeout.tv_nsecs* is less than 0, or greater than or equal to 1000 million.

ETIMEDOUT
(**sem_timedwait**()) The call timed out before the semaphore could be locked.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sem_wait(), **sem_trywait**(), **sem_timedwait**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The (somewhat trivial) program shown below operates on an unnamed semaphore. The program expects two command-line arguments. The first argument specifies a seconds value that is used to set an alarm timer to generate a **SIGALRM** signal. This handler performs a *sem_post*(3) to increment the semaphore that is being waited on in *main()* using **sem_timedwait**(). The second command-line argument specifies the length of the timeout, in seconds, for **sem_timedwait**(). The following shows what happens on two different runs of the program:

```
$ ./a.out 2 3
About to call sem_timedwait()
sem_post() from handler
sem_timedwait() succeeded
$ ./a.out 2 1
About to call sem_timedwait()
sem_timedwait() timed out
```

Program source

```
#include <errno.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include <assert.h>

sem_t sem;

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void
handler(int sig)
{
    write(STDOUT_FILENO, "sem_post() from handler\n", 24);
    if (sem_post(&sem) == -1) {
        write(STDERR_FILENO, "sem_post() failed\n", 18);
        _exit(EXIT_FAILURE);
    }
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    struct timespec ts;
    int s;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <alarm-secs> <wait-secs>\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
        if (sem_init(&sem, 0, 0) == -1)
            handle_error("sem_init");

        /* Establish SIGALRM handler; set alarm timer using argv[1]. */

        sa.sa_handler = handler;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        if (sigaction(SIGALRM, &sa, NULL) == -1)
            handle_error("sigaction");

        alarm(atoi(argv[1]));

        /* Calculate relative interval as current time plus
           number of seconds given argv[2]. */

        if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
            handle_error("clock_gettime");

        ts.tv_sec += atoi(argv[2]);

        printf("%s() about to call sem_timedwait()\n", __func__);
        while ((s = sem_timedwait(&sem, &ts)) == -1 && errno == EINTR)
            continue;       /* Restart if interrupted by handler. */

        /* Check what happened. */

        if (s == -1) {
            if (errno == ETIMEDOUT)
                printf("sem_timedwait() timed out\n");
            else
                perror("sem_timedwait");
        } else
            printf("sem_timedwait() succeeded\n");

        exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
    }
```

SEE ALSO

clock_gettime(2), *sem_getvalue*(3), *sem_post*(3), *timespec*(3), *sem_overview*(7), *time*(7)

NAME

setaliasent, endaliasent, getaliasent, getaliasent_r, getaliasbyname, getaliasbyname_r − read an alias entry

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <aliases.h>

void setaliasent(void);
void endaliasent(void);

struct aliasent *getaliasent(void);
int getaliasent_r(struct aliasent *restrict result,
                  char buffer[restrict .buflen], size_t buflen,
                  struct aliasent **restrict res);

struct aliasent *getaliasbyname(const char *name);
int getaliasbyname_r(const char *restrict name,
                  struct aliasent *restrict result,
                  char buffer[restrict .buflen], size_t buflen,
                  struct aliasent **restrict res);
```

DESCRIPTION

One of the databases available with the Name Service Switch (NSS) is the aliases database, that contains mail aliases. (To find out which databases are supported, try *getent −−help*.) Six functions are provided to access the aliases database.

The **getaliasent**() function returns a pointer to a structure containing the group information from the aliases database. The first time it is called it returns the first entry; thereafter, it returns successive entries.

The **setaliasent**() function rewinds the file pointer to the beginning of the aliases database.

The **endaliasent**() function closes the aliases database.

getaliasent_r() is the reentrant version of the previous function. The requested structure is stored via the first argument but the programmer needs to fill the other arguments also. Not providing enough space causes the function to fail.

The function **getaliasbyname**() takes the name argument and searches the aliases database. The entry is returned as a pointer to a *struct aliasent*.

getaliasbyname_r() is the reentrant version of the previous function. The requested structure is stored via the second argument but the programmer needs to fill the other arguments also. Not providing enough space causes the function to fail.

The *struct aliasent* is defined in <*aliases.h*>:

```
struct aliasent {
    char    *alias_name;             /* alias name */
    size_t   alias_members_len;
    char   **alias_members;          /* alias name list */
    int      alias_local;
};
```

RETURN VALUE

The functions **getaliasent_r**() and **getaliasbyname_r**() return a nonzero value on error.

FILES

The default alias database is the file */etc/aliases*. This can be changed in the */etc/nsswitch.conf* file.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setaliasent(), **endaliasent**(), **getaliasent_r**(), **getaliasbyname_r**()	Thread safety	MT-Safe locale
getaliasent(), **getaliasbyname**()	Thread safety	MT-Unsafe

STANDARDS

GNU.

HISTORY

The NeXT system has similar routines:

```
#include <aliasdb.h>

void alias_setent(void);
void alias_endent(void);
alias_ent *alias_getent(void);
alias_ent *alias_getbyname(char *name);
```

EXAMPLES

The following example compiles with *gcc example.c −o example*. It will dump all names in the alias database.

```
#include <aliases.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    struct aliasent *al;

    setaliasent();
    for (;;) {
        al = getaliasent();
        if (al == NULL)
            break;
        printf("Name: %s\n", al->alias_name);
    }
    if (errno) {
        perror("reading alias");
        exit(EXIT_FAILURE);
    }
    endaliasent();
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

getgrent(3), *getpwent*(3), *getspent*(3), *aliases*(5)

NAME

setbuf, setbuffer, setlinebuf, setvbuf − stream buffering operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int setvbuf(FILE *restrict stream, char buf[restrict .size],
        int mode, size_t size);

void setbuf(FILE *restrict stream, char *restrict buf);
void setbuffer(FILE *restrict stream, char buf[restrict .size],
        size_t size);
void setlinebuf(FILE *stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setbuffer(), **setlinebuf**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered, many characters are saved up and written as a block; when it is line buffered, characters are saved up until a newline is output or input is read from any stream attached to a terminal device (typically *stdin*). The function *fflush*(3) may be used to force the block out early. (See *fclose*(3).)

Normally all files are block buffered. If a stream refers to a terminal (as *stdout* normally does), it is line buffered. The standard error stream *stderr* is always unbuffered by default.

The **setvbuf**() function may be used on any open stream to change its buffer. The *mode* argument must be one of the following three macros:

_IONBF
unbuffered

_IOLBF
line buffered

_IOFBF
fully buffered

Except for unbuffered files, the *buf* argument should point to a buffer at least *size* bytes long; this buffer will be used instead of the current buffer. If the argument *buf* is NULL, only the mode is affected; a new buffer will be allocated on the next read or write operation. The **setvbuf**() function may be used only after opening a stream and before any other operations have been performed on it.

The other three calls are, in effect, simply aliases for calls to **setvbuf**(). The **setbuf**() function is exactly equivalent to the call

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

The **setbuffer**() function is the same, except that the size of the buffer is up to the caller, rather than being determined by the default **BUFSIZ**. The **setlinebuf**() function is exactly equivalent to the call:

```
setvbuf(stream, NULL, _IOLBF, 0);
```

RETURN VALUE

The function **setvbuf**() returns 0 on success. It returns nonzero on failure (*mode* is invalid or the request cannot be honored). It may set *errno* on failure.

The other functions do not return a value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setbuf(), **setbuffer**(), **setlinebuf**(), **setvbuf**()	Thread safety	MT-Safe

STANDARDS

setbuf()
setvbuf()
C11, POSIX.1-2008.

HISTORY

setbuf()
setvbuf()
C89, POSIX.1-2001.

CAVEATS

POSIX notes that the value of *errno* is unspecified after a call to **setbuf**() and further notes that, since the value of *errno* is not required to be unchanged after a successful call to **setbuf**(), applications should instead use **setvbuf**() in order to detect errors.

BUGS

You must make sure that the space that *buf* points to still exists by the time *stream* is closed, which also happens at program termination. For example, the following is invalid:

```
#include <stdio.h>

int
main(void)
{
    char buf[BUFSIZ];

    setbuf(stdout, buf);
    printf("Hello, world!\n");
    return 0;
}
```

SEE ALSO

stdbuf (1), *fclose*(3), *fflush*(3), *fopen*(3), *fread*(3), *malloc*(3), *printf*(3), *puts*(3)

NAME

setenv − change or add an environment variable

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setenv(), **unsetenv**():

```
_POSIX_C_SOURCE >= 200112L
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION

The **setenv**() function adds the variable *name* to the environment with the value *value*, if *name* does not already exist. If *name* does exist in the environment, then its value is changed to *value* if *overwrite* is nonzero; if *overwrite* is zero, then the value of *name* is not changed (and **setenv**() returns a success status). This function makes copies of the strings pointed to by *name* and *value* (by contrast with *putenv*(3)).

The **unsetenv**() function deletes the variable *name* from the environment. If *name* does not exist in the environment, then the function succeeds, and the environment is unchanged.

RETURN VALUE

setenv() and **unsetenv**() functions return zero on success, or −1 on error, with *errno* set to indicate the error.

ERRORS

EINVAL
: *name* is NULL, points to a string of length 0, or contains an '=' character.

ENOMEM
: Insufficient memory to add a new variable to the environment.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setenv(), **unsetenv**()	Thread safety	MT-Unsafe const:env

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

Prior to glibc 2.2.2, **unsetenv**() was prototyped as returning *void*; more recent glibc versions follow the POSIX.1-compliant prototype shown in the SYNOPSIS.

CAVEATS

POSIX.1 does not require **setenv**() or **unsetenv**() to be reentrant.

BUGS

POSIX.1 specifies that if *name* contains an '=' character, then **setenv**() should fail with the error **EINVAL**; however, versions of glibc before glibc 2.3.4 allowed an '=' sign in *name*.

SEE ALSO

clearenv(3), *getenv*(3), *putenv*(3), *environ*(7)

NAME

__setfpucw − set FPU control word on i386 architecture (obsolete)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <i386/fpu_control.h>

[[deprecated]] void __setfpucw(unsigned short *control_word***);**

DESCRIPTION

__setfpucw() transfers *control_word* to the registers of the FPU (floating-point unit) on the i386 architecture. This was used to control floating-point precision, rounding and floating-point exceptions.

STANDARDS

GNU.

HISTORY

Removed in glibc 2.1.

NOTES

There are new functions from C99, with prototypes in <*fenv.h*>, to control FPU rounding modes, like *fegetround*(3), *fesetround*(3), and the floating-point environment, like *fegetenv*(3), *feholdexcept*(3), *fesetenv*(3), *feupdateenv*(3), and FPU exception handling, like *feclearexcept*(3), *fegetexceptflag*(3), *feraiseexcept*(3), *fesetexceptflag*(3), and *fetestexcept*(3).

If direct access to the FPU control word is still needed, the **_FPU_GETCW** and **_FPU_SETCW** macros from <*fpu_control.h*> can be used.

EXAMPLES

__setfpucw(0x1372)

Set FPU control word on the i386 architecture to

- extended precision
- rounding to nearest
- exceptions on overflow, zero divide and NaN

SEE ALSO

feclearexcept(3)

<*fpu_control.h*>

NAME

setjmp, sigsetjmp, longjmp, siglongjmp − performing a nonlocal goto

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);

[[noreturn]] void longjmp(jmp_buf env, int val);
[[noreturn]] void siglongjmp(sigjmp_buf env, int val);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setjmp(): see NOTES.

sigsetjmp():
_POSIX_C_SOURCE

DESCRIPTION

The functions described on this page are used for performing "nonlocal gotos": transferring execution from one function to a predetermined location in another function. The **setjmp**() function dynamically establishes the target to which control will later be transferred, and **longjmp**() performs the transfer of execution.

The **setjmp**() function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer *env* for later use by **longjmp**(). In this case, **setjmp**() returns 0.

The **longjmp**() function uses the information saved in *env* to transfer control back to the point where **setjmp**() was called and to restore ("rewind") the stack to its state at the time of the **setjmp**() call. In addition, and depending on the implementation (see NOTES), the values of some other registers and the process signal mask may be restored to their state at the time of the **setjmp**() call.

Following a successful **longjmp**(), execution continues as if **setjmp**() had returned for a second time. This "fake" return can be distinguished from a true **setjmp**() call because the "fake" return returns the value provided in *val*. If the programmer mistakenly passes the value 0 in *val*, the "fake" return will instead return 1.

sigsetjmp() and siglongjmp()

sigsetjmp() and **siglongjmp**() also perform nonlocal gotos, but provide predictable handling of the process signal mask.

If, and only if, the *savesigs* argument provided to **sigsetjmp**() is nonzero, the process's current signal mask is saved in *env* and will be restored if a **siglongjmp**() is later performed with this *env*.

RETURN VALUE

setjmp() and **sigsetjmp**() return 0 when called directly; on the "fake" return that occurs after **longjmp**() or **siglongjmp**(), the nonzero value specified in *val* is returned.

The **longjmp**() or **siglongjmp**() functions do not return.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setjmp(), **sigsetjmp**()	Thread safety	MT-Safe
longjmp(), **siglongjmp**()	Thread safety	MT-Safe

STANDARDS

setjmp()
longjmp()
C11, POSIX.1-2008.

sigsetjmp()

siglongjmp()
POSIX.1-2008.

HISTORY

setjmp()
longjmp()
POSIX.1-2001, C89.

sigsetjmp()
siglongjmp()
POSIX.1-2001.

POSIX does not specify whether **setjmp**() will save the signal mask (to be later restored during *longjmp*())In System V it will not. In 4.3BSD it will, and there is a function **_setjmp**() that will not. The behavior under Linux depends on the glibc version and the setting of feature test macros. Before glibc 2.19, **setjmp**() follows the System V behavior by default, but the BSD behavior is provided if the **_BSD_SOURCE** feature test macro is explicitly defined and none of **_POSIX_SOURCE**, **_POSIX_C_SOURCE**, **_XOPEN_SOURCE**, **_GNU_SOURCE**, or **_SVID_SOURCE** is defined. Since glibc 2.19, <*setjmp.h*> exposes only the System V version of **setjmp**(). Programs that need the BSD semantics should replace calls to **setjmp**() with calls to **sigsetjmp**() with a nonzero *savesigs* argument.

NOTES

setjmp() and **longjmp**() can be useful for dealing with errors inside deeply nested function calls or to allow a signal handler to pass control to a specific point in the program, rather than returning to the point where the handler interrupted the main program. In the latter case, if you want to portably save and restore signal masks, use **sigsetjmp**() and **siglongjmp**(). See also the discussion of program readability below.

CAVEATS

The compiler may optimize variables into registers, and **longjmp**() may restore the values of other registers in addition to the stack pointer and program counter. Consequently, the values of automatic variables are unspecified after a call to **longjmp**() if they meet all the following criteria:

- they are local to the function that made the corresponding **setjmp**() call;
- their values are changed between the calls to **setjmp**() and **longjmp**(); and
- they are not declared as *volatile*.

Analogous remarks apply for **siglongjmp**().

Nonlocal gotos and program readability

While it can be abused, the traditional C "goto" statement at least has the benefit that lexical cues (the goto statement and the target label) allow the programmer to easily perceive the flow of control. Nonlocal gotos provide no such cues: multiple **setjmp**() calls might employ the same *jmp_buf* variable so that the content of the variable may change over the lifetime of the application. Consequently, the programmer may be forced to perform detailed reading of the code to determine the dynamic target of a particular **longjmp**() call. (To make the programmer's life easier, each **setjmp**() call should employ a unique *jmp_buf* variable.)

Adding further difficulty, the **setjmp**() and **longjmp**() calls may not even be in the same source code module.

In summary, nonlocal gotos can make programs harder to understand and maintain, and an alternative should be used if possible.

Undefined Behavior

If the function which called **setjmp**() returns before **longjmp**() is called, the behavior is undefined. Some kind of subtle or unsubtle chaos is sure to result.

If, in a multithreaded program, a **longjmp**() call employs an *env* buffer that was initialized by a call to **setjmp**() in a different thread, the behavior is undefined.

POSIX.1-2008 Technical Corrigendum 2 adds **longjmp**() and **siglongjmp**() to the list of async-signal-safe functions. However, the standard recommends avoiding the use of these functions from signal handlers and goes on to point out that if these functions are called from a signal handler that interrupted a call to a non-async-signal-safe function (or some equivalent, such as the steps equivalent to *exit*(3)

that occur upon a return from the initial call to *main*()), the behavior is undefined if the program subsequently makes a call to a non-async-signal-safe function. The only way of avoiding undefined behavior is to ensure one of the following:

- After long jumping from the signal handler, the program does not call any non-async-signal-safe functions and does not return from the initial call to *main*().
- Any signal whose handler performs a long jump must be blocked during *every* call to a non-async-signal-safe function and no non-async-signal-safe functions are called after returning from the initial call to *main*().

SEE ALSO

signal(7), *signal-safety*(7)

NAME

setlocale − set the current locale

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <locale.h>

char *setlocale(int *category***, const char *** *locale***);**

DESCRIPTION

The **setlocale**() function is used to set or query the program's current locale.

If *locale* is not NULL, the program's current locale is modified according to the arguments. The argument *category* determines which parts of the program's current locale should be modified.

Category	Governs
LC_ALL	All of the locale
LC_ADDRESS	Formatting of addresses and geography-related items (*)
LC_COLLATE	String collation
LC_CTYPE	Character classification
LC_IDENTIFICATION	Metadata describing the locale (*)
LC_MEASUREMENT	Settings related to measurements (metric versus US customary) (*)
LC_MESSAGES	Localizable natural-language messages
LC_MONETARY	Formatting of monetary values
LC_NAME	Formatting of salutations for persons (*)
LC_NUMERIC	Formatting of nonmonetary numeric values
LC_PAPER	Settings related to the standard paper size (*)
LC_TELEPHONE	Formats to be used with telephone services (*)
LC_TIME	Formatting of date and time values

The categories marked with an asterisk in the above table are GNU extensions. For further information on these locale categories, see *locale*(7).

The argument *locale* is a pointer to a character string containing the required setting of *category*. Such a string is either a well-known constant like "C" or "da_DK" (see below), or an opaque string that was returned by another call of **setlocale**().

If *locale* is an empty string, **""**, each part of the locale that should be modified is set according to the environment variables. The details are implementation-dependent. For glibc, first (regardless of *category*), the environment variable **LC_ALL** is inspected, next the environment variable with the same name as the category (see the table above), and finally the environment variable **LANG**. The first existing environment variable is used. If its value is not a valid locale specification, the locale is unchanged, and **setlocale**() returns NULL.

The locale **"C"** or **"POSIX"** is a portable locale; it exists on all conforming systems.

A locale name is typically of the form *language*[*_territory*][*.codeset*][*@modifier*], where *language* is an ISO 639 language code, *territory* is an ISO 3166 country code, and *codeset* is a character set or encoding identifier like **ISO−8859−1** or **UTF-8**. For a list of all supported locales, try "locale −a" (see *locale*(1)).

If *locale* is NULL, the current locale is only queried, not modified.

On startup of the main program, the portable **"C"** locale is selected as default. A program may be made portable to all locales by calling:

```
setlocale(LC_ALL, "");
```

after program initialization, and then:

- using the values returned from a *localeconv*(3) call for locale-dependent information;
- using the multibyte and wide character functions for text processing if **MB_CUR_MAX > 1**;
- using *strcoll*(3) and *strxfrm*(3) to compare strings; and

- using *wcscoll*(3) and *wcsxfrm*(3) to compare wide-character strings.

RETURN VALUE

A successful call to **setlocale**() returns an opaque string that corresponds to the locale set. This string may be allocated in static storage. The string returned is such that a subsequent call with that string and its associated category will restore that part of the process's locale. The return value is NULL if the request cannot be honored.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setlocale()	Thread safety	MT-Unsafe const:locale env

STANDARDS

C11, POSIX.1-2008.

Categories

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
C11, POSIX.1-2008.

LC_MESSAGES
POSIX.1-2008.

Others: GNU.

HISTORY

POSIX.1-2001, C89.

Categories

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
C89, POSIX.1-2001.

LC_MESSAGES
POSIX.1-2001.

Others: GNU.

SEE ALSO

locale(1), *localedef*(1), *isalpha*(3), *localeconv*(3), *nl_langinfo*(3), *rpmatch*(3), *strcoll*(3), *strftime*(3), *charsets*(7), *locale*(7)

NAME

setlogmask − set log priority mask

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <syslog.h>

int setlogmask(int *mask***);**

DESCRIPTION

A process has a log priority mask that determines which calls to *syslog*(3) may be logged. All other calls will be ignored. Logging is enabled for the priorities that have the corresponding bit set in *mask*. The initial mask is such that logging is enabled for all priorities.

The **setlogmask**() function sets this logmask for the calling process, and returns the previous mask. If the *mask* argument is **0**, the current logmask is not modified.

The eight priorities are **LOG_EMERG**, **LOG_ALERT**, **LOG_CRIT**, **LOG_ERR**, **LOG_WARNING**, **LOG_NOTICE**, **LOG_INFO**, and **LOG_DEBUG**. The bit corresponding to a priority *p* is *LOG_MASK(p)*. Some systems also provide a macro *LOG_UPTO(p)* for the mask of all priorities in the above list up to and including *p*.

RETURN VALUE

This function returns the previous log priority mask.

ERRORS

None.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setlogmask()	Thread safety	MT-Unsafe race:LogMask

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

LOG_UPTO() will be included in the next release of the POSIX specification (Issue 8).

SEE ALSO

closelog(3), *openlog*(3), *syslog*(3)

NAME

setnetgrent, endnetgrent, getnetgrent, getnetgrent_r, innetgr − handle network group entries

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <netdb.h>

int setnetgrent(const char *netgroup);
void endnetgrent(void);

int getnetgrent(char **restrict host,
        char **restrict user, char **restrict domain);
int getnetgrent_r(char **restrict host,
        char **restrict user, char **restrict domain,
        char buf[restrict .buflen], size_t buflen);

int innetgr(const char *netgroup, const char *host,
        const char *user, const char *domain);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

setnetgrent(), **endnetgrent**(), **getnetgrent**(), **getnetgrent_r**(), **innetgr**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The *netgroup* is a SunOS invention. A netgroup database is a list of string triples (*hostname*, *username*, *domainname*) or other netgroup names. Any of the elements in a triple can be empty, which means that anything matches. The functions described here allow access to the netgroup databases. The file */etc/nsswitch.conf* defines what database is searched.

The **setnetgrent**() call defines the netgroup that will be searched by subsequent **getnetgrent**() calls. The **getnetgrent**() function retrieves the next netgroup entry, and returns pointers in *host*, *user*, *domain*. A null pointer means that the corresponding entry matches any string. The pointers are valid only as long as there is no call to other netgroup-related functions. To avoid this problem you can use the GNU function **getnetgrent_r**() that stores the strings in the supplied buffer. To free all allocated buffers use **endnetgrent**().

In most cases you want to check only if the triplet (*hostname*, *username*, *domainname*) is a member of a netgroup. The function **innetgr**() can be used for this without calling the above three functions. Again, a null pointer is a wildcard and matches any string. The function is thread-safe.

RETURN VALUE

These functions return 1 on success and 0 for failure.

FILES

/etc/netgroup
/etc/nsswitch.conf

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
setnetgrent(), **getnetgrent_r**(), **innetgr**()	Thread safety	MT-Unsafe race:netgrent locale
endnetgrent()	Thread safety	MT-Unsafe race:netgrent
getnetgrent()	Thread safety	MT-Unsafe race:netgrent race:netgrentbuf locale

In the above table, *netgrent* in *race:netgrent* signifies that if any of the functions **setnetgrent**(), **getnetgrent_r**(), **innetgr**(), **getnetgrent**(), or **endnetgrent**() are used in parallel in different threads of a program, then data races could occur.

VERSIONS

In the BSD implementation, **setnetgrent**() returns void.

STANDARDS

None.

HISTORY

setnetgrent(), **endnetgrent**(), **getnetgrent**(), and **innetgr**() are available on most UNIX systems. **getnetgrent_r**() is not widely available on other systems.

SEE ALSO

sethostent(3), *setprotoent*(3), *setservent*(3)

NAME

shm_open, shm_unlink − create/open or unlink POSIX shared memory objects

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

DESCRIPTION

shm_open() creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to *mmap*(2) the same region of shared memory. The **shm_unlink**() function performs the converse operation, removing an object previously created by **shm_open**().

The operation of **shm_open**() is analogous to that of *open*(2). *name* specifies the shared memory object to be created or opened. For portable use, a shared memory object should be identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX** (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes.

oflag is a bit mask created by ORing together exactly one of **O_RDONLY** or **O_RDWR** and any of the other flags listed here:

O_RDONLY
: Open the object for read access. A shared memory object opened in this way can be *mmap*(2)ed only for read (**PROT_READ**) access.

O_RDWR
: Open the object for read-write access.

O_CREAT
: Create the shared memory object if it does not exist. The user and group ownership of the object are taken from the corresponding effective IDs of the calling process, and the object's permission bits are set according to the low-order 9 bits of *mode*, except that those bits set in the process file mode creation mask (see *umask*(2)) are cleared for the new object. A set of macro constants which can be used to define *mode* is listed in *open*(2). (Symbolic definitions of these constants can be obtained by including <*sys/stat.h*>.)

  A new shared memory object initially has zero length—the size of the object can be set using *ftruncate*(2). The newly allocated bytes of a shared memory object are automatically initialized to 0.

O_EXCL
: If **O_CREAT** was also specified, and a shared memory object with the given *name* already exists, return an error. The check for the existence of the object, and its creation if it does not exist, are performed atomically.

O_TRUNC
: If the shared memory object already exists, truncate it to zero bytes.

Definitions of these flag values can be obtained by including <*fcntl.h*>.

On successful completion **shm_open**() returns a new file descriptor referring to the shared memory object. This file descriptor is guaranteed to be the lowest-numbered file descriptor not previously opened within the process. The **FD_CLOEXEC** flag (see *fcntl*(2)) is set for the file descriptor.

The file descriptor is normally used in subsequent calls to *ftruncate*(2) (for a newly created object) and *mmap*(2). After a call to *mmap*(2) the file descriptor may be closed without affecting the memory mapping.

The operation of **shm_unlink**() is analogous to *unlink*(2): it removes a shared memory object name, and, once all processes have unmapped the object, deallocates and destroys the contents of the associated memory region. After a successful **shm_unlink**(), attempts to **shm_open**() an object with the

same *name* fail (unless **O_CREAT** was specified, in which case a new, distinct object is created).

RETURN VALUE

On success, **shm_open**() returns a file descriptor (a nonnegative integer). On success, **shm_unlink**() returns 0. On failure, both functions return −1 and set *errno* to indicate the error.

ERRORS

EACCES
Permission to **shm_unlink**() the shared memory object was denied.

EACCES
Permission was denied to **shm_open**() *name* in the specified *mode*, or **O_TRUNC** was specified and the caller does not have write permission on the object.

EEXIST
Both **O_CREAT** and **O_EXCL** were specified to **shm_open**() and the shared memory object specified by *name* already exists.

EINVAL
The *name* argument to **shm_open**() was invalid.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG
The length of *name* exceeds **PATH_MAX**.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
An attempt was made to **shm_open**() a *name* that did not exist, and **O_CREAT** was not specified.

ENOENT
An attempt was to made to **shm_unlink**() a *name* that does not exist.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
shm_open(), **shm_unlink**()	Thread safety	MT-Safe locale

VERSIONS

POSIX leaves the behavior of the combination of **O_RDONLY** and **O_TRUNC** unspecified. On Linux, this will successfully truncate an existing shared memory object—this may not be so on other UNIX systems.

The POSIX shared memory object implementation on Linux makes use of a dedicated *tmpfs*(5) filesystem that is normally mounted under */dev/shm*.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2. POSIX.1-2001.

POSIX.1-2001 says that the group ownership of a newly created shared memory object is set to either the calling process's effective group ID or "a system default group ID". POSIX.1-2008 says that the group ownership may be set to either the calling process's effective group ID or, if the object is visible in the filesystem, the group ID of the parent directory.

EXAMPLES

The programs below employ POSIX shared memory and POSIX unnamed semaphores to exchange a piece of data. The "bounce" program (which must be run first) raises the case of a string that is placed into the shared memory by the "send" program. Once the data has been modified, the "send" program then prints the contents of the modified shared memory. An example execution of the two programs is the following:

```
$ ./pshm_ucase_bounce /myshm &
```

```
[1] 270171
$ ./pshm_ucase_send /myshm hello
HELLO
```

Further detail about these programs is provided below.

Program source: pshm_ucase.h

The following header file is included by both programs below. Its primary purpose is to define a structure that will be imposed on the memory object that is shared between the two programs.

```
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

#define BUF_SIZE 1024   /* Maximum size for exchanged string */

/* Define a structure that will be imposed on the shared
   memory object */

struct shmbuf {
    sem_t  sem1;            /* POSIX unnamed semaphore */
    sem_t  sem2;            /* POSIX unnamed semaphore */
    size_t cnt;             /* Number of bytes used in 'buf' */
    char   buf[BUF_SIZE];   /* Data being transferred */
};
```

Program source: pshm_ucase_bounce.c

The "bounce" program creates a new shared memory object with the name given in its command-line argument and sizes the object to match the size of the *shmbuf* structure defined in the header file. It then maps the object into the process's address space, and initializes two POSIX semaphores inside the object to 0.

After the "send" program has posted the first of the semaphores, the "bounce" program upper cases the data that has been placed in the memory by the "send" program and then posts the second semaphore to tell the "send" program that it may now access the shared memory.

```
/* pshm_ucase_bounce.c

   Licensed under GNU General Public License v2 or later.
*/
#include <ctype.h>

#include "pshm_ucase.h"

int
main(int argc, char *argv[])
{
    int            fd;
    char           *shmpath;
    struct shmbuf  *shmp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s /shm-path\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
        shmpath = argv[1];

        /* Create shared memory object and set its size to the size
           of our structure. */

        fd = shm_open(shmpath, O_CREAT | O_EXCL | O_RDWR, 0600);
        if (fd == -1)
            errExit("shm_open");

        if (ftruncate(fd, sizeof(struct shmbuf)) == -1)
            errExit("ftruncate");

        /* Map the object into the caller's address space. */

        shmp = mmap(NULL, sizeof(*shmp), PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
        if (shmp == MAP_FAILED)
            errExit("mmap");

        /* Initialize semaphores as process-shared, with value 0. */

        if (sem_init(&shmp->sem1, 1, 0) == -1)
            errExit("sem_init-sem1");
        if (sem_init(&shmp->sem2, 1, 0) == -1)
            errExit("sem_init-sem2");

        /* Wait for 'sem1' to be posted by peer before touching
           shared memory. */

        if (sem_wait(&shmp->sem1) == -1)
            errExit("sem_wait");

        /* Convert data in shared memory into upper case. */

        for (size_t j = 0; j < shmp->cnt; j++)
            shmp->buf[j] = toupper((unsigned char) shmp->buf[j]);

        /* Post 'sem2' to tell the peer that it can now
           access the modified data in shared memory. */

        if (sem_post(&shmp->sem2) == -1)
            errExit("sem_post");

        /* Unlink the shared memory object. Even if the peer process
           is still using the object, this is okay. The object will
           be removed only after all open references are closed. */

        shm_unlink(shmpath);

        exit(EXIT_SUCCESS);
    }
```

Program source: pshm_ucase_send.c

The "send" program takes two command-line arguments: the pathname of a shared memory object previously created by the "bounce" program and a string that is to be copied into that object.

The program opens the shared memory object and maps the object into its address space. It then copies the data specified in its second argument into the shared memory, and posts the first semaphore, which tells the "bounce" program that it can now access that data. After the "bounce" program posts the

second semaphore, the "send" program prints the contents of the shared memory on standard output.

```
/* pshm_ucase_send.c

   Licensed under GNU General Public License v2 or later.
*/
#include <string.h>

#include "pshm_ucase.h"

int
main(int argc, char *argv[])
{
    int            fd;
    char           *shmpath, *string;
    size_t         len;
    struct shmbuf  *shmp;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s /shm-path string\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    shmpath = argv[1];
    string = argv[2];
    len = strlen(string);

    if (len > BUF_SIZE) {
        fprintf(stderr, "String is too long\n");
        exit(EXIT_FAILURE);
    }

    /* Open the existing shared memory object and map it
       into the caller's address space. */

    fd = shm_open(shmpath, O_RDWR, 0);
    if (fd == -1)
        errExit("shm_open");

    shmp = mmap(NULL, sizeof(*shmp), PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
    if (shmp == MAP_FAILED)
        errExit("mmap");

    /* Copy data into the shared memory object. */

    shmp->cnt = len;
    memcpy(&shmp->buf, string, len);

    /* Tell peer that it can now access shared memory. */

    if (sem_post(&shmp->sem1) == -1)
        errExit("sem_post");

    /* Wait until peer says that it has finished accessing
       the shared memory. */

    if (sem_wait(&shmp->sem2) == -1)
        errExit("sem_wait");
```

```
            /* Write modified data in shared memory to standard output. */

            write(STDOUT_FILENO, &shmp->buf, len);
            write(STDOUT_FILENO, "\n", 1);

            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

close(2), *fchmod*(2), *fchown*(2), *fcntl*(2), *fstat*(2), *ftruncate*(2), *memfd_create*(2), *mmap*(2), *open*(2), *umask*(2), *shm_overview*(7)

NAME

siginterrupt − allow signals to interrupt system calls

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

[[deprecated]] int siginterrupt(int *sig***, int** *flag***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

siginterrupt():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

The **siginterrupt**() function changes the restart behavior when a system call is interrupted by the signal *sig*. If the *flag* argument is false (0), then system calls will be restarted if interrupted by the specified signal *sig*. This is the default behavior in Linux.

If the *flag* argument is true (1) and no data has been transferred, then a system call interrupted by the signal *sig* will return −1 and *errno* will be set to **EINTR**.

If the *flag* argument is true (1) and data transfer has started, then the system call will be interrupted and will return the actual amount of data transferred.

RETURN VALUE

The **siginterrupt**() function returns 0 on success. It returns −1 if the signal number *sig* is invalid, with *errno* set to indicate the error.

ERRORS

EINVAL
The specified signal number is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
siginterrupt()	Thread safety	MT-Unsafe const:sigintr

STANDARDS

POSIX.1-2008.

HISTORY

4.3BSD, POSIX.1-2001. Obsolete in POSIX.1-2008, recommending the use of *sigaction*(2) with the **SA_RESTART** flag instead.

SEE ALSO

signal(2)

NAME

signbit − test sign of a real floating-point number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

int signbit(*x***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

signbit():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

signbit() is a generic macro which can work on all real floating-point types. It returns a nonzero value if the value of *x* has its sign bit set.

This is not the same as *x < 0.0*, because IEEE 754 floating point allows zero to be signed. The comparison *−0.0 < 0.0* is false, but *signbit(−0.0)* will return a nonzero value.

NaNs and infinities have a sign bit.

RETURN VALUE

The **signbit**() macro returns nonzero if the sign of *x* is negative; otherwise it returns zero.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
signbit()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

SEE ALSO

copysign(3)

NAME

significand, significandf, significandl − get mantissa of floating-point number

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double significand(double *x***);**
float significandf(float *x***);**
long double significandl(long double *x***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

significand(), **significandf**(), **significandl**():
/* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the mantissa of *x* scaled to the range [**1**, **FLT_RADIX**). They are equivalent to

```
scalb(x, (double) −ilogb(x))
```

This function exists mainly for use in certain standardized tests for IEEE 754 conformance.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
significand(), **significandf**(), **significandl**()	Thread safety	MT-Safe

STANDARDS

None.

significand()
BSD.

HISTORY

significand()
BSD.

SEE ALSO

ilogb(3), *scalb*(3)

NAME

sigpause − atomically release blocked signals and wait for interrupt

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

[[deprecated]] int sigpause(int *sigmask***); /* BSD (but see NOTES) */**

[[deprecated]] int sigpause(int *sig***); /* POSIX.1 / SysV / UNIX 95 */**

DESCRIPTION

Don't use this function. Use *sigsuspend*(2) instead.

The function **sigpause**() is designed to wait for some signal. It changes the process's signal mask (set of blocked signals), and then waits for a signal to arrive. Upon arrival of a signal, the original signal mask is restored.

RETURN VALUE

If **sigpause**() returns, it was interrupted by a signal and the return value is −1 with *errno* set to **EINTR**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigpause()	Thread safety	MT-Safe

VERSIONS

On Linux, this routine is a system call only on the Sparc (sparc64) architecture.

glibc uses the BSD version if the **_BSD_SOURCE** feature test macro is defined and none of **_POSIX_SOURCE**, **_POSIX_C_SOURCE**, **_XOPEN_SOURCE**, **_GNU_SOURCE**, or **_SVID_SOURCE** is defined. Otherwise, the System V version is used, and feature test macros must be defined as follows to obtain the declaration:

- Since glibc 2.26: _XOPEN_SOURCE >= 500
- glibc 2.25 and earlier: _XOPEN_SOURCE

Since glibc 2.19, only the System V version is exposed by <*signal.h*>; applications that formerly used the BSD **sigpause**() should be amended to use *sigsuspend*(2).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. Obsoleted in POSIX.1-2008.

The classical BSD version of this function appeared in 4.2BSD. It sets the process's signal mask to *sigmask*. UNIX 95 standardized the incompatible System V version of this function, which removes only the specified signal *sig* from the process's signal mask. The unfortunate situation with two incompatible functions with the same name was solved by the **sigsuspend**(2) function, that takes a *sigset_t* * argument (instead of an *int*).

SEE ALSO

kill(2), *sigaction*(2), *sigprocmask*(2), *sigsuspend*(2), *sigblock*(3), *sigvec*(3), *feature_test_macros*(7)

NAME

sigqueue − queue a signal and data to a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigqueue():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

sigqueue() sends the signal specified in *sig* to the process whose PID is given in *pid*. The permissions required to send a signal are the same as for *kill*(2). As with *kill*(2), the null signal (0) can be used to check if a process with a given PID exists.

The *value* argument is used to specify an accompanying item of data (either an integer or a pointer value) to be sent with the signal, and has the following type:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
};
```

If the receiving process has installed a handler for this signal using the **SA_SIGINFO** flag to *sigaction*(2), then it can obtain this data via the *si_value* field of the *siginfo_t* structure passed as the second argument to the handler. Furthermore, the *si_code* field of that structure will be set to **SI_QUEUE**.

RETURN VALUE

On success, **sigqueue**() returns 0, indicating that the signal was successfully queued to the receiving process. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

EAGAIN
The limit of signals which may be queued has been reached. (See *signal*(7) for further information.)

EINVAL
sig was invalid.

EPERM
The process does not have permission to send the signal to the receiving process. For the required permissions, see *kill*(2).

ESRCH
No process has a PID matching *pid*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigqueue()	Thread safety	MT-Safe

VERSIONS

C library/kernel differences

On Linux, **sigqueue**() is implemented using the *rt_sigqueueinfo*(2) system call. The system call differs in its third argument, which is the *siginfo_t* structure that will be supplied to the receiving process's signal handler or returned by the receiving process's *sigtimedwait*(2) call. Inside the glibc **sigqueue**() wrapper, this argument, *uinfo*, is initialized as follows:

```
uinfo.si_signo = sig;      /* Argument supplied to sigqueue() */
uinfo.si_code = SI_QUEUE;
uinfo.si_pid = getpid();   /* Process ID of sender */
uinfo.si_uid = getuid();   /* Real UID of sender */
uinfo.si_value = val;      /* Argument supplied to sigqueue() */
```

STANDARDS

POSIX.1-2008.

HISTORY

Linux 2.2. POSIX.1-2001.

NOTES

If this function results in the sending of a signal to the process that invoked it, and that signal was not blocked by the calling thread, and no other threads were willing to handle this signal (either by having it unblocked, or by waiting for it using *sigwait*(3)), then at least some signal must be delivered to this thread before this function returns.

SEE ALSO

kill(2), *rt_sigqueueinfo*(2), *sigaction*(2), *signal*(2), *pthread_sigqueue*(3), *sigwait*(3), *signal*(7)

NAME

sigset, sighold, sigrelse, sigignore − System V signal API

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

typedef void (*sighandler_t)(int);

[[deprecated]] sighandler_t sigset(int sig, sighandler_t disp);

[[deprecated]] int sighold(int sig);
[[deprecated]] int sigrelse(int sig);
[[deprecated]] int sigignore(int sig);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigset(), **sighold**(), **sigrelse**(), **sigignore**():
_XOPEN_SOURCE >= 500

DESCRIPTION

These functions are provided in glibc as a compatibility interface for programs that make use of the historical System V signal API. This API is obsolete: new applications should use the POSIX signal API (**sigaction**(2), *sigprocmask*(2), etc.)

The **sigset**() function modifies the disposition of the signal *sig*. The *disp* argument can be the address of a signal handler function, or one of the following constants:

SIG_DFL
: Reset the disposition of *sig* to the default.

SIG_IGN
: Ignore *sig*.

SIG_HOLD
: Add *sig* to the process's signal mask, but leave the disposition of *sig* unchanged.

If *disp* specifies the address of a signal handler, then *sig* is added to the process's signal mask during execution of the handler.

If *disp* was specified as a value other than **SIG_HOLD**, then *sig* is removed from the process's signal mask.

The dispositions for **SIGKILL** and **SIGSTOP** cannot be changed.

The **sighold**() function adds *sig* to the calling process's signal mask.

The **sigrelse**() function removes *sig* from the calling process's signal mask.

The **sigignore**() function sets the disposition of *sig* to **SIG_IGN**.

RETURN VALUE

On success, **sigset**() returns **SIG_HOLD** if *sig* was blocked before the call, or the signal's previous disposition if it was not blocked before the call. On error, **sigset**() returns −1, with *errno* set to indicate the error. (But see BUGS below.)

The **sighold**(), **sigrelse**(), and **sigignore**() functions return 0 on success; on error, these functions return −1 and set *errno* to indicate the error.

ERRORS

For **sigset**() see the ERRORS under *sigaction*(2) and *sigprocmask*(2).

For **sighold**() and **sigrelse**() see the ERRORS under *sigprocmask*(2).

For **sigignore**(), see the errors under *sigaction*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigset(), **sighold**(), **sigrelse**(), **sigignore**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

sighandler_t

GNU. POSIX.1 uses the same type but without a *typedef*.

HISTORY

glibc 2.1. SVr4, POSIX.1-2001. POSIX.1-2008 marks these functions as obsolete, recommending the use of *sigaction*(2), *sigprocmask*(2), *pthread_sigmask*(3), and *sigsuspend*(2) instead.

NOTES

The **sigset**() function provides reliable signal handling semantics (as when calling *sigaction*(2) with *sa_mask* equal to 0).

On System V, the **signal**() function provides unreliable semantics (as when calling *sigaction*(2) with *sa_mask* equal to *SA_RESETHAND* | *SA_NODEFER*). On BSD, **signal**() provides reliable semantics. POSIX.1-2001 leaves these aspects of **signal**() unspecified. See *signal*(2) for further details.

In order to wait for a signal, BSD and System V both provided a function named *sigpause*(3), but this function has a different argument on the two systems. See *sigpause*(3) for details.

BUGS

Before glibc 2.2, **sigset**() did not unblock *sig* if *disp* was specified as a value other than **SIG_HOLD**.

Before glibc 2.5, **sigset**() does not correctly return the previous disposition of the signal in two cases. First, if *disp* is specified as **SIG_HOLD**, then a successful **sigset**() always returns **SIG_HOLD**. Instead, it should return the previous disposition of the signal (unless the signal was blocked, in which case **SIG_HOLD** should be returned). Second, if the signal is currently blocked, then the return value of a successful **sigset**() should be **SIG_HOLD**. Instead, the previous disposition of the signal is returned. These problems have been fixed since glibc 2.5.

SEE ALSO

kill(2), *pause*(2), *sigaction*(2), *signal*(2), *sigprocmask*(2), *raise*(3), *sigpause*(3), *sigvec*(3), *signal*(7)

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember − POSIX signal set operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigemptyset(), **sigfillset**(), **sigaddset**(), **sigdelset**(), **sigismember**():
_POSIX_C_SOURCE

DESCRIPTION

These functions allow the manipulation of POSIX signal sets.

sigemptyset() initializes the signal set given by *set* to empty, with all signals excluded from the set.

sigfillset() initializes *set* to full, including all signals.

sigaddset() and **sigdelset**() add and delete respectively signal *signum* from *set*.

sigismember() tests whether *signum* is a member of *set*.

Objects of type *sigset_t* must be initialized by a call to either **sigemptyset**() or **sigfillset**() before being passed to the functions **sigaddset**(), **sigdelset**(), and **sigismember**() or the additional glibc functions described below (**sigisemptyset**(), **sigandset**(), and *sigorset*())The results are undefined if this is not done.

RETURN VALUE

sigemptyset(), **sigfillset**(), **sigaddset**(), and **sigdelset**() return 0 on success and −1 on error.

sigismember() returns 1 if *signum* is a member of *set*, 0 if *signum* is not a member, and −1 on error.

On error, these functions set *errno* to indicate the error.

ERRORS

EINVAL
signum is not a valid signal.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigemptyset(), **sigfillset**(), **sigaddset**(), **sigdelset**(), **sigismember**(), **sigisemptyset**(), **sigorset**(), **sigandset**()	Thread safety	MT-Safe

VERSIONS

GNU

If the **_GNU_SOURCE** feature test macro is defined, then <*signal.h*> exposes three other functions for manipulating signal sets:

```
int sigisemptyset(const sigset_t *set);
int sigorset(sigset_t *dest, const sigset_t *left,
        const sigset_t *right);
int sigandset(sigset_t *dest, const sigset_t *left,
        const sigset_t *right);
```

sigisemptyset() returns 1 if *set* contains no signals, and 0 otherwise.

sigorset() places the union of the sets *left* and *right* in *dest*. **sigandset**() places the intersection of the sets *left* and *right* in *dest*. Both functions return 0 on success, and −1 on failure.

These functions are nonstandard (a few other systems provide similar functions) and their use should

be avoided in portable applications.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

When creating a filled signal set, the glibc **sigfillset**() function does not include the two real-time signals used internally by the NPTL threading implementation. See *nptl*(7) for details.

SEE ALSO

sigaction(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2)

NAME

sigvec, sigblock, sigsetmask, siggetmask, sigmask − BSD signal API

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <signal.h>

[[deprecated]] int sigvec(int sig, const struct sigvec *vec,
                          struct sigvec *ovec);

[[deprecated]] int sigmask(int signum);

[[deprecated]] int sigblock(int mask);
[[deprecated]] int sigsetmask(int mask);
[[deprecated]] int siggetmask(void);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

These functions are provided in glibc as a compatibility interface for programs that make use of the historical BSD signal API. This API is obsolete: new applications should use the POSIX signal API (**sigaction**(2), *sigprocmask*(2), etc.).

The **sigvec**() function sets and/or gets the disposition of the signal *sig* (like the POSIX *sigaction*(2)). If *vec* is not NULL, it points to a *sigvec* structure that defines the new disposition for *sig*. If *ovec* is not NULL, it points to a *sigvec* structure that is used to return the previous disposition of *sig*. To obtain the current disposition of *sig* without changing it, specify NULL for *vec*, and a non-null pointer for *ovec*.

The dispositions for **SIGKILL** and **SIGSTOP** cannot be changed.

The *sigvec* structure has the following form:

```
struct sigvec {
    void (*sv_handler)(int); /* Signal disposition */
    int    sv_mask;          /* Signals to be blocked in handler */
    int    sv_flags;         /* Flags */
};
```

The *sv_handler* field specifies the disposition of the signal, and is either: the address of a signal handler function; **SIG_DFL**, meaning the default disposition applies for the signal; or **SIG_IGN**, meaning that the signal is ignored.

If *sv_handler* specifies the address of a signal handler, then *sv_mask* specifies a mask of signals that are to be blocked while the handler is executing. In addition, the signal for which the handler is invoked is also blocked. Attempts to block **SIGKILL** or **SIGSTOP** are silently ignored.

If *sv_handler* specifies the address of a signal handler, then the *sv_flags* field specifies flags controlling what happens when the handler is called. This field may contain zero or more of the following flags:

SV_INTERRUPT
: If the signal handler interrupts a blocking system call, then upon return from the handler the system call is not restarted: instead it fails with the error **EINTR**. If this flag is not specified, then system calls are restarted by default.

SV_RESETHAND
: Reset the disposition of the signal to the default before calling the signal handler. If this flag is not specified, then the handler remains established until explicitly removed by a later call to **sigvec**() or until the process performs an *execve*(2).

SV_ONSTACK
Handle the signal on the alternate signal stack (historically established under BSD using the obsolete **sigstack**() function; the POSIX replacement is *sigaltstack*(2)).

The **sigmask**() macro constructs and returns a "signal mask" for *signum*. For example, we can initialize the *vec.sv_mask* field given to **sigvec**() using code such as the following:

```
vec.sv_mask = sigmask(SIGQUIT) | sigmask(SIGABRT);
            /* Block SIGQUIT and SIGABRT during
               handler execution */
```

The **sigblock**() function adds the signals in *mask* to the process's signal mask (like POSIX *sigprocmask(SIG_BLOCK)*), and returns the process's previous signal mask. Attempts to block **SIGKILL** or **SIGSTOP** are silently ignored.

The **sigsetmask**() function sets the process's signal mask to the value given in *mask* (like POSIX *sigprocmask(SIG_SETMASK)*), and returns the process's previous signal mask.

The **siggetmask**() function returns the process's current signal mask. This call is equivalent to *sigblock(0)*.

RETURN VALUE

The **sigvec**() function returns 0 on success; on error, it returns −1 and sets *errno* to indicate the error.

The **sigblock**() and **sigsetmask**() functions return the previous signal mask.

The **sigmask**() macro returns the signal mask for *signum*.

ERRORS

See the ERRORS under *sigaction*(2) and *sigprocmask*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigvec(), **sigmask**(), **sigblock**(), **sigsetmask**(), **siggetmask**()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

sigvec()
sigblock()
sigmask()
sigsetmask()
4.3BSD.

siggetmask()
Unclear origin.

sigvec()
Removed in glibc 2.21.

NOTES

On 4.3BSD, the **signal**() function provided reliable semantics (as when calling **sigvec**() with *vec.sv_mask* equal to 0). On System V, **signal**() provides unreliable semantics. POSIX.1 leaves these aspects of **signal**() unspecified. See *signal*(2) for further details.

In order to wait for a signal, BSD and System V both provided a function named *sigpause*(3), but this function has a different argument on the two systems. See *sigpause*(3) for details.

SEE ALSO

kill(2), *pause*(2), *sigaction*(2), *signal*(2), *sigprocmask*(2), *raise*(3), *sigpause*(3), *sigset*(3), *signal*(7)

NAME

sigwait − wait for a signal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <signal.h>

int sigwait(const sigset_t *restrict *set***, int *restrict** *sig***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigwait():
Since glibc 2.26:
_POSIX_C_SOURCE >= 199506L
glibc 2.25 and earlier:
_POSIX_C_SOURCE

DESCRIPTION

The **sigwait**() function suspends execution of the calling thread until one of the signals specified in the signal set *set* becomes pending. The function accepts the signal (removes it from the pending list of signals), and returns the signal number in *sig*.

The operation of **sigwait**() is the same as *sigwaitinfo*(2), except that:

- **sigwait**() returns only the signal number, rather than a *siginfo_t* structure describing the signal.
- The return values of the two functions are different.

RETURN VALUE

On success, **sigwait**() returns 0. On error, it returns a positive error number (listed in ERRORS).

ERRORS

EINVAL
set contains an invalid signal number.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sigwait()	Thread safety	MT-Safe

VERSIONS

sigwait() is implemented using *sigtimedwait*(2).

The glibc implementation of **sigwait**() silently ignores attempts to wait for the two real-time signals that are used internally by the NPTL threading implementation. See *nptl*(7) for details.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

See *pthread_sigmask*(3).

SEE ALSO

sigaction(2), *signalfd*(2), *sigpending*(2), *sigsuspend*(2), *sigwaitinfo*(2), *sigsetops*(3), *signal*(7)

NAME

sin, sinf, sinl − sine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sinf(), **sinl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the sine of *x*, where *x* is given in radians.

RETURN VALUE

On success, these functions return the sine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity or negative infinity, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sin(), **sinf**(), **sinl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

BUGS

Before glibc 2.10, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred.

SEE ALSO

acos(3), *asin*(3), *atan*(3), *atan2*(3), *cos*(3), *csin*(3), *sincos*(3), *tan*(3)

NAME

sincos, sincosf, sincosl − calculate sin and cos simultaneously

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <math.h>

void sincos(double x, double *sin, double *cos);
void sincosf(float x, float *sin, float *cos);
void sincosl(long double x, long double *sin, long double *cos);
```

DESCRIPTION

Several applications need sine and cosine of the same angle *x*. These functions compute both at the same time, and store the results in **sin* and **cos*. Using this function can be more efficient than two separate calls to *sin*(3) and *cos*(3).

If *x* is a NaN, a NaN is returned in **sin* and **cos*.

If *x* is positive infinity or negative infinity, a domain error occurs, and a NaN is returned in **sin* and **cos*.

RETURN VALUE

These functions return *void*.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sincos(), **sincosf**(), **sincosl**()	Thread safety	MT-Safe

STANDARDS

GNU.

HISTORY

glibc 2.1.

NOTES

To see the performance advantage of **sincos**(), it may be necessary to disable *gcc*(1) built-in optimizations, using flags such as:

```
cc -O -lm -fno-builtin prog.c
```

BUGS

Before glibc 2.22, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred.

SEE ALSO

cos(3), *sin*(3), *tan*(3)

NAME

sinh, sinhf, sinhl − hyperbolic sine function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sinhf(), **sinhl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the hyperbolic sine of *x*, which is defined mathematically as:

sinh(x) = (exp(x) − exp(−x)) / 2

RETURN VALUE

On success, these functions return the hyperbolic sine of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity (negative infinity), positive infinity (negative infinity) is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the same sign as *x*.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sinh(), **sinhf**(), **sinhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

acosh(3), *asinh*(3), *atanh*(3), *cosh*(3), *csinh*(3), *tanh*(3)

NAME

sleep − sleep for a specified number of seconds

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

unsigned int sleep(unsigned int *seconds***);**

DESCRIPTION

sleep() causes the calling thread to sleep either until the number of real-time seconds specified in *seconds* have elapsed or until a signal arrives which is not ignored.

RETURN VALUE

Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sleep()	Thread safety	MT-Unsafe sig:SIGCHLD/linux

VERSIONS

On Linux, **sleep**() is implemented via *nanosleep*(2). See the *nanosleep*(2) man page for a discussion of the clock used.

On some systems, **sleep**() may be implemented using *alarm*(2) and **SIGALRM** (POSIX.1 permits this); mixing calls to *alarm*(2) and **sleep**() is a bad idea.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

CAVEATS

Using *longjmp*(3) from a signal handler or modifying the handling of **SIGALRM** while sleeping will cause undefined results.

SEE ALSO

sleep(1), *alarm*(2), *nanosleep*(2), *signal*(2), *signal*(7)

NAME

SLIST_EMPTY, SLIST_ENTRY, SLIST_FIRST, SLIST_FOREACH, SLIST_HEAD, SLIST_HEAD_INITIALIZER, SLIST_INIT, SLIST_INSERT_AFTER, SLIST_INSERT_HEAD, SLIST_NEXT, SLIST_REMOVE, SLIST_REMOVE_HEAD − implementation of a singly linked list

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/queue.h>

SLIST_ENTRY(TYPE);

SLIST_HEAD(HEADNAME, TYPE);
SLIST_HEAD SLIST_HEAD_INITIALIZER(SLIST_HEAD *head***);**
void SLIST_INIT(SLIST_HEAD ***_head_);**

int SLIST_EMPTY(SLIST_HEAD ***_head_);**

void SLIST_INSERT_HEAD(SLIST_HEAD ***_head_,**
struct TYPE ***_elm_, SLIST_ENTRY** *NAME***);**
void SLIST_INSERT_AFTER(struct TYPE ***_listelm_,**
struct TYPE ***_elm_, SLIST_ENTRY** *NAME***);**

struct TYPE *SLIST_FIRST(SLIST_HEAD ***_head_);**
struct TYPE *SLIST_NEXT(struct TYPE ***_elm_, SLIST_ENTRY** *NAME***);**

SLIST_FOREACH(struct TYPE ***_var_, SLIST_HEAD ***_head_**, SLIST_ENTRY** *NAME***);**

void SLIST_REMOVE(SLIST_HEAD ***_head_, struct TYPE ***_elm_**,**
SLIST_ENTRY *NAME***);**
void SLIST_REMOVE_HEAD(SLIST_HEAD ***_head_,**
SLIST_ENTRY *NAME***);**

DESCRIPTION

These macros define and operate on doubly linked lists.

In the macro definitions, *TYPE* is the name of a user-defined structure, that must contain a field of type *SLIST_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user-defined structure that must be declared using the macro **SLIST_HEAD**().

Creation

A singly linked list is headed by a structure defined by the **SLIST_HEAD**() macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An *SLIST_HEAD* structure is declared as follows:

```
SLIST_HEAD(HEADNAME, TYPE) head;
```

where *struct HEADNAME* is the structure to be defined, and *struct TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

SLIST_ENTRY() declares a structure that connects the elements in the list.

SLIST_HEAD_INITIALIZER() evaluates to an initializer for the list *head*.

SLIST_INIT() initializes the list referenced by *head*.

SLIST_EMPTY() evaluates to true if there are no elements in the list.

Insertion

SLIST_INSERT_HEAD() inserts the new element *elm* at the head of the list.

SLIST_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

Traversal

SLIST_FIRST() returns the first element in the list, or NULL if the list is empty.

SLIST_NEXT() returns the next element in the list.

SLIST_FOREACH() traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

Removal

SLIST_REMOVE() removes the element *elm* from the list.

SLIST_REMOVE_HEAD() removes the element *elm* from the head of the list. For optimum efficiency, elements being removed from the head of the list should explicitly use this macro instead of the generic **SLIST_REMOVE**().

RETURN VALUE

SLIST_EMPTY() returns nonzero if the list is empty, and zero if the list contains at least one entry.

SLIST_FIRST(), and **SLIST_NEXT**() return a pointer to the first or next *TYPE* structure, respectively.

SLIST_HEAD_INITIALIZER() returns an initializer that can be assigned to the list *head*.

STANDARDS

BSD.

HISTORY

4.4BSD.

BUGS

SLIST_FOREACH() doesn't allow *var* to be removed or freed within the loop, as it would interfere with the traversal. **SLIST_FOREACH_SAFE**(), which is present on the BSDs but is not present in glibc, fixes this limitation by allowing *var* to safely be removed from the list and freed from within the loop without interfering with the traversal.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    SLIST_ENTRY(entry) entries;             /* Singly linked list */
};

SLIST_HEAD(slisthead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct slisthead head;                  /* Singly linked list
                                               head */

    SLIST_INIT(&head);                      /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));      /* Insert at the head */
    SLIST_INSERT_HEAD(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));      /* Insert after */
    SLIST_INSERT_AFTER(n1, n2, entries);

    SLIST_REMOVE(&head, n2, entry, entries);/* Deletion */
    free(n2);
```

```
        n3 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);      /* Deletion from the head */
        free(n3);

        for (unsigned int i = 0; i < 5; i++) {
            n1 = malloc(sizeof(struct entry));
            SLIST_INSERT_HEAD(&head, n1, entries);
            n1->data = i;
        }

                                                /* Forward traversal */
        SLIST_FOREACH(np, &head, entries)
            printf("%i\n", np->data);

        while (!SLIST_EMPTY(&head)) {           /* List deletion */
            n1 = SLIST_FIRST(&head);
            SLIST_REMOVE_HEAD(&head, entries);
            free(n1);
        }
        SLIST_INIT(&head);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

insque(3), *queue*(7)

NAME

sockatmark − determine whether socket is at out-of-band mark

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/socket.h>

int sockatmark(int *sockfd***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sockatmark():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

sockatmark() returns a value indicating whether or not the socket referred to by the file descriptor *sockfd* is at the out-of-band mark. If the socket is at the mark, then 1 is returned; if the socket is not at the mark, 0 is returned. This function does not remove the out-of-band mark.

RETURN VALUE

A successful call to **sockatmark**() returns 1 if the socket is at the out-of-band mark, or 0 if it is not. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF
sockfd is not a valid file descriptor.

EINVAL
sockfd is not a file descriptor to which **sockatmark**() can be applied.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sockatmark()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.2.4. POSIX.1-2001.

NOTES

If **sockatmark**() returns 1, then the out-of-band data can be read using the **MSG_OOB** flag of *recv*(2).

Out-of-band data is supported only on some stream socket protocols.

sockatmark() can safely be called from a handler for the **SIGURG** signal.

sockatmark() is implemented using the **SIOCATMARK** *ioctl*(2) operation.

BUGS

Prior to glibc 2.4, **sockatmark**() did not work.

EXAMPLES

The following code can be used after receipt of a **SIGURG** signal to read (and discard) all data up to the mark, and then read the byte of data at the mark:

```
char buf[BUF_LEN];
char oobdata;
int atmark, s;

for (;;) {
    atmark = sockatmark(sockfd);
    if (atmark == -1) {
        perror("sockatmark");
        break;
    }
```

```
            if (atmark)
                break;

            s = read(sockfd, buf, BUF_LEN);
            if (s == -1)
                perror("read");
            if (s <= 0)
                break;
        }

        if (atmark == 1) {
            if (recv(sockfd, &oobdata, 1, MSG_OOB) == -1) {
                perror("recv");
                ...
            }
        }
```

SEE ALSO

fcntl(2), *recv*(2), *send*(2), *tcp*(7)

NAME

sqrt, sqrtf, sqrtl − square root function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sqrtf(), **sqrtl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the nonnegative square root of *x*.

RETURN VALUE

On success, these functions return the square root of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is less than −0, a domain error occurs, and a NaN is returned.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* less than −0
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sqrt(), **sqrtf**(), **sqrtl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

cbrt(3), *csqrt*(3), *hypot*(3)

NAME

sscanf, vsscanf − input string format conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int sscanf(const char *restrict str,
           const char *restrict format, ...);

#include <stdarg.h>

int vsscanf(const char *restrict str,
           const char *restrict format, va_list ap);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

vsscanf():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **sscanf**() family of functions scans formatted input according to *format* as described below. This format may contain *conversion specifications*; the results from such conversions, if any, are stored in the locations pointed to by the *pointer* arguments that follow *format*. Each *pointer* argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.

If the number of conversion specifications in *format* exceeds the number of *pointer* arguments, the results are undefined. If the number of *pointer* arguments exceeds the number of conversion specifications, then the excess *pointer* arguments are evaluated, but are otherwise ignored.

sscanf() These functions read their input from the string pointed to by *str*.

The **vsscanf**() function is analogous to *vsprintf*(3).

The *format* string consists of a sequence of *directives* which describe how to process the sequence of input characters. If processing of a directive fails, no further input is read, and **sscanf**() returns. A "failure" can be either of the following: *input failure*, meaning that input characters were unavailable, or *matching failure*, meaning that the input was inappropriate (see below).

A directive is one of the following:

- A sequence of white-space characters (space, tab, newline, etc.; see *isspace*(3)). This directive matches any amount of white space, including none, in the input.
- An ordinary character (i.e., one other than white space or '%'). This character must exactly match the next character of input.
- A conversion specification, which commences with a '%' (percent) character. A sequence of characters from the input is converted according to this specification, and the result is placed in the corresponding *pointer* argument. If the next item of input does not match the conversion specification, the conversion fails—this is a *matching failure*.

Each *conversion specification* in *format* begins with either the character '%' or the character sequence "**%***n***$**" (see below for the distinction) followed by:

- An optional '*' assignment-suppression character: **sscanf**() reads input as directed by the conversion specification, but discards the input. No corresponding *pointer* argument is required, and this specification is not included in the count of successful assignments returned by **scanf**().
- For decimal conversions, an optional quote character ('). This specifies that the input number may include thousands' separators as defined by the **LC_NUMERIC** category of the current locale. (See *setlocale*(3).) The quote character may precede or follow the '*' assignment-suppression character.
- An optional 'm' character. This is used with string conversions (*%s*, *%c*, *%[*), and relieves the caller of the need to allocate a corresponding buffer to hold the input: instead, **sscanf**() allocates a buffer of sufficient size, and assigns the address of this buffer to the corresponding *pointer* argument, which should be a pointer to a *char* * variable (this variable does not need

to be initialized before the call). The caller should subsequently *free*(3) this buffer when it is no longer required.

- An optional decimal integer which specifies the *maximum field width*. Reading of characters stops either when this maximum is reached or when a nonmatching character is found, whichever happens first. Most conversions discard initial white space characters (the exceptions are noted below), and these discarded characters don't count toward the maximum field width. String input conversions store a terminating null byte ('\0') to mark the end of the input; the maximum field width does not include this terminator.
- An optional *type modifier character*. For example, the **l** type modifier is used with integer conversions such as **%d** to specify that the corresponding *pointer* argument refers to a *long* rather than a pointer to an *int*.
- A *conversion specifier* that specifies the type of input conversion to be performed.

The conversion specifications in *format* are of two forms, either beginning with '%' or beginning with "**%***n***$**". The two forms should not be mixed in the same *format* string, except that a string containing "**%***n***$**" specifications can include **%%** and **%***. If *format* contains '%' specifications, then these correspond in order with successive *pointer* arguments. In the "**%***n***$**" form (which is specified in POSIX.1-2001, but not C99), *n* is a decimal integer that specifies that the converted input should be placed in the location referred to by the *n*-th *pointer* argument following *format*.

Conversions

The following *type modifier characters* can appear in a conversion specification:

h Indicates that the conversion will be one of **d**, **i**, **o**, **u**, **x**, **X**, or **n** and the next pointer is a pointer to a *short* or *unsigned short* (rather than *int*).

hh As for **h**, but the next pointer is a pointer to a *signed char* or *unsigned char*.

j As for **h**, but the next pointer is a pointer to an *intmax_t* or a *uintmax_t*. This modifier was introduced in C99.

l Indicates either that the conversion will be one of **d**, **i**, **o**, **u**, **x**, **X**, or **n** and the next pointer is a pointer to a *long* or *unsigned long* (rather than *int*), or that the conversion will be one of **e**, **f**, or **g** and the next pointer is a pointer to *double* (rather than *float*). If used with **%c** or **%s**, the corresponding parameter is considered as a pointer to a wide character or wide-character string respectively.

ll (ell-ell) Indicates that the conversion will be one of **b**, **d**, **i**, **o**, **u**, **x**, **X**, or **n** and the next pointer is a pointer to a *long long* or *unsigned long long* (rather than *int*).

L Indicates that the conversion will be either **e**, **f**, or **g** and the next pointer is a pointer to *long double* or (as a GNU extension) the conversion will be **d**, **i**, **o**, **u**, or **x** and the next pointer is a pointer to *long long*.

q equivalent to **L**. This specifier does not exist in ANSI C.

t As for **h**, but the next pointer is a pointer to a *ptrdiff_t*. This modifier was introduced in C99.

z As for **h**, but the next pointer is a pointer to a *size_t*. This modifier was introduced in C99.

The following *conversion specifiers* are available:

% Matches a literal '%'. That is, **%%** in the format string matches a single input '%' character. No conversion is done (but initial white space characters are discarded), and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with *0x* or *0X*, in base 8 if it begins with *0*, and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an unsigned octal integer; the next pointer must be a pointer to *unsigned int*.

u Matches an unsigned decimal integer; the next pointer must be a pointer to *unsigned int*.

x Matches an unsigned hexadecimal integer (that may optionally begin with a prefix of *0x* or *0X*, which is discarded); the next pointer must be a pointer to *unsigned int*.

X Equivalent to **x**.

f Matches an optionally signed floating-point number; the next pointer must be a pointer to *float*.

e Equivalent to **f**.

g Equivalent to **f**.

E Equivalent to **f**.

a (C99) Equivalent to **f**.

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to the initial element of a character array that is long enough to hold the input sequence and the terminating null byte ('\0'), which is added automatically. The input string stops at white space or at the maximum field width, whichever occurs first.

c Matches a sequence of characters whose length is specified by the *maximum field width* (default 1); the next pointer must be a pointer to *char*, and there must be enough room for all the characters (no terminating null byte is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for all the characters in the string, plus a terminating null byte. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket **[** character and a close bracket **]** character. The set *excludes* those characters if the first character after the open bracket is a circumflex (^). To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character − is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, **[^]0−9−]** means the set "everything except close bracket, zero through nine, and hyphen". The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

p Matches a pointer value (as printed by **%p** in *printf*(3)); the next pointer must be a pointer to a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*, or variant whose size matches the (optionally) supplied integer length modifier. This is *not* a conversion and does *not* increase the count returned by the function. The assignment can be suppressed with the ***** assignment-suppression character, but the effect on the return value is undefined. Therefore **%*n** conversions should not be used.

RETURN VALUE

On success, these functions return the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure.

The value **EOF** is returned if the end of input is reached before either the first successful conversion or a matching failure occurs.

ERRORS

EILSEQ
Input byte sequence does not form a valid character.

EINVAL
Not enough arguments; or *format* is NULL.

ENOMEM
Out of memory.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sscanf(), **vsscanf**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

The **q** specifier is the 4.4BSD notation for *long long*, while **ll** or the usage of **L** in integer conversions is the GNU notation.

The Linux version of these functions is based on the *GNU libio* library. Take a look at the *info* documentation of *GNU libc (glibc-1.08)* for a more concise description.

NOTES

The 'a' assignment-allocation modifier

Originally, the GNU C library supported dynamic allocation for string inputs (as a nonstandard extension) via the **a** character. (This feature is present at least as far back as glibc 2.0.) Thus, one could write the following to have **sscanf**() allocate a buffer for a string, with a pointer to that buffer being returned in **buf*:

```
char *buf;
sscanf(str, "%as", &buf);
```

The use of the letter **a** for this purpose was problematic, since **a** is also specified by the ISO C standard as a synonym for **f** (floating-point input). POSIX.1-2008 instead specifies the **m** modifier for assignment allocation (as documented in DESCRIPTION, above).

Note that the **a** modifier is not available if the program is compiled with *gcc −std=c99* or *gcc −D_ISOC99_SOURCE* (unless **_GNU_SOURCE** is also specified), in which case the **a** is interpreted as a specifier for floating-point numbers (see above).

Support for the **m** modifier was added to glibc 2.7, and new programs should use that modifier instead of **a**.

As well as being standardized by POSIX, the **m** modifier has the following further advantages over the use of **a**:

- It may also be applied to **%c** conversion specifiers (e.g., **%3mc**).
- It avoids ambiguity with respect to the **%a** floating-point conversion specifier (and is unaffected by *gcc −std=c99* etc.).

BUGS

Numeric conversion specifiers

Use of the numeric conversion specifiers produces Undefined Behavior for invalid input. See C11 7.21.6.2/10. This is a bug in the ISO C standard, and not an inherent design issue with the API. However, current implementations are not safe from that bug, so it is not recommended to use them. Instead, programs should use functions such as *strtol*(3) to parse numeric input. Alternatively, mitigate it by specifying a maximum field width.

Nonstandard modifiers

These functions are fully C99 conformant, but provide the additional modifiers **q** and **a** as well as an additional behavior of the **L** and **ll** modifiers. The latter may be considered to be a bug, as it changes the behavior of modifiers defined in C99.

Some combinations of the type modifiers and conversion specifiers defined by C99 do not make sense (e.g., **%Ld**). While they may have a well-defined behavior on Linux, this need not to be so on other architectures. Therefore it usually is better to use modifiers that are not defined by C99 at all, that is, use **q** instead of **L** in combination with **d**, **i**, **o**, **u**, **x**, and **X** conversions or **ll**.

The usage of **q** is not the same as on 4.4BSD, as it may be used in float conversions equivalently to **L**.

EXAMPLES

To use the dynamic allocation conversion specifier, specify **m** as a length modifier (thus **%ms** or **%m[***range***]**). The caller must *free*(3) the returned string, as in the following example:

```
char *p;
int n;

errno = 0;
n = sscanf(str, "%m[a-z]", &p);
if (n == 1) {
    printf("read: %s\n", p);
    free(p);
} else if (errno != 0) {
    perror("sscanf");
} else {
    fprintf(stderr, "No matching characters\n");
}
```

As shown in the above example, it is necessary to call *free*(3) only if the **sscanf**() call successfully read a string.

SEE ALSO

getc(3), *printf*(3), *setlocale*(3), *strtod*(3), *strtol*(3), *strtoul*(3)

NAME

SIMPLEQ_EMPTY, SIMPLEQ_ENTRY, SIMPLEQ_FIRST, SIMPLEQ_FOREACH, SIMPLEQ_HEAD, SIMPLEQ_HEAD_INITIALIZER, SIMPLEQ_INIT, SIMPLEQ_INSERT_AFTER, SIMPLEQ_INSERT_HEAD, SIMPLEQ_INSERT_TAIL, SIMPLEQ_NEXT, SIMPLEQ_REMOVE, SIMPLEQ_REMOVE_HEAD, STAILQ_CONCAT, STAILQ_EMPTY, STAILQ_ENTRY, STAILQ_FIRST, STAILQ_FOREACH, STAILQ_HEAD, STAILQ_HEAD_INITIALIZER, STAILQ_INIT, STAILQ_INSERT_AFTER, STAILQ_INSERT_HEAD, STAILQ_INSERT_TAIL, STAILQ_NEXT, STAILQ_REMOVE, STAILQ_REMOVE_HEAD, − implementation of a singly linked tail queue

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/queue.h>

STAILQ_ENTRY(TYPE);

STAILQ_HEAD(HEADNAME, TYPE);
STAILQ_HEAD STAILQ_HEAD_INITIALIZER(STAILQ_HEAD *head***);**
void STAILQ_INIT(STAILQ_HEAD ***_head_);**

int STAILQ_EMPTY(STAILQ_HEAD ***_head_);**

void STAILQ_INSERT_HEAD(STAILQ_HEAD ***_head_,**
struct TYPE ***_elm_, STAILQ_ENTRY** *NAME***);**
void STAILQ_INSERT_TAIL(STAILQ_HEAD ***_head_,**
struct TYPE ***_elm_, STAILQ_ENTRY** *NAME***);**
void STAILQ_INSERT_AFTER(STAILQ_HEAD ***_head_, struct TYPE ***_listelm_**,**
struct TYPE ***_elm_, STAILQ_ENTRY** *NAME***);**

struct TYPE *STAILQ_FIRST(STAILQ_HEAD ***_head_);**
struct TYPE *STAILQ_NEXT(struct TYPE ***_elm_, STAILQ_ENTRY** *NAME***);**

STAILQ_FOREACH(struct TYPE ***_var_, STAILQ_HEAD ***_head_**, STAILQ_ENTRY** *NAME***);**

void STAILQ_REMOVE(STAILQ_HEAD ***_head_, struct TYPE ***_elm_**, TYPE,**
STAILQ_ENTRY *NAME***);**
void STAILQ_REMOVE_HEAD(STAILQ_HEAD ***_head_,**
STAILQ_ENTRY *NAME***);**

void STAILQ_CONCAT(STAILQ_HEAD ***_head1_, STAILQ_HEAD ***_head2_**);**
Note: Identical macros prefixed with SIMPLEQ instead of STAILQ exist; see NOTES.

DESCRIPTION

These macros define and operate on singly linked tail queues.

In the macro definitions, *TYPE* is the name of a user-defined structure, that must contain a field of type *STAILQ_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user-defined structure that must be declared using the macro **STAILQ_HEAD**().

Creation

A singly linked tail queue is headed by a structure defined by the **STAILQ_HEAD**() macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the tail queue after an existing element, at the head of the tail queue, or at the end of the tail queue. A *STAILQ_HEAD* structure is declared as follows:

```
STAILQ_HEAD(HEADNAME, TYPE) head;
```

where *struct HEADNAME* is the structure to be defined, and *struct TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

STAILQ_ENTRY() declares a structure that connects the elements in the tail queue.

STAILQ_HEAD_INITIALIZER() evaluates to an initializer for the tail queue *head*.

STAILQ_INIT() initializes the tail queue referenced by *head*.

STAILQ_EMPTY() evaluates to true if there are no items on the tail queue.

Insertion

STAILQ_INSERT_HEAD() inserts the new element *elm* at the head of the tail queue.

STAILQ_INSERT_TAIL() inserts the new element *elm* at the end of the tail queue.

STAILQ_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

Traversal

STAILQ_FIRST() returns the first item on the tail queue or NULL if the tail queue is empty.

STAILQ_NEXT() returns the next item on the tail queue, or NULL this item is the last.

STAILQ_FOREACH() traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

Removal

STAILQ_REMOVE() removes the element *elm* from the tail queue.

STAILQ_REMOVE_HEAD() removes the element at the head of the tail queue. For optimum efficiency, elements being removed from the head of the tail queue should use this macro explicitly rather than the generic **STAILQ_REMOVE**() macro.

Other features

STAILQ_CONCAT() concatenates the tail queue headed by *head2* onto the end of the one headed by *head1* removing all entries from the former.

RETURN VALUE

STAILQ_EMPTY() returns nonzero if the queue is empty, and zero if the queue contains at least one entry.

STAILQ_FIRST(), and **STAILQ_NEXT**() return a pointer to the first or next *TYPE* structure, respectively.

STAILQ_HEAD_INITIALIZER() returns an initializer that can be assigned to the queue *head*.

VERSIONS

Some BSDs provide SIMPLEQ instead of STAILQ. They are identical, but for historical reasons they were named differently on different BSDs. STAILQ originated on FreeBSD, and SIMPLEQ originated on NetBSD. For compatibility reasons, some systems provide both sets of macros. glibc provides both STAILQ and SIMPLEQ, which are identical except for a missing SIMPLEQ equivalent to **STAILQ_CONCAT**().

BUGS

STAILQ_FOREACH() doesn't allow *var* to be removed or freed within the loop, as it would interfere with the traversal. **STAILQ_FOREACH_SAFE**(), which is present on the BSDs but is not present in glibc, fixes this limitation by allowing *var* to safely be removed from the list and freed from within the loop without interfering with the traversal.

STANDARDS

BSD.

HISTORY

4.4BSD.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    STAILQ_ENTRY(entry) entries;        /* Singly linked tail queue */
};
```

```
STAILQ_HEAD(stailhead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct stailhead head;                  /* Singly linked tail queue
                                               head */

    STAILQ_INIT(&head);                     /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));      /* Insert at the head */
    STAILQ_INSERT_HEAD(&head, n1, entries);

    n1 = malloc(sizeof(struct entry));      /* Insert at the tail */
    STAILQ_INSERT_TAIL(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));      /* Insert after */
    STAILQ_INSERT_AFTER(&head, n1, n2, entries);

    STAILQ_REMOVE(&head, n2, entry, entries); /* Deletion */
    free(n2);

    n3 = STAILQ_FIRST(&head);
    STAILQ_REMOVE_HEAD(&head, entries);     /* Deletion from the head */
    free(n3);

    n1 = STAILQ_FIRST(&head);
    n1->data = 0;
    for (unsigned int i = 1; i < 5; i++) {
        n1 = malloc(sizeof(struct entry));
        STAILQ_INSERT_HEAD(&head, n1, entries);
        n1->data = i;
    }
                                            /* Forward traversal */
    STAILQ_FOREACH(np, &head, entries)
        printf("%i\n", np->data);
                                            /* TailQ deletion */
    n1 = STAILQ_FIRST(&head);
    while (n1 != NULL) {
        n2 = STAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    STAILQ_INIT(&head);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

insque(3), *queue*(7)

NAME

static_assert, _Static_assert − fail compilation if assertion is false

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <assert.h>

void static_assert(scalar *constant-expression***, const char ****msg***);**

/* Since C23: */
void static_assert(scalar *constant-expression***);**

DESCRIPTION

This macro is similar to **assert**(3), but it works at compile time, generating a compilation error (with an optional message) when the input is false (i.e., compares equal to zero).

If the input is nonzero, no code is emitted.

msg must be a string literal. Since C23, this argument is optional.

There's a keyword, **_Static_assert**(), that behaves identically, and can be used without including <*assert.h*>.

RETURN VALUE

No value is returned.

VERSIONS

In C11, the second argument (*msg*) was mandatory; since C23, it can be omitted.

STANDARDS

C11 and later.

EXAMPLES

static_assert() can't be used in some places, like for example at global scope. For that, a macro **must_be**() can be written in terms of **static_assert**(). The following program uses the macro to get the size of an array safely.

```
#include <assert.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * This macro behaves like static_assert(), failing to
 * compile if its argument is not true.  However, it always
 * returns 0, which allows using it everywhere an expression
 * can be used.
 */
#define must_be(e)                                              \
(                                                               \
    0 * (int) sizeof(                                           \
        struct {                                                \
            static_assert(e);                                   \
            int  ISO_C_forbids_a_struct_with_no_members;        \
        }                                                       \
    )                                                           \
)

#define is_same_type(a, b)  \
    __builtin_types_compatible_p(typeof(a), typeof(b))

#define is_array(arr)       (!is_same_type((arr), &*(arr)))
```

```
#define must_be_array(arr)  must_be(is_array(arr))

#define sizeof_array(arr)   (sizeof(arr) + must_be_array(arr))
#define nitems(arr)         (sizeof((arr)) / sizeof((arr)[0]) \
                             + must_be_array(arr))

int     foo[10];
int8_t  bar[sizeof_array(foo)];

int
main(void)
{
    for (size_t i = 0; i < nitems(foo); i++) {
        foo[i] = i;
    }

    memcpy(bar, foo, sizeof_array(bar));

    for (size_t i = 0; i < nitems(bar); i++) {
        printf("%d,", bar[i]);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

assert(3)

NAME

statvfs, fstatvfs − get filesystem statistics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/statvfs.h>

int statvfs(const char *restrict *path***, struct statvfs *restrict** *buf***);**
int fstatvfs(int *fd***, struct statvfs ***buf***);**

DESCRIPTION

The function **statvfs**() returns information about a mounted filesystem. *path* is the pathname of any file within the mounted filesystem. *buf* is a pointer to a *statvfs* structure defined approximately as follows:

```
struct statvfs {
    unsigned long  f_bsize;    /* Filesystem block size */
    unsigned long  f_frsize;   /* Fragment size */
    fsblkcnt_t     f_blocks;   /* Size of fs in f_frsize units */
    fsblkcnt_t     f_bfree;    /* Number of free blocks */
    fsblkcnt_t     f_bavail;   /* Number of free blocks for
                                  unprivileged users */
    fsfilcnt_t     f_files;    /* Number of inodes */
    fsfilcnt_t     f_ffree;    /* Number of free inodes */
    fsfilcnt_t     f_favail;   /* Number of free inodes for
                                  unprivileged users */
    unsigned long  f_fsid;     /* Filesystem ID */
    unsigned long  f_flag;     /* Mount flags */
    unsigned long  f_namemax;  /* Maximum filename length */
};
```

Here the types *fsblkcnt_t* and *fsfilcnt_t* are defined in <*sys/types.h*>. Both used to be *unsigned long*.

The field *f_flag* is a bit mask indicating various options that were employed when mounting this filesystem. It contains zero or more of the following flags:

ST_MANDLOCK
Mandatory locking is permitted on the filesystem (see *fcntl*(2)).

ST_NOATIME
Do not update access times; see *mount*(2).

ST_NODEV
Disallow access to device special files on this filesystem.

ST_NODIRATIME
Do not update directory access times; see *mount*(2).

ST_NOEXEC
Execution of programs is disallowed on this filesystem.

ST_NOSUID
The set-user-ID and set-group-ID bits are ignored by *exec*(3) for executable files on this filesystem

ST_RDONLY
This filesystem is mounted read-only.

ST_RELATIME
Update atime relative to mtime/ctime; see *mount*(2).

ST_SYNCHRONOUS
Writes are synched to the filesystem immediately (see the description of **O_SYNC** in *open*(2)).

It is unspecified whether all members of the returned struct have meaningful values on all filesystems.

fstatvfs() returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
(**statvfs**()) Search permission is denied for a component of the path prefix of *path*. (See also *path_resolution*(7).)

EBADF
(**fstatvfs**()) *fd* is not a valid open file descriptor.

EFAULT
Buf or *path* points to an invalid address.

EINTR
This call was interrupted by a signal; see *signal*(7).

EIO An I/O error occurred while reading from the filesystem.

ELOOP
(**statvfs**()) Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG
(**statvfs**()) *path* is too long.

ENOENT
(**statvfs**()) The file referred to by *path* does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOSYS
The filesystem does not support this call.

ENOTDIR
(**statvfs**()) A component of the path prefix of *path* is not a directory.

EOVERFLOW
Some values were too large to be represented in the returned struct.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
statvfs(), **fstatvfs**()	Thread safety	MT-Safe

VERSIONS

Only the **ST_NOSUID** and **ST_RDONLY** flags of the *f_flag* field are specified in POSIX.1. To obtain definitions of the remaining flags, one must define **_GNU_SOURCE**.

NOTES

The Linux kernel has system calls *statfs*(2) and *fstatfs*(2) to support this library call.

The glibc implementations of

```
pathconf(path, _PC_REC_XFER_ALIGN);
pathconf(path, _PC_ALLOC_SIZE_MIN);
pathconf(path, _PC_REC_MIN_XFER_SIZE);
```

respectively use the *f_frsize*, *f_frsize*, and *f_bsize* fields returned by a call to **statvfs**() with the argument *path*.

Under Linux, *f_favail* is always the same as *f_ffree*, and there's no way for a filesystem to report otherwise. This is not an issue, since no filesystems with an inode root reservation exist.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Before glibc 2.13, **statvfs**() populated the bits of the *f_flag* field by scanning the mount options shown in */proc/mounts*. However, starting with Linux 2.6.36, the underlying *statfs*(2) system call provides the necessary information via the *f_flags* field, and since glibc 2.13, the **statvfs**() function will use information from that field rather than scanning */proc/mounts*.

SEE ALSO

statfs(2)

NAME

stdarg, va_start, va_arg, va_end, va_copy − variable argument lists

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdarg.h>

void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void va_end(va_list ap);
void va_copy(va_list dest, va_list src);
```

DESCRIPTION

A function may be called with a varying number of arguments of varying types. The include file *<stdarg.h>* declares a type *va_list* and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must declare an object of type *va_list* which is used by the macros **va_start**(), **va_arg**(), and **va_end**().

va_start()

The **va_start**() macro initializes *ap* for subsequent use by **va_arg**() and **va_end**(), and must be called first.

The argument *last* is the name of the last argument before the variable argument list, that is, the last argument of which the calling function knows the type.

Because the address of this argument may be used in the **va_start**() macro, it should not be declared as a register variable, or as a function or an array type.

va_arg()

The **va_arg**() macro expands to an expression that has the type and value of the next argument in the call. The argument *ap* is the *va_list ap* initialized by **va_start**(). Each call to **va_arg**() modifies *ap* so that the next call returns the next argument. The argument *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to *type*.

The first use of the **va_arg**() macro after that of the **va_start**() macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

If there is no next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

If *ap* is passed to a function that uses **va_arg(***ap***,***type***),** then the value of *ap* is undefined after the return of that function.

va_end()

Each invocation of **va_start**() must be matched by a corresponding invocation of **va_end**() in the same function. After the call **va_end(***ap***)** the variable *ap* is undefined. Multiple traversals of the list, each bracketed by **va_start**() and **va_end**() are possible. **va_end**() may be a macro or a function.

va_copy()

The **va_copy**() macro copies the (previously initialized) variable argument list *src* to *dest*. The behavior is as if **va_start**() were applied to *dest* with the same *last* argument, followed by the same number of **va_arg**() invocations that was used to reach the current state of *src*.

An obvious implementation would have a *va_list* be a pointer to the stack frame of the variadic function. In such a setup (by far the most common) there seems nothing against an assignment

```
va_list aq = ap;
```

Unfortunately, there are also systems that make it an array of pointers (of length 1), and there one needs

```
va_list aq;
*aq = *ap;
```

Finally, on systems where arguments are passed in registers, it may be necessary for **va_start**() to allocate memory, store the arguments there, and also an indication of which argument is next, so that **va_arg**() can step through the list. Now **va_end**() can free the allocated memory again. To

accommodate this situation, C99 adds a macro **va_copy**(), so that the above assignment can be replaced by

```
va_list aq;
va_copy(aq, ap);
...
va_end(aq);
```

Each invocation of **va_copy**() must be matched by a corresponding invocation of **va_end**() in the same function. Some systems that do not supply **va_copy**() have **__va_copy** instead, since that was the name used in the draft proposal.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
va_start(), **va_end**(), **va_copy**()	Thread safety	MT-Safe
va_arg()	Thread safety	MT-Safe race:ap

STANDARDS

C11, POSIX.1-2008.

HISTORY

va_start()
va_arg()
va_end()
C89, POSIX.1-2001.

va_copy()
C99, POSIX.1-2001.

CAVEATS

Unlike the historical **varargs** macros, the **stdarg** macros do not permit programmers to code a function with no fixed arguments. This problem generates work mainly when converting **varargs** code to **stdarg** code, but it also creates difficulties for variadic functions that wish to pass all of their arguments on to a function that takes a *va_list* argument, such as *vfprintf*(3).

EXAMPLES

The function *foo* takes a string of format characters and prints out the argument associated with each format character based on the type.

```
#include <stdio.h>
#include <stdarg.h>

void
foo(char *fmt, ...)   /* '...' is C syntax for a variadic function */

{
    va_list ap;
    int d;
    char c;
    char *s;

    va_start(ap, fmt);
    while (*fmt)
        switch (*fmt++) {
        case 's':              /* string */
            s = va_arg(ap, char *);
            printf("string %s\n", s);
            break;
        case 'd':              /* int */
            d = va_arg(ap, int);
            printf("int %d\n", d);
            break;
        case 'c':              /* char */
```

```
                /* need a cast here since va_arg only
                   takes fully promoted types */
                c = (char) va_arg(ap, int);
                printf("char %c\n", c);
                break;
            }
        va_end(ap);
    }
```

SEE ALSO

vprintf(3), *vscanf*(3), *vsyslog*(3)

NAME

stdin, stdout, stderr − standard I/O streams

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

DESCRIPTION

Under normal circumstances every UNIX program has three streams opened for it when it starts up, one for input, one for output, and one for printing diagnostic or error messages. These are typically attached to the user's terminal (see *tty*(4)) but might instead refer to files or other devices, depending on what the parent process chose to set up. (See also the "Redirection" section of *sh*(1)

The input stream is referred to as "standard input"; the output stream is referred to as "standard output"; and the error stream is referred to as "standard error". These terms are abbreviated to form the symbols used to refer to these files, namely *stdin*, *stdout*, and *stderr*.

Each of these symbols is a *stdio*(3) macro of type pointer to *FILE*, and can be used with functions like *fprintf*(3) or *fread*(3).

Since *FILE*s are a buffering wrapper around UNIX file descriptors, the same underlying files may also be accessed using the raw UNIX file interface, that is, the functions like *read*(2) and *lseek*(2).

On program startup, the integer file descriptors associated with the streams *stdin*, *stdout*, and *stderr* are 0, 1, and 2, respectively. The preprocessor symbols **STDIN_FILENO**, **STDOUT_FILENO**, and **STDERR_FILENO** are defined with these values in <*unistd.h*>. (Applying *freopen*(3) to one of these streams can change the file descriptor number associated with the stream.)

Note that mixing use of *FILE*s and raw file descriptors can produce unexpected results and should generally be avoided. (For the masochistic among you: POSIX.1, section 8.2.3, describes in detail how this interaction is supposed to work.) A general rule is that file descriptors are handled in the kernel, while stdio is just a library. This means for example, that after an *exec*(3), the child inherits all open file descriptors, but all old streams have become inaccessible.

Since the symbols *stdin*, *stdout*, and *stderr* are specified to be macros, assigning to them is nonportable. The standard streams can be made to refer to different files with help of the library function *freopen*(3), specially introduced to make it possible to reassign *stdin*, *stdout*, and *stderr*. The standard streams are closed by a call to *exit*(3) and by normal program termination.

STANDARDS

C11, POSIX.1-2008.

The standards also stipulate that these three streams shall be open at program startup.

HISTORY

C89, POSIX.1-2001.

NOTES

The stream *stderr* is unbuffered. The stream *stdout* is line-buffered when it points to a terminal. Partial lines will not appear until *fflush*(3) or *exit*(3) is called, or a newline is printed. This can produce unexpected results, especially with debugging output. The buffering mode of the standard streams (or any other stream) can be changed using the *setbuf*(3) or *setvbuf*(3) call. Note that in case *stdin* is associated with a terminal, there may also be input buffering in the terminal driver, entirely unrelated to stdio buffering. (Indeed, normally terminal input is line buffered in the kernel.) This kernel input handling can be modified using calls like *tcsetattr*(3); see also *stty*(1), and *termios*(3).

SEE ALSO

csh(1), *sh*(1), *open*(2), *fopen*(3), *stdio*(3)

NAME

stdio − standard input/output library functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

FILE ****stdin*;**
FILE ****stdout*;**
FILE ****stderr*;**

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start of the file (byte zero), unless the file is opened with append mode. If append mode is used, it is unspecified whether the position indicator will be placed at the start or the end of the file. The position indicator is maintained by subsequent reads, writes, and positioning requests. All input occurs as if the characters were read by successive calls to the *fgetc*(3) function; all output takes place as if all characters were written by successive calls to the *fputc*(3) function.

A file is disassociated from a stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transferred to the host environment) before the stream is disassociated from the file. The value of a pointer to a *FILE* object is indeterminate after a file is closed (garbage).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at the start). If the main function returns to its original caller, or the *exit*(3) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other methods of program termination, such as *abort*(3) do not bother about closing files properly.

At program startup, three text streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). These streams are abbreviated *stdin*, *stdout*, and *stderr*. When opened, the standard error stream is not fully buffered; the standard input and output streams are fully buffered if and only if the streams do not refer to an interactive device.

Output streams that refer to terminal devices are always line buffered by default; pending output to such streams is written automatically whenever an input stream that refers to a terminal device is read. In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush*(3) the standard output before going off and computing so that the output will appear.

The *stdio* library is a part of the library **libc** and routines are automatically loaded as needed by *cc*(1)The SYNOPSIS sections of the following manual pages indicate which include files are to be used, what the compiler declaration for the function looks like and which external variables are of interest.

The following are defined as macros; these names may not be reused without first removing their current definitions with **#undef**: **BUFSIZ**, **EOF**, **FILENAME_MAX**, **FOPEN_MAX**, **L_cuserid**, **L_ctermid**, **L_tmpnam**, **NULL**, **SEEK_END**, **SEEK_SET**, **SEEK_CUR**, **TMP_MAX**, **clearerr**, **feof**, **ferror**, **fileno**, **getc**, **getchar**, **putc**, **putchar**, **stderr**, **stdin**, **stdout**. Function versions of the macro functions **feof**, **ferror**, **clearerr**, **fileno**, **getc**, **getchar**, **putc**, and **putchar** exist and will be used if the macros definitions are explicitly removed.

List of functions

Function	Description
clearerr(3)	check and reset stream status

fclose(3)	close a stream
fdopen(3)	stream open functions
feof(3)	check and reset stream status
ferror(3)	check and reset stream status
fflush(3)	flush a stream
fgetc(3)	get next character or word from input stream
fgetpos(3)	reposition a stream
fgets(3)	get a line from a stream
fileno(3)	return the integer descriptor of the argument stream
fmemopen(3)	open memory as stream
fopen(3)	stream open functions
fopencookie(3)	open a custom stream
fprintf(3)	formatted output conversion
fpurge(3)	flush a stream
fputc(3)	output a character or word to a stream
fputs(3)	output a line to a stream
fread(3)	binary stream input/output
freopen(3)	stream open functions
fscanf(3)	input format conversion
fseek(3)	reposition a stream
fsetpos(3)	reposition a stream
ftell(3)	reposition a stream
fwrite(3)	binary stream input/output
getc(3)	get next character or word from input stream
getchar(3)	get next character or word from input stream
gets(3)	get a line from a stream
getw(3)	get next character or word from input stream
mktemp(3)	make temporary filename (unique)
open_memstream(3)	open a dynamic memory buffer stream
open_wmemstream(3)	open a dynamic memory buffer stream
perror(3)	system error messages
printf(3)	formatted output conversion
putc(3)	output a character or word to a stream
putchar(3)	output a character or word to a stream
puts(3)	output a line to a stream
putw(3)	output a character or word to a stream
remove(3)	remove directory entry
rewind(3)	reposition a stream
scanf(3)	input format conversion
setbuf(3)	stream buffering operations
setbuffer(3)	stream buffering operations
setlinebuf(3)	stream buffering operations
setvbuf(3)	stream buffering operations
sprintf(3)	formatted output conversion
sscanf(3)	input format conversion
strerror(3)	system error messages
sys_errlist(3)	system error messages
sys_nerr(3)	system error messages
tempnam(3)	temporary file routines
tmpfile(3)	temporary file routines
tmpnam(3)	temporary file routines
ungetc(3)	un-get character from input stream
vfprintf(3)	formatted output conversion
vfscanf(3)	input format conversion
vprintf(3)	formatted output conversion
vscanf(3)	input format conversion
vsprintf(3)	formatted output conversion
vsscanf(3)	input format conversion

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

SEE ALSO

close(2), *open*(2), *read*(2), *write*(2), *stdout*(3), *unlocked_stdio*(3)

NAME

__fbufsize, __flbf, __fpending, __fpurge, __freadable, __freading, __fsetlocking, __fwritable, __fwriting, _flushlbf − interfaces to stdio FILE structure

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <stdio_ext.h>

size_t __fbufsize(FILE *stream);
size_t __fpending(FILE *stream);
int __flbf(FILE *stream);
int __freadable(FILE *stream);
int __fwritable(FILE *stream);
int __freading(FILE *stream);
int __fwriting(FILE *stream);
int __fsetlocking(FILE *stream, int type);
void _flushlbf(void);
void __fpurge(FILE *stream);
```

DESCRIPTION

Solaris introduced routines to allow portable access to the internals of the *FILE* structure, and glibc also implemented these.

The **__fbufsize**() function returns the size of the buffer currently used by the given stream.

The **__fpending**() function returns the number of bytes in the output buffer. For wide-oriented streams the unit is wide characters. This function is undefined on buffers in reading mode, or opened read-only.

The **__flbf**() function returns a nonzero value if the stream is line-buffered, and zero otherwise.

The **__freadable**() function returns a nonzero value if the stream allows reading, and zero otherwise.

The **__fwritable**() function returns a nonzero value if the stream allows writing, and zero otherwise.

The **__freading**() function returns a nonzero value if the stream is read-only, or if the last operation on the stream was a read operation, and zero otherwise.

The **__fwriting**() function returns a nonzero value if the stream is write-only (or append-only), or if the last operation on the stream was a write operation, and zero otherwise.

The **__fsetlocking**() function can be used to select the desired type of locking on the stream. It returns the current type. The *type* argument can take the following three values:

FSETLOCKING_INTERNAL
: Perform implicit locking around every operation on the given stream (except for the *_unlocked ones). This is the default.

FSETLOCKING_BYCALLER
: The caller will take care of the locking (possibly using *flockfile*(3) in case there is more than one thread), and the stdio routines will not do locking until the state is reset to **FSETLOCKING_INTERNAL**.

FSETLOCKING_QUERY
: Don't change the type of locking. (Only return it.)

The **_flushlbf**() function flushes all line-buffered streams. (Presumably so that output to a terminal is forced out, say before reading keyboard input.)

The **__fpurge**() function discards the contents of the stream's buffer.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
__fbufsize(), **__fpending**(), **__fpurge**(), **__fsetlocking**()	Thread safety	MT-Safe race:stream
__flbf(), **__freadable**(), **__freading**(), **__fwritable**(), **__fwriting**(), **_flushlbf**()	Thread safety	MT-Safe

SEE ALSO

flockfile(3), *fpurge*(3)

NAME

stpncpy, strncpy − fill a fixed-size buffer with non-null bytes from a string, padding with null bytes as needed

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strncpy(char dst[restrict .dsize], const char *restrict src,
        size_t dsize);
char *stpncpy(char dst[restrict .dsize], const char *restrict src,
        size_t dsize);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

stpncpy():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _GNU_SOURCE

DESCRIPTION

These functions copy non-null bytes from the string pointed to by *src* into the array pointed to by *dst*. If the source has too few non-null bytes to fill the destination, the functions pad the destination with trailing null bytes. If the destination buffer, limited by its size, isn't large enough to hold the copy, the resulting character sequence is truncated. For the difference between the two functions, see RETURN VALUE.

An implementation of these functions might be:

```
char *
strncpy(char *restrict dst, const char *restrict src, size_t dsize)
{
    stpncpy(dst, src, dsize);
    return dst;
}

char *
stpncpy(char *restrict dst, const char *restrict src, size_t dsize)
{
    size_t  dlen;

    dlen = strnlen(src, dsize);
    return memset(mempcpy(dst, src, dlen), 0, dsize - dlen);
}
```

RETURN VALUE

strncpy()
: returns *dst*.

stpncpy()
: returns a pointer to one after the last character in the destination character sequence.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
stpncpy(), **strncpy**()	Thread safety	MT-Safe

STANDARDS

strncpy()
: C11, POSIX.1-2008.

stpncpy()
POSIX.1-2008.

HISTORY

strncpy()
C89, POSIX.1-2001, SVr4, 4.3BSD.

stpncpy()
glibc 1.07. POSIX.1-2008.

CAVEATS

The name of these functions is confusing. These functions produce a null-padded character sequence, not a string (see *string_copying*(7)). For example:

```
strncpy(buf, "1", 5);       // { '1',   0,   0,   0,   0 }
strncpy(buf, "1234", 5);    // { '1', '2', '3', '4',   0 }
strncpy(buf, "12345", 5);   // { '1', '2', '3', '4', '5' }
strncpy(buf, "123456", 5);  // { '1', '2', '3', '4', '5' }
```

It's impossible to distinguish truncation by the result of the call, from a character sequence that just fits the destination buffer; truncation should be detected by comparing the length of the input string with the size of the destination buffer.

If you're going to use this function in chained calls, it would be useful to develop a similar function that accepts a pointer to the end (one after the last element) of the destination buffer instead of its size.

EXAMPLES

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(void)
{
    char    *p;
    char    buf1[20];
    char    buf2[20];
    size_t  len;

    if (sizeof(buf2) < strlen("Hello world!"))
        errx("strncpy: truncating character sequence");
    strncpy(buf2, "Hello world!", sizeof(buf2));
    len = strnlen(buf2, sizeof(buf2));

    printf("[len = %zu]: ", len);
    fwrite(buf2, 1, len, stdout);
    putchar('\n');

    if (sizeof(buf1) < strlen("Hello world!"))
        errx("stpncpy: truncating character sequence");
    p = stpncpy(buf1, "Hello world!", sizeof(buf1));
    len = p - buf1;

    printf("[len = %zu]: ", len);
    fwrite(buf1, 1, len, stdout);
    putchar('\n');

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

wcpncpy(3), *string_copying*(7)

NAME

strcasecmp, strncasecmp − compare two strings ignoring case

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char s1[.n], const char s2[.n], size_t n);
```

DESCRIPTION

The **strcasecmp**() function performs a byte-by-byte comparison of the strings *s1* and *s2*, ignoring the case of the characters. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

The **strncasecmp**() function is similar, except that it compares no more than *n* bytes of *s1* and *s2*.

RETURN VALUE

The **strcasecmp**() and **strncasecmp**() functions return an integer less than, equal to, or greater than zero if *s1* is, after ignoring case, found to be less than, to match, or be greater than *s2*, respectively.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strcasecmp(), **strncasecmp**()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

4.4BSD, POSIX.1-2001.

The **strcasecmp**() and **strncasecmp**() functions first appeared in 4.4BSD, where they were declared in <*string.h*>. Thus, for reasons of historical compatibility, the glibc <*string.h*> header file also declares these functions, if the **_DEFAULT_SOURCE** (or, in glibc 2.19 and earlier, **_BSD_SOURCE**) feature test macro is defined.

The POSIX.1-2008 standard says of these functions:

> When the **LC_CTYPE** category of the locale being used is from the POSIX locale, these functions shall behave as if the strings had been converted to lowercase and then a byte comparison performed. Otherwise, the results are unspecified.

SEE ALSO

memcmp(3), *strcmp*(3), *strcoll*(3), *string*(3), *strncmp*(3), *wcscasecmp*(3), *wcsncasecmp*(3)

NAME

strchr, strrchr, strchrnul − locate character in string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

char *strchrnul(const char *s, int c);
```

DESCRIPTION

The **strchr**() function returns a pointer to the first occurrence of the character *c* in the string *s*.

The **strrchr**() function returns a pointer to the last occurrence of the character *c* in the string *s*.

The **strchrnul**() function is like **strchr**() except that if *c* is not found in *s*, then it returns a pointer to the null byte at the end of *s*, rather than NULL.

Here "character" means "byte"; these functions do not work with wide or multibyte characters.

RETURN VALUE

The **strchr**() and **strrchr**() functions return a pointer to the matched character or NULL if the character is not found. The terminating null byte is considered part of the string, so that if *c* is specified as '\0', these functions return a pointer to the terminator.

The **strchrnul**() function returns a pointer to the matched character, or a pointer to the null byte at the end of *s* (i.e., *s+strlen(s)*) if the character is not found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strchr(), **strrchr**(), **strchrnul**()	Thread safety	MT-Safe

STANDARDS

strchr()
strrchr()
: C11, POSIX.1-2008.

strchrnul()
: GNU.

HISTORY

strchr()
strrchr()
: POSIX.1-2001, C89, SVr4, 4.3BSD.

strchrnul()
: glibc 2.1.1.

SEE ALSO

memchr(3), *string*(3), *strlen*(3), *strpbrk*(3), *strsep*(3), *strspn*(3), *strstr*(3), *strtok*(3), *wcschr*(3), *wcsrchr*(3)

NAME

strcmp, strncmp − compare two strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
int strncmp(const char s1[.n], const char s2[.n], size_t n);
```

DESCRIPTION

The **strcmp**() function compares the two strings *s1* and *s2*. The locale is not taken into account (for a locale-aware comparison, see *strcoll*(3)). The comparison is done using unsigned characters.

strcmp() returns an integer indicating the result of the comparison, as follows:

- 0, if the *s1* and *s2* are equal;
- a negative value if *s1* is less than *s2*;
- a positive value if *s1* is greater than *s2*.

The **strncmp**() function is similar, except it compares only the first (at most) *n* bytes of *s1* and *s2*.

RETURN VALUE

The **strcmp**() and **strncmp**() functions return an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strcmp(), **strncmp**()	Thread safety	MT-Safe

VERSIONS

POSIX.1 specifies only that:

> The sign of a nonzero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type *unsigned char*) that differ in the strings being compared.

In glibc, as in most other implementations, the return value is the arithmetic result of subtracting the last compared byte in *s2* from the last compared byte in *s1*. (If the two characters are equal, this difference is 0.)

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

EXAMPLES

The program below can be used to demonstrate the operation of **strcmp**() (when given two arguments) and **strncmp**() (when given three arguments). First, some examples using **strcmp**():

```
$ ./string_comp ABC ABC
<str1> and <str2> are equal
$ ./string_comp ABC AB      # 'C' is ASCII 67; 'C' - '\0' = 67
<str1> is greater than <str2> (67)
$ ./string_comp ABA ABZ     # 'A' is ASCII 65; 'Z' is ASCII 90
<str1> is less than <str2> (-25)
$ ./string_comp ABJ ABC
<str1> is greater than <str2> (7)
$ ./string_comp $'\201' A   # 0201 - 0101 = 0100 (or 64 decimal)
<str1> is greater than <str2> (64)
```

The last example uses *bash*(1)-specific syntax to produce a string containing an 8-bit ASCII code; the result demonstrates that the string comparison uses unsigned characters.

And then some examples using **strncmp**():

```
$ ./string_comp ABC AB 3
<str1> is greater than <str2> (67)
$ ./string_comp ABC AB 2
<str1> and <str2> are equal in the first 2 bytes
```

Program source

```
/* string_comp.c

   Licensed under GNU General Public License v2 or later.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int res;

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <str1> <str2> [<len>]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (argc == 3)
        res = strcmp(argv[1], argv[2]);
    else
        res = strncmp(argv[1], argv[2], atoi(argv[3]));

    if (res == 0) {
        printf("<str1> and <str2> are equal");
        if (argc > 3)
            printf(" in the first %d bytes\n", atoi(argv[3]));
        printf("\n");
    } else if (res < 0) {
        printf("<str1> is less than <str2> (%d)\n", res);
    } else {
        printf("<str1> is greater than <str2> (%d)\n", res);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

memcmp(3), *strcasecmp*(3), *strcoll*(3), *string*(3), *strncasecmp*(3), *strverscmp*(3), *wcscmp*(3), *wcsncmp*(3), *ascii*(7)

NAME

strcoll − compare two strings using the current locale

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

int strcoll(const char *_s1_, const char *_s2_);

DESCRIPTION

The **strcoll**() function compares the two strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. The comparison is based on strings interpreted as appropriate for the program's current locale for category **LC_COLLATE**. (See *setlocale*(3).)

RETURN VALUE

The **strcoll**() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*, when both are interpreted as appropriate for the current locale.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strcoll()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

NOTES

In the *POSIX* or *C* locales **strcoll**() is equivalent to *strcmp*(3).

SEE ALSO

memcmp(3), *setlocale*(3), *strcasecmp*(3), *strcmp*(3), *string*(3), *strxfrm*(3)

NAME

stpcpy, strcpy, strcat − copy or catenate a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *stpcpy(char *restrict dst, const char *restrict src);
char *strcpy(char *restrict dst, const char *restrict src);
char *strcat(char *restrict dst, const char *restrict src);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

stpcpy():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

stpcpy()
strcpy()
These functions copy the string pointed to by *src*, into a string at the buffer pointed to by *dst*. The programmer is responsible for allocating a destination buffer large enough, that is, *strlen(src) + 1*. For the difference between the two functions, see RETURN VALUE.

strcat() This function catenates the string pointed to by *src*, after the string pointed to by *dst* (overwriting its terminating null byte). The programmer is responsible for allocating a destination buffer large enough, that is, *strlen(dst) + strlen(src) + 1*.

An implementation of these functions might be:

```
char *
stpcpy(char *restrict dst, const char *restrict src)
{
    char  *p;

    p = mempcpy(dst, src, strlen(src));
    *p = '\0';

    return p;
}

char *
strcpy(char *restrict dst, const char *restrict src)
{
    stpcpy(dst, src);
    return dst;
}

char *
strcat(char *restrict dst, const char *restrict src)
{
    stpcpy(dst + strlen(dst), src);
    return dst;
}
```

RETURN VALUE

stpcpy()
This function returns a pointer to the terminating null byte of the copied string.

strcpy()
strcat() These functions return *dst*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
stpcpy(), **strcpy**(), **strcat**()	Thread safety	MT-Safe

STANDARDS

stpcpy()
POSIX.1-2008.

strcpy()
strcat() C11, POSIX.1-2008.

STANDARDS

stpcpy()
POSIX.1-2008.

strcpy()
strcat() POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS

The strings *src* and *dst* may not overlap.

If the destination buffer is not large enough, the behavior is undefined. See **_FORTIFY_SOURCE** in *feature_test_macros*(7).

strcat() can be very inefficient. Read about Shlemiel the painter.

EXAMPLES

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(void)
{
    char    *p;
    char    *buf1;
    char    *buf2;
    size_t  len, maxsize;

    maxsize = strlen("Hello ") + strlen("world") + strlen("!") + 1;
    buf1 = malloc(sizeof(*buf1) * maxsize);
    if (buf1 == NULL)
        err(EXIT_FAILURE, "malloc()");
    buf2 = malloc(sizeof(*buf2) * maxsize);
    if (buf2 == NULL)
        err(EXIT_FAILURE, "malloc()");

    p = buf1;
    p = stpcpy(p, "Hello ");
    p = stpcpy(p, "world");
    p = stpcpy(p, "!");
    len = p - buf1;

    printf("[len = %zu]: ", len);
    puts(buf1);  // "Hello world!"
    free(buf1);

    strcpy(buf2, "Hello ");
    strcat(buf2, "world");
```

```
        strcat(buf2, "!");
        len = strlen(buf2);

        printf("[len = %zu]: ", len);
        puts(buf2);  // "Hello world!"
        free(buf2);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

strdup(3), *string*(3), *wcscpy*(3), *string_copying*(7)

NAME

strdup, strndup, strdupa, strndupa − duplicate a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strdup(const char *s);

char *strndup(const char s[.n], size_t n);
char *strdupa(const char *s);
char *strndupa(const char s[.n], size_t n);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strdup():
_XOPEN_SOURCE >= 500
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

strndup():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

strdupa(), **strndupa**():
_GNU_SOURCE

DESCRIPTION

The **strdup**() function returns a pointer to a new string which is a duplicate of the string *s*. Memory for the new string is obtained with *malloc*(3), and can be freed with *free*(3).

The **strndup**() function is similar, but copies at most *n* bytes. If *s* is longer than *n*, only *n* bytes are copied, and a terminating null byte ('\0') is added.

strdupa() and **strndupa**() are similar, but use *alloca*(3) to allocate the buffer.

RETURN VALUE

On success, the **strdup**() function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with *errno* set to indicate the error.

ERRORS

ENOMEM
Insufficient memory available to allocate duplicate string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strdup(), **strndup**(), **strdupa**(), **strndupa**()	Thread safety	MT-Safe

STANDARDS

strdup()
strndup()
POSIX.1-2008.

strdupa()
strndupa()
GNU.

HISTORY

strdup()
SVr4, 4.3BSD-Reno, POSIX.1-2001.

strndup()
POSIX.1-2008.

strdupa()
strndupa()
GNU.

SEE ALSO

alloca(3), *calloc*(3), *free*(3), *malloc*(3), *realloc*(3), *string*(3), *wcsdup*(3)

NAME

strerror, strerrorname_np, strerrordesc_np, strerror_r, strerror_l − return string describing error number

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);
const char *strerrorname_np(int errnum);
const char *strerrordesc_np(int errnum);

int strerror_r(int errnum, char buf[.buflen], size_t buflen);
        /* XSI-compliant */

char *strerror_r(int errnum, char buf[.buflen], size_t buflen);
        /* GNU-specific */

char *strerror_l(int errnum, locale_t locale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strerrorname_np(), **strerrordesc_np**():
_GNU_SOURCE

strerror_r():
The XSI-compliant version is provided if:
(_POSIX_C_SOURCE >= 200112L) && ! _GNU_SOURCE
Otherwise, the GNU-specific version is provided.

DESCRIPTION

The **strerror**() function returns a pointer to a string that describes the error code passed in the argument *errnum*, possibly using the **LC_MESSAGES** part of the current locale to select the appropriate language. (For example, if *errnum* is **EINVAL**, the returned description will be "Invalid argument".) This string must not be modified by the application, and the returned pointer will be invalidated on a subsequent call to **strerror**() or **strerror_l**(), or if the thread that obtained the string exits. No other library function, including *perror*(3), will modify this string.

Like **strerror**(), the **strerrordesc_np**() function returns a pointer to a string that describes the error code passed in the argument *errnum*, with the difference that the returned string is not translated according to the current locale.

The **strerrorname_np**() function returns a pointer to a string containing the name of the error code passed in the argument *errnum*. For example, given **EPERM** as an argument, this function returns a pointer to the string "EPERM". Given **0** as an argument, this function returns a pointer to the string "0".

strerror_r()

strerror_r() is like **strerror**(), but might use the supplied buffer *buf* instead of allocating one internally. This function is available in two versions: an XSI-compliant version specified in POSIX.1-2001 (available since glibc 2.3.4, but not POSIX-compliant until glibc 2.13), and a GNU-specific version (available since glibc 2.0). The XSI-compliant version is provided with the feature test macros settings shown in the SYNOPSIS; otherwise the GNU-specific version is provided. If no feature test macros are explicitly defined, then (since glibc 2.4) **_POSIX_C_SOURCE** is defined by default with the value 200112L, so that the XSI-compliant version of **strerror_r**() is provided by default.

The XSI-compliant **strerror_r**() is preferred for portable applications. It returns the error string in the user-supplied buffer *buf* of length *buflen*.

The GNU-specific **strerror_r**() returns a pointer to a string containing the error message. This may be either a pointer to a string that the function stores in *buf*, or a pointer to some (immutable) static string (in which case *buf* is unused). If the function stores a string in *buf*, then at most *buflen* bytes are stored (the string may be truncated if *buflen* is too small and *errnum* is unknown). The string always includes a terminating null byte ('\0').

strerror_l()

strerror_l() is like **strerror**(), but maps *errnum* to a locale-dependent error message in the locale specified by *locale*. The behavior of **strerror_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** or is not a valid locale object handle.

RETURN VALUE

The **strerror**(), **strerror_l**(), and the GNU-specific **strerror_r**() functions return the appropriate error description string, or an "Unknown error nnn" message if the error number is unknown.

On success, **strerrorname_np**() and **strerrordesc_np**() return the appropriate error description string. If *errnum* is an invalid error number, these functions return NULL.

The XSI-compliant **strerror_r**() function returns 0 on success. On error, a (positive) error number is returned (since glibc 2.13), or −1 is returned and *errno* is set to indicate the error (before glibc 2.13).

POSIX.1-2001 and POSIX.1-2008 require that a successful call to **strerror**() or **strerror_l**() shall leave *errno* unchanged, and note that, since no function return value is reserved to indicate an error, an application that wishes to check for errors should initialize *errno* to zero before the call, and then check *errno* after the call.

ERRORS

EINVAL
The value of *errnum* is not a valid error number.

ERANGE
Insufficient storage was supplied to contain the error description string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strerror()	Thread safety	MT-Safe
strerrorname_np(), **strerrordesc_np**()	Thread safety	MT-Safe
strerror_r(), **strerror_l**()	Thread safety	MT-Safe

Before glibc 2.32, **strerror**() is not MT-Safe.

STANDARDS

strerror()
C11, POSIX.1-2008.

strerror_r()
strerror_l()
POSIX.1-2008.

strerrorname_np()
strerrordesc_np()
GNU.

POSIX.1-2001 permits **strerror**() to set *errno* if the call encounters an error, but does not specify what value should be returned as the function result in the event of an error. On some systems, **strerror**() returns NULL if the error number is unknown. On other systems, **strerror**() returns a string something like "Error nnn occurred" and sets *errno* to **EINVAL** if the error number is unknown. C99 and POSIX.1-2008 require the return value to be non-NULL.

HISTORY

strerror()
POSIX.1-2001, C89.

strerror_r()
POSIX.1-2001.

strerror_l()
glibc 2.6. POSIX.1-2008.

strerrorname_np()

strerrordesc_np()
glibc 2.32.

NOTES

strerrorname_np() and **strerrordesc_np**() are thread-safe and async-signal-safe.

SEE ALSO

err(3), *errno*(3), *error*(3), *perror*(3), *strsignal*(3), *locale*(7), *signal-safety*(7)

NAME

strfmon, strfmon_l − convert monetary value to a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <monetary.h>

ssize_t strfmon(char s[restrict .max], size_t max,
                const char *restrict format, ...);
ssize_t strfmon_l(char s[restrict .max], size_t max, locale_t locale,
                const char *restrict format, ...);
```

DESCRIPTION

The **strfmon**() function formats the specified monetary amount according to the current locale and format specification *format* and places the result in the character array *s* of size *max*.

The **strfmon_l**() function performs the same task, but uses the locale specified by *locale*. The behavior of **strfmon_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)) or is not a valid locale object handle.

Ordinary characters in *format* are copied to *s* without conversion. Conversion specifiers are introduced by a '%' character. Immediately following it there can be zero or more of the following flags:

=*f* The single-byte character *f* is used as the numeric fill character (to be used with a left precision, see below). When not specified, the space character is used.

^ Do not use any grouping characters that might be defined for the current locale. By default, grouping is enabled.

(or **+** The (flag indicates that negative amounts should be enclosed between parentheses. The + flag indicates that signs should be handled in the default way, that is, amounts are preceded by the locale's sign indication, for example, nothing for positive, "−" for negative.

! Omit the currency symbol.

− Left justify all fields. The default is right justification.

Next, there may be a field width: a decimal digit string specifying a minimum field width in bytes. The default is 0. A result smaller than this width is padded with spaces (on the left, unless the left-justify flag was given).

Next, there may be a left precision of the form "#" followed by a decimal digit string. If the number of digits left of the radix character is smaller than this, the representation is padded on the left with the numeric fill character. Grouping characters are not counted in this field width.

Next, there may be a right precision of the form "." followed by a decimal digit string. The amount being formatted is rounded to the specified number of digits prior to formatting. The default is specified in the *frac_digits* and *int_frac_digits* items of the current locale. If the right precision is 0, no radix character is printed. (The radix character here is determined by **LC_MONETARY**, and may differ from that specified by **LC_NUMERIC**.)

Finally, the conversion specification must be ended with a conversion character. The three conversion characters are

% (In this case, the entire specification must be exactly "%%".) Put a '%' character in the result string.

i One argument of type *double* is converted using the locale's international currency format.

n One argument of type *double* is converted using the locale's national currency format.

RETURN VALUE

The **strfmon**() function returns the number of characters placed in the array *s*, not including the terminating null byte, provided the string, including the terminating null byte, fits. Otherwise, it sets *errno* to **E2BIG**, returns −1, and the contents of the array is undefined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strfmon()	Thread safety	MT-Safe locale
strfmon_l()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

EXAMPLES

The call

```
strfmon(buf, sizeof(buf), "[%^=*#6n] [%=*#6i]",
        1234.567, 1234.567);
```

outputs

```
[€ **1234,57] [EUR **1 234,57]
```

in the *nl_NL* locale. The *de_DE*, *de_CH*, *en_AU*, and *en_GB* locales yield

```
[ **1234,57 €] [ **1.234,57 EUR]
[ Fr. **1234.57] [ CHF **1'234.57]
[ $**1234.57] [ AUD**1,234.57]
[ £**1234.57] [ GBP**1,234.57]
```

SEE ALSO

duplocale(3), *setlocale*(3), *sprintf*(3), *locale*(7)

NAME

strfromd, strfromf, strfroml − convert a floating-point value into a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

int strfromd(char str[restrict .n], size_t n,
        const char *restrict format, double fp);
int strfromf(char str[restrict .n], size_t n,
        const char *restrict format, float fp);
int strfroml(char str[restrict .n], size_t n,
        const char *restrict format, long double fp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strfromd(), **strfromf**(), **strfroml**():
__STDC_WANT_IEC_60559_BFP_EXT__

DESCRIPTION

These functions convert a floating-point value, *fp*, into a string of characters, *str*, with a configurable *format* string. At most *n* characters are stored into *str*.

The terminating null byte ('\0') is written if and only if *n* is sufficiently large, otherwise the written string is truncated at *n* characters.

The **strfromd**(), **strfromf**(), and **strfroml**() functions are equivalent to

```
snprintf(str, n, format, fp);
```

except for the *format* string.

Format of the format string

The *format* string must start with the character '%'. This is followed by an optional precision which starts with the period character (.), followed by an optional decimal integer. If no integer is specified after the period character, a precision of zero is used. Finally, the format string should have one of the conversion specifiers **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G**.

The conversion specifier is applied based on the floating-point type indicated by the function suffix. Therefore, unlike **snprintf**(), the format string does not have a length modifier character. See *snprintf*(3) for a detailed description of these conversion specifiers.

The implementation conforms to the C99 standard on conversion of NaN and infinity values:

> If *fp* is a NaN, +NaN, or −NaN, and **f** (or **a**, **e**, **g**) is the conversion specifier, the conversion is to "nan", "nan", or "−nan", respectively. If **F** (or **A**, **E**, **G**) is the conversion specifier, the conversion is to "NAN" or "−NAN".
>
> Likewise if *fp* is infinity, it is converted to [−]inf or [−]INF.

A malformed *format* string results in undefined behavior.

RETURN VALUE

The **strfromd**(), **strfromf**(), and **strfroml**() functions return the number of characters that would have been written in *str* if *n* had enough space, not counting the terminating null byte. Thus, a return value of *n* or greater means that the output was truncated.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7) and the **POSIX Safety Concepts** section in GNU C Library manual.

Interface	Attribute	Value
strfromd(), **strfromf**(), **strfroml**()	Thread safety	MT-Safe locale
	Async-signal safety	AS-Unsafe heap
	Async-cancel safety	AC-Unsafe mem

Note: these attributes are preliminary.

STANDARDS

ISO/IEC TS 18661-1.

VERSIONS

strfromd()
strfromf()
strfroml()
glibc 2.25.

NOTES

These functions take account of the **LC_NUMERIC** category of the current locale.

EXAMPLES

To convert the value 12.1 as a float type to a string using decimal notation, resulting in "12.100000":

```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdlib.h>
int ssize = 10;
char s[ssize];
strfromf(s, ssize, "%f", 12.1);
```

To convert the value 12.3456 as a float type to a string using decimal notation with two digits of precision, resulting in "12.35":

```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdlib.h>
int ssize = 10;
char s[ssize];
strfromf(s, ssize, "%.2f", 12.3456);
```

To convert the value 12.345e19 as a double type to a string using scientific notation with zero digits of precision, resulting in "1E+20":

```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdlib.h>
int ssize = 10;
char s[ssize];
strfromd(s, ssize, "%.E", 12.345e19);
```

SEE ALSO

atof(3), *snprintf*(3), *strtod*(3)

NAME

strfry − randomize a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

char *strfry(char *string);
```

DESCRIPTION

The **strfry**() function randomizes the contents of *string* by randomly swapping characters in the string. The result is an anagram of *string*.

RETURN VALUE

The **strfry**() functions returns a pointer to the randomized string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strfry()	Thread safety	MT-Safe

STANDARDS

GNU.

SEE ALSO

memfrob(3), *string*(3)

NAME

strftime − format date and time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

size_t strftime(char s[restrict .max], size_t max,
                const char *restrict format,
                const struct tm *restrict tm);

size_t strftime_l(char s[restrict .max], size_t max,
                const char *restrict format,
                const struct tm *restrict tm,
                locale_t locale);
```

DESCRIPTION

The **strftime**() function formats the broken-down time *tm* according to the format specification *format* and places the result in the character array *s* of size *max*. The broken-down time structure *tm* is defined in <*time.h*>. See also *ctime*(3).

The format specification is a null-terminated string and may contain special character sequences called *conversion specifications*, each of which is introduced by a '%' character and terminated by some other character known as a *conversion specifier character*. All other character sequences are *ordinary character sequences*.

The characters of ordinary character sequences (including the null byte) are copied verbatim from *format* to *s*. However, the characters of conversion specifications are replaced as shown in the list below. In this list, the field(s) employed from the *tm* structure are also shown.

%a The abbreviated name of the day of the week according to the current locale. (Calculated from *tm_wday*.) (The specific names used in the current locale can be obtained by calling *nl_langinfo*(3) with **ABDAY_**{**1**–**7**} as an argument.)

%A The full name of the day of the week according to the current locale. (Calculated from *tm_wday*.) (The specific names used in the current locale can be obtained by calling *nl_langinfo*(3) with **DAY_**{**1**–**7**} as an argument.)

%b The abbreviated month name according to the current locale. (Calculated from *tm_mon*.) (The specific names used in the current locale can be obtained by calling *nl_langinfo*(3) with **ABMON_**{**1**–**12**} as an argument.)

%B The full month name according to the current locale. (Calculated from *tm_mon*.) (The specific names used in the current locale can be obtained by calling *nl_langinfo*(3) with **MON_**{**1**–**12**} as an argument.)

%c The preferred date and time representation for the current locale. (The specific format used in the current locale can be obtained by calling *nl_langinfo*(3) with **D_T_FMT** as an argument for the **%c** conversion specification, and with **ERA_D_T_FMT** for the **%Ec** conversion specification.) (In the POSIX locale this is equivalent to **%a %b %e %H:%M:%S %Y**.)

%C The century number (year/100) as a 2-digit integer. (SU) (The **%EC** conversion specification corresponds to the name of the era.) (Calculated from *tm_year*.)

%d The day of the month as a decimal number (range 01 to 31). (Calculated from *tm_mday*.)

%D Equivalent to **%m/%d/%y**. (Yecch—for Americans only. Americans should note that in other countries **%d/%m/%y** is rather common. This means that in international context this format is ambiguous and should not be used.) (SU)

%e Like **%d**, the day of the month as a decimal number, but a leading zero is replaced by a space. (SU) (Calculated from *tm_mday*.)

%E Modifier: use alternative ("era-based") format, see below. (SU)

%F Equivalent to **%Y−%m−%d** (the ISO 8601 date format). (C99)

%G The ISO 8601 week-based year (see NOTES) with century as a decimal number. The 4-digit year corresponding to the ISO week number (see **%V**). This has the same format and value as **%Y**, except that if the ISO week number belongs to the previous or next year, that year is used instead. (TZ) (Calculated from *tm_year*, *tm_yday*, and *tm_wday*.)

%g Like **%G**, but without century, that is, with a 2-digit year (00–99). (TZ) (Calculated from *tm_year*, *tm_yday*, and *tm_wday*.)

%h Equivalent to **%b**. (SU)

%H The hour as a decimal number using a 24-hour clock (range 00 to 23). (Calculated from *tm_hour*.)

%I The hour as a decimal number using a 12-hour clock (range 01 to 12). (Calculated from *tm_hour*.)

%j The day of the year as a decimal number (range 001 to 366). (Calculated from *tm_yday*.)

%k The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also **%H**.) (Calculated from *tm_hour*.) (TZ)

%l The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also **%I**.) (Calculated from *tm_hour*.) (TZ)

%m The month as a decimal number (range 01 to 12). (Calculated from *tm_mon*.)

%M The minute as a decimal number (range 00 to 59). (Calculated from *tm_min*.)

%n A newline character. (SU)

%O Modifier: use alternative numeric symbols, see below. (SU)

%p Either "AM" or "PM" according to the given time value, or the corresponding strings for the current locale. Noon is treated as "PM" and midnight as "AM". (Calculated from *tm_hour*.) (The specific string representations used for "AM" and "PM" in the current locale can be obtained by calling *nl_langinfo*(3) with **AM_STR** and **PM_STR**, respectively.)

%P Like **%p** but in lowercase: "am" or "pm" or a corresponding string for the current locale. (Calculated from *tm_hour*.) (GNU)

%r The time in a.m. or p.m. notation. (SU) (The specific format used in the current locale can be obtained by calling *nl_langinfo*(3) with **T_FMT_AMPM** as an argument.) (In the POSIX locale this is equivalent to **%I:%M:%S %p**.)

%R The time in 24-hour notation (**%H:%M**). (SU) For a version including the seconds, see **%T** below.

%s The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). (TZ) (Calculated from *mktime(tm)*.)

%S The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.) (Calculated from *tm_sec*.)

%t A tab character. (SU)

%T The time in 24-hour notation (**%H:%M:%S**). (SU)

%u The day of the week as a decimal, range 1 to 7, Monday being 1. See also **%w**. (Calculated from *tm_wday*.) (SU)

%U The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also **%V** and **%W**. (Calculated from *tm_yday* and *tm_wday*.)

%V The ISO 8601 week number (see NOTES) of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year. See also **%U** and **%W**. (Calculated from *tm_year*, *tm_yday*, and *tm_wday*.) (SU)

%w The day of the week as a decimal, range 0 to 6, Sunday being 0. See also **%u**. (Calculated from *tm_wday*.)

%W The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01. (Calculated from *tm_yday* and *tm_wday*.)

%x The preferred date representation for the current locale without the time. (The specific format used in the current locale can be obtained by calling *nl_langinfo*(3) with **D_FMT** as an argument for the **%x** conversion specification, and with **ERA_D_FMT** for the **%Ex** conversion specification.) (In the POSIX locale this is equivalent to **%m/%d/%y**.)

%X The preferred time representation for the current locale without the date. (The specific format used in the current locale can be obtained by calling *nl_langinfo*(3) with **T_FMT** as an argument for the **%X** conversion specification, and with **ERA_T_FMT** for the **%EX** conversion specification.) (In the POSIX locale this is equivalent to **%H:%M:%S**.)

%y The year as a decimal number without a century (range 00 to 99). (The **%Ey** conversion specification corresponds to the year since the beginning of the era denoted by the **%EC** conversion specification.) (Calculated from *tm_year*)

%Y The year as a decimal number including the century. (The **%EY** conversion specification corresponds to the full alternative year representation.) (Calculated from *tm_year*)

%z The *+hhmm* or *−hhmm* numeric timezone (that is, the hour and minute offset from UTC). (SU)

%Z The timezone name or abbreviation.

%+ The date and time in *date*(1) format. (TZ) (Not supported in glibc2.)

%% A literal '%' character.

Some conversion specifications can be modified by preceding the conversion specifier character by the **E** or **O** *modifier* to indicate that an alternative format should be used. If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used. (SU) The Single UNIX Specification mentions **%Ec**, **%EC**, **%Ex**, **%EX**, **%Ey**, **%EY**, **%Od**, **%Oe**, **%OH**, **%OI**, **%Om**, **%OM**, **%OS**, **%Ou**, **%OU**, **%OV**, **%Ow**, **%OW**, **%Oy**, where the effect of the **O** modifier is to use alternative numeric symbols (say, roman numerals), and that of the **E** modifier is to use a locale-dependent alternative representation. The rules governing date representation with the **E** modifier can be obtained by supplying **ERA** as an argument to a *nl_langinfo*(3). One example of such alternative forms is the Japanese era calendar scheme in the **ja_JP** glibc locale.

strftime_l() is equivalent to **strftime**(), except it uses the specified *locale* instead of the current locale. The behaviour is undefined if *locale* is invalid or **LC_GLOBAL_LOCALE**.

RETURN VALUE

Provided that the result string, including the terminating null byte, does not exceed *max* bytes, **strftime**() returns the number of bytes (excluding the terminating null byte) placed in the array *s*. If the length of the result string (including the terminating null byte) would exceed *max* bytes, then **strftime**() returns 0, and the contents of the array are undefined.

Note that the return value 0 does not necessarily indicate an error. For example, in many locales **%p** yields an empty string. An empty *format* string will likewise yield an empty string.

ENVIRONMENT

The environment variables **TZ** and **LC_TIME** are used.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strftime(), **strftime_l**()	Thread safety	MT-Safe env locale

STANDARDS

strftime()
C11, POSIX.1-2008.

strftime_l()
POSIX.1-2008.

HISTORY

strftime()
SVr4, C89.

strftime_l()
POSIX.1-2008.

There are strict inclusions between the set of conversions given in ANSI C (unmarked), those given in the Single UNIX Specification (marked SU), those given in Olson's timezone package (marked TZ), and those given in glibc (marked GNU), except that **%+** is not supported in glibc2. On the other hand glibc2 has several more extensions. POSIX.1 only refers to ANSI C; POSIX.2 describes under *date*(1) several extensions that could apply to **strftime**() as well. The **%F** conversion is in C99 and POSIX.1-2001.

In SUSv2, the **%S** specifier allowed a range of 00 to 61, to allow for the theoretical possibility of a minute that included a double leap second (there never has been such a minute).

NOTES

ISO 8601 week dates

%G, **%g**, and **%V** yield values calculated from the week-based year defined by the ISO 8601 standard. In this system, weeks start on a Monday, and are numbered from 01, for the first week, up to 52 or 53, for the last week. Week 1 is the first week where four or more days fall within the new year (or, synonymously, week 01 is: the first week of the year that contains a Thursday; or, the week that has 4 January in it). When three or fewer days of the first calendar week of the new year fall within that year, then the ISO 8601 week-based system counts those days as part of week 52 or 53 of the preceding year. For example, 1 January 2010 is a Friday, meaning that just three days of that calendar week fall in 2010. Thus, the ISO 8601 week-based system considers these days to be part of week 53 (**%V**) of the year 2009 (**%G**); week 01 of ISO 8601 year 2010 starts on Monday, 4 January 2010. Similarly, the first two days of January 2011 are considered to be part of week 52 of the year 2010.

glibc notes

glibc provides some extensions for conversion specifications. (These extensions are not specified in POSIX.1-2001, but a few other systems provide similar features.) Between the '%' character and the conversion specifier character, an optional *flag* and field *width* may be specified. (These precede the **E** or **O** modifiers, if present.)

The following flag characters are permitted:

_ (underscore) Pad a numeric result string with spaces.

- (dash) Do not pad a numeric result string.

0 Pad a numeric result string with zeros even if the conversion specifier character uses space-padding by default.

^ Convert alphabetic characters in result string to uppercase.

Swap the case of the result string. (This flag works only with certain conversion specifier characters, and of these, it is only really useful with **%Z**.)

An optional decimal width specifier may follow the (possibly absent) flag. If the natural size of the field is smaller than this width, then the result string is padded (on the left) to the specified width.

BUGS

If the output string would exceed *max* bytes, *errno* is *not* set. This makes it impossible to distinguish this error case from cases where the *format* string legitimately produces a zero-length output string. POSIX.1-2001 does *not* specify any *errno* settings for **strftime**().

Some buggy versions of *gcc*(1) complain about the use of **%c**: *warning: `%c' yields only last 2 digits of year in some locales*. Of course programmers are encouraged to use **%c**, as it gives the preferred date and time representation. One meets all kinds of strange obfuscations to circumvent this *gcc*(1) problem. A relatively clean one is to add an intermediate function

```
size_t
my_strftime(char *s, size_t max, const char *fmt,
            const struct tm *tm)
{
    return strftime(s, max, fmt, tm);
```

```
}
```

Nowadays, *gcc*(1) provides the *−Wno−format−y2k* option to prevent the warning, so that the above workaround is no longer required.

EXAMPLES

RFC 2822-compliant date format (with an English locale for %a and %b)

```
"%a, %d %b %Y %T %z"
```

RFC 822-compliant date format (with an English locale for %a and %b)

```
"%a, %d %b %y %T %z"
```

Example program

The program below can be used to experiment with **strftime**().

Some examples of the result string produced by the glibc implementation of **strftime**() are as follows:

```
$ ./a.out '%m'
Result string is "11"
$ ./a.out '%5m'
Result string is "00011"
$ ./a.out '%_5m'
Result string is "   11"
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int
main(int argc, char *argv[])
{
    char outstr[200];
    time_t t;
    struct tm *tmp;

    t = time(NULL);
    tmp = localtime(&t);
    if (tmp == NULL) {
        perror("localtime");
        exit(EXIT_FAILURE);
    }

    if (strftime(outstr, sizeof(outstr), argv[1], tmp) == 0) {
        fprintf(stderr, "strftime returned 0");
        exit(EXIT_FAILURE);
    }

    printf("Result string is \"%s\"\n", outstr);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

date(1), *time*(2), *ctime*(3), *nl_langinfo*(3), *setlocale*(3), *sprintf*(3), *strptime*(3)

NAME

stpcpy, strcasecmp, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strdup, strfry, strlen, strncat, strncmp, strncpy, strncasecmp, strpbrk, strrchr, strsep, strspn, strstr, strtok, strxfrm, index, rindex − string operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <strings.h>

int strcasecmp(const char ****s1*, const char ****s2***);**
Compare the strings *s1* and *s2* ignoring case.

int strncasecmp(const char *s1***[.***n***], const char** *s2***[.***n***], size_t** *n***);**
Compare the first *n* bytes of the strings *s1* and *s2* ignoring case.

char *index(const char ****s*, int** *c***);**
Identical to *strchr*(3).

char *rindex(const char ****s*, int** *c***);**
Identical to *strrchr*(3).

#include <string.h>

char *stpcpy(char *restrict *dest***, const char *restrict** *src***);**
Copy a string from *src* to *dest*, returning a pointer to the end of the resulting string at *dest*.

char *strcat(char *restrict *dest***, const char *restrict** *src***);**
Append the string *src* to the string *dest*, returning a pointer *dest*.

char *strchr(const char ****s*, int** *c***);**
Return a pointer to the first occurrence of the character *c* in the string *s*.

int strcmp(const char ****s1*, const char ****s2***);**
Compare the strings *s1* with *s2*.

int strcoll(const char ****s1*, const char ****s2***);**
Compare the strings *s1* with *s2* using the current locale.

char *strcpy(char *restrict *dest***, const char *restrict** *src***);**
Copy the string *src* to *dest*, returning a pointer to the start of *dest*.

size_t strcspn(const char ****s*, const char ****reject***);**
Calculate the length of the initial segment of the string *s* which does not contain any of bytes in the string *reject*,

char *strdup(const char ****s*);**
Return a duplicate of the string *s* in memory allocated using *malloc*(3).

char *strfry(char ****string*);**
Randomly swap the characters in *string*.

size_t strlen(const char ****s*);**
Return the length of the string *s*.

char *strncat(char *dest***[restrict strlen(.***dest***) + .***n* **+ 1],**
const char *src***[restrict .***n***],**
size_t *n***);**
Append at most *n* bytes from the unterminated string *src* to the string *dest*, returning a pointer to *dest*.

int strncmp(const char *s1***[.***n***], const char** *s2***[.***n***], size_t** *n***);**
Compare at most *n* bytes of the strings *s1* and *s2*.

char *strpbrk(const char ****s*, const char ****accept***);**
Return a pointer to the first occurrence in the string *s* of one of the bytes in the string *accept*.

char *strrchr(const char ****s*, int** *c***);**
Return a pointer to the last occurrence of the character *c* in the string *s*.

char *strsep(char **restrict *stringp***, const char *restrict** *delim***);**
Extract the initial token in *stringp* that is delimited by one of the bytes in *delim*.

size_t strspn(const char * *s***, const char *** *accept***);**
Calculate the length of the starting segment in the string *s* that consists entirely of bytes in *accept*.

char *strstr(const char * *haystack***, const char *** *needle***);**
Find the first occurrence of the substring *needle* in the string *haystack*, returning a pointer to the found substring.

char *strtok(char *restrict *s***, const char *restrict** *delim***);**
Extract tokens from the string *s* that are delimited by one of the bytes in *delim*.

size_t strxfrm(char *dest***[restrict .**n**], const char** *src***[restrict .**n**],**
size_t *n***);**
Transforms *src* to the current locale and copies the first *n* bytes to *dest*.

char *strncpy(char *dest***[restrict .**n**], const char** *src***[restrict .**n**],**
size_t *n***);**
Fill a fixed-size buffer with leading non-null bytes from a source array, padding with null bytes as needed.

DESCRIPTION

The string functions perform operations on null-terminated strings. See the individual man pages for descriptions of each function.

SEE ALSO

bstring(3), *stpcpy*(3), *strcasecmp*(3), *strcat*(3), *strchr*(3), *strcmp*(3), *strcoll*(3), *strcpy*(3), *strcspn*(3), *strdup*(3), *strfry*(3), *strlen*(3), *strncasecmp*(3), *strncat*(3), *strncmp*(3), *strncpy*(3), *strpbrk*(3), *strrchr*(3), *strsep*(3), *strspn*(3), *strstr*(3), *strtok*(3), *strxfrm*(3)

NAME

strlen − calculate the length of a string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

size_t strlen(const char *s);

DESCRIPTION

The **strlen**() function calculates the length of the string pointed to by *s*, excluding the terminating null byte ('\0').

RETURN VALUE

The **strlen**() function returns the number of bytes in the string pointed to by *s*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strlen()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

NOTES

In cases where the input buffer may not contain a terminating null byte, *strnlen*(3) should be used instead.

SEE ALSO

string(3), *strnlen*(3), *wcslen*(3), *wcsnlen*(3)

NAME

strncat − append non-null bytes from a source array to a string, and null-terminate the result

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

char *strncat(char *restrict *dst***, const char** *src***[restrict .**_ssize_**],**
size_t *ssize***);**

DESCRIPTION

This function appends at most *ssize* non-null bytes from the array pointed to by *src*, followed by a null character, to the end of the string pointed to by *dst*. *dst* must point to a string contained in a buffer that is large enough, that is, the buffer size must be at least *strlen(dst) + strnlen(src, ssize) + 1*.

An implementation of this function might be:

```
char *
strncat(char *restrict dst, const char *restrict src, size_t ssize)
{
    #define strnul(s)  (s + strlen(s))

    stpcpy(mempcpy(strnul(dst), src, strnlen(src, ssize)), "");
    return dst;
}
```

RETURN VALUE

strncat() returns *dst*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strncat()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

CAVEATS

The name of this function is confusing; it has no relation to *strncpy*(3).

If the destination buffer does not already contain a string, or is not large enough, the behavior is undefined. See **_FORTIFY_SOURCE** in *feature_test_macros*(7).

BUGS

This function can be very inefficient. Read about Shlemiel the painter.

EXAMPLES

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define nitems(arr)  (sizeof((arr)) / sizeof((arr)[0]))

int
main(void)
{
    size_t  n;

    // Null-padded fixed-size character sequences
    char    pre[4] = "pre.";
    char    new_post[50] = ".foo.bar";
```

```
        // Strings
        char    post[] = ".post";
        char    src[] = "some_long_body.post";
        char    *dest;

        n = nitems(pre) + strlen(src) - strlen(post) + nitems(new_post) + 1;
        dest = malloc(sizeof(*dest) * n);
        if (dest == NULL)
            err(EXIT_FAILURE, "malloc()");

        dest[0] = '\0';  // There's no 'cpy' function to this 'cat'.
        strncat(dest, pre, nitems(pre));
        strncat(dest, src, strlen(src) - strlen(post));
        strncat(dest, new_post, nitems(new_post));

        puts(dest);  // "pre.some_long_body.foo.bar"
        free(dest);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

string(3), *string_copying*(7)

NAME

strnlen − determine the length of a fixed-size string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

size_t strnlen(const char *s***[.***maxlen***], size_t** *maxlen***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strnlen():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **strnlen**() function returns the number of bytes in the string pointed to by *s*, excluding the terminating null byte ('\0'), but at most *maxlen*. In doing this, **strnlen**() looks only at the first *maxlen* characters in the string pointed to by *s* and never beyond *s[maxlen−1]*.

RETURN VALUE

The **strnlen**() function returns *strlen(s)*, if that is less than *maxlen*, or *maxlen* if there is no null terminating ('\0') among the first *maxlen* characters pointed to by *s*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strnlen()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2008.

SEE ALSO

strlen(3)

NAME

strpbrk − search a string for any of a set of bytes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

char *strpbrk(const char *s, const char *accept);

DESCRIPTION

The **strpbrk**() function locates the first occurrence in the string *s* of any of the bytes in the string *accept*.

RETURN VALUE

The **strpbrk**() function returns a pointer to the byte in *s* that matches one of the bytes in *accept*, or NULL if no such byte is found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strpbrk()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

memchr(3), *strchr*(3), *string*(3), *strsep*(3), *strspn*(3), *strstr*(3), *strtok*(3), *wcspbrk*(3)

NAME

strptime − convert a string representation of time to a time tm structure

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE       /* See feature_test_macros(7) */
#include <time.h>

char *strptime(const char *restrict s, const char *restrict format,
               struct tm *restrict tm);
```

DESCRIPTION

The **strptime**() function is the converse of *strftime*(3); it converts the character string pointed to by *s* to values which are stored in the "broken-down time" structure pointed to by *tm*, using the format specified by *format*.

The broken-down time structure *tm* is described in *tm*(3type).

The *format* argument is a character string that consists of field descriptors and text characters, reminiscent of *scanf*(3). Each field descriptor consists of a **%** character followed by another character that specifies the replacement for the field descriptor. All other characters in the *format* string must have a matching character in the input string, except for whitespace, which matches zero or more whitespace characters in the input string. There should be whitespace or other alphanumeric characters between any two field descriptors.

The **strptime**() function processes the input string from left to right. Each of the three possible input elements (whitespace, literal, or format) are handled one after the other. If the input cannot be matched to the format string, the function stops. The remainder of the format and input strings are not processed.

The supported input field descriptors are listed below. In case a text string (such as the name of a day of the week or a month name) is to be matched, the comparison is case insensitive. In case a number is to be matched, leading zeros are permitted but not required.

%% The **%** character.

%a or **%A**
The name of the day of the week according to the current locale, in abbreviated form or the full name.

%b or **%B** or **%h**
The month name according to the current locale, in abbreviated form or the full name.

%c The date and time representation for the current locale.

%C The century number (0–99).

%d or **%e**
The day of month (1–31).

%D Equivalent to **%m/%d/%y**. (This is the American style date, very confusing to non-Americans, especially since **%d/%m/%y** is widely used in Europe. The ISO 8601 standard format is **%Y-%m-%d**.)

%H The hour (0–23).

%I The hour on a 12-hour clock (1–12).

%j The day number in the year (1–366).

%m The month number (1–12).

%M The minute (0–59).

%n Arbitrary whitespace.

%p The locale’s equivalent of AM or PM. (Note: there may be none.)

%r The 12-hour clock time (using the locale's AM or PM). In the POSIX locale equivalent to **%I:%M:%S %p**. If *t_fmt_ampm* is empty in the **LC_TIME** part of the current locale, then the behavior is undefined.

%R Equivalent to **%H:%M**.

%S The second (0–60; 60 may occur for leap seconds; earlier also 61 was allowed).

%t Arbitrary whitespace.

%T Equivalent to **%H:%M:%S**.

%U The week number with Sunday the first day of the week (0–53). The first Sunday of January is the first day of week 1.

%w The ordinal number of the day of the week (0–6), with Sunday = 0.

%W The week number with Monday the first day of the week (0–53). The first Monday of January is the first day of week 1.

%x The date, using the locale's date format.

%X The time, using the locale's time format.

%y The year within century (0–99). When a century is not otherwise specified, values in the range 69–99 refer to years in the twentieth century (1969–1999); values in the range 00–68 refer to years in the twenty-first century (2000–2068).

%Y The year, including century (for example, 1991).

Some field descriptors can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used. If the alternative format or specification does not exist in the current locale, the unmodified field descriptor is used.

The E modifier specifies that the input string may contain alternative locale-dependent versions of the date and time representation:

%Ec The locale's alternative date and time representation.

%EC The name of the base year (period) in the locale's alternative representation.

%Ex The locale's alternative date representation.

%EX The locale's alternative time representation.

%Ey The offset from **%EC** (year only) in the locale's alternative representation.

%EY The full alternative year representation.

The O modifier specifies that the numerical input may be in an alternative locale-dependent format:

%Od or **%Oe**
The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.

%OH The hour (24-hour clock) using the locale's alternative numeric symbols.

%OI The hour (12-hour clock) using the locale's alternative numeric symbols.

%Om The month using the locale's alternative numeric symbols.

%OM The minutes using the locale's alternative numeric symbols.

%OS The seconds using the locale's alternative numeric symbols.

%OU The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

%Ow The ordinal number of the day of the week (Sunday=0), using the locale's alternative numeric symbols.

%OW The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy The year (offset from **%C**) using the locale's alternative numeric symbols.

RETURN VALUE

The return value of the function is a pointer to the first character not processed in this function call. In case the input string contains more characters than required by the format string, the return value points right after the last consumed input character. In case the whole input string is consumed, the return value points to the null byte at the end of the string. If **strptime**() fails to match all of the format string and therefore an error occurred, the function returns NULL.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strptime()	Thread safety	MT-Safe env locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SUSv2.

NOTES

In principle, this function does not initialize *tm* but stores only the values specified. This means that *tm* should be initialized before the call. Details differ a bit between different UNIX systems. The glibc implementation does not touch those fields which are not explicitly specified, except that it recomputes the *tm_wday* and *tm_yday* field if any of the year, month, or day elements changed.

The 'y' (year in century) specification is taken to specify a year in the range 1950–2049 by glibc 2.0. It is taken to be a year in 1969–2068 since glibc 2.1.

glibc notes

For reasons of symmetry, glibc tries to support for **strptime**() the same format characters as for *strftime*(3). (In most cases, the corresponding fields are parsed, but no field in *tm* is changed.) This leads to

%F Equivalent to **%Y-%m-%d**, the ISO 8601 date format.

%g The year corresponding to the ISO week number, but without the century (0–99).

%G The year corresponding to the ISO week number. (For example, 1991.)

%u The day of the week as a decimal number (1–7, where Monday = 1).

%V The ISO 8601:1988 week number as a decimal number (1–53). If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1.

%z An RFC-822/ISO 8601 standard timezone specification.

%Z The timezone name.

Similarly, because of GNU extensions to *strftime*(3), **%k** is accepted as a synonym for **%H**, and **%l** should be accepted as a synonym for **%I**, and **%P** is accepted as a synonym for **%p**. Finally

%s The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). Leap seconds are not counted unless leap second support is available.

The glibc implementation does not require whitespace between two field descriptors.

EXAMPLES

The following example demonstrates the use of **strptime**() and *strftime*(3).

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int
main(void)
{
    struct tm tm;
```

```
        char buf[255];

        memset(&tm, 0, sizeof(tm));
        strptime("2001-11-12 18:31:01", "%Y-%m-%d %H:%M:%S", &tm);
        strftime(buf, sizeof(buf), "%d %b %Y %H:%M", &tm);
        puts(buf);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

time(2), *getdate*(3), *scanf*(3), *setlocale*(3), *strftime*(3)

NAME

strsep − extract token from string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

char *strsep(char **restrict *stringp***, const char *restrict** *delim***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strsep():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

If **stringp* is NULL, the **strsep**() function returns NULL and does nothing else. Otherwise, this function finds the first token in the string **stringp* that is delimited by one of the bytes in the string *delim*. This token is terminated by overwriting the delimiter with a null byte ('\0'), and **stringp* is updated to point past the token. In case no delimiter was found, the token is taken to be the entire string **stringp*, and **stringp* is made NULL.

RETURN VALUE

The **strsep**() function returns a pointer to the token, that is, it returns the original value of **stringp*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strsep()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.4BSD.

The **strsep**() function was introduced as a replacement for *strtok*(3), since the latter cannot handle empty fields. However, *strtok*(3) conforms to C89/C99 and hence is more portable.

BUGS

Be cautious when using this function. If you do use it, note that:

- This function modifies its first argument.
- This function cannot be used on constant strings.
- The identity of the delimiting character is lost.

EXAMPLES

The program below is a port of the one found in *strtok*(3), which, however, doesn't discard multiple delimiters or empty tokens:

```
$ ./a.out 'a/bbb///cc;xxx:yyy:' ':;' '/'
1: a/bbb///cc
        --> a
        --> bbb
        -->
        -->
        --> cc
2: xxx
        --> xxx
3: yyy
        --> yyy
4:
        -->
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    char *token, *subtoken;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s string delim subdelim\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    for (unsigned int j = 1; (token = strsep(&argv[1], argv[2])); j++) {
        printf("%u: %s\n", j, token);

        while ((subtoken = strsep(&token, argv[3])))
            printf("\t --> %s\n", subtoken);
    }

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

memchr(3), *strchr*(3), *string*(3), *strpbrk*(3), *strspn*(3), *strstr*(3), *strtok*(3)

NAME

strsignal, sigabbrev_np, sigdescr_np, sys_siglist − return string describing signal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strsignal(int sig);
const char *sigdescr_np(int sig);
const char *sigabbrev_np(int sig);

[[deprecated]] extern const char *const sys_siglist[];
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

sigabbrev_np(), **sigdescr_np**():
_GNU_SOURCE

strsignal():
From glibc 2.10 to glibc 2.31:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

sys_siglist:
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The **strsignal**() function returns a string describing the signal number passed in the argument *sig*. The string can be used only until the next call to **strsignal**(). The string returned by **strsignal**() is localized according to the **LC_MESSAGES** category in the current locale.

The **sigdescr_np**() function returns a string describing the signal number passed in the argument *sig*. Unlike **strsignal**() this string is not influenced by the current locale.

The **sigabbrev_np**() function returns the abbreviated name of the signal, *sig*. For example, given the value **SIGINT**, it returns the string "INT".

The (deprecated) array *sys_siglist* holds the signal description strings indexed by signal number. The **strsignal**() or the **sigdescr_np**() function should be used instead of this array; see also VERSIONS.

RETURN VALUE

The **strsignal**() function returns the appropriate description string, or an unknown signal message if the signal number is invalid. On some systems (but not on Linux), NULL may instead be returned for an invalid signal number.

The **sigdescr_np**() and **sigabbrev_np**() functions return the appropriate description string. The returned string is statically allocated and valid for the lifetime of the program. These functions return NULL for an invalid signal number.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strsignal()	Thread safety	MT-Unsafe race:strsignal locale
sigdescr_np(), **sigabbrev_np**()	Thread safety	MT-Safe

STANDARDS

strsignal()
POSIX.1-2008.

sigdescr_np()
sigabbrev_np()
GNU.

sys_siglist
None.

HISTORY

strsignal()
POSIX.1-2008. Solaris, BSD.

sigdescr_np()
sigabbrev_np()
glibc 2.32.

sys_siglist
Removed in glibc 2.32.

NOTES

sigdescr_np() and **sigabbrev_np**() are thread-safe and async-signal-safe.

SEE ALSO

psignal(3), *strerror*(3)

NAME

strspn, strcspn − get length of a prefix substring

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

size_t strspn(const char *_s_, const char *_accept_);
size_t strcspn(const char *_s_, const char *_reject_);

DESCRIPTION

The **strspn**() function calculates the length (in bytes) of the initial segment of *s* which consists entirely of bytes in *accept*.

The **strcspn**() function calculates the length of the initial segment of *s* which consists entirely of bytes not in *reject*.

RETURN VALUE

The **strspn**() function returns the number of bytes in the initial segment of *s* which consist only of bytes from *accept*.

The **strcspn**() function returns the number of bytes in the initial segment of *s* which are not in the string *reject*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strspn(), **strcspn**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

memchr(3), *strchr*(3), *string*(3), *strpbrk*(3), *strsep*(3), *strstr*(3), *strtok*(3), *wcscspn*(3), *wcsspn*(3)

NAME

strstr, strcasestr − locate a substring

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <string.h>

char *strcasestr(const char *haystack, const char *needle);
```

DESCRIPTION

The **strstr**() function finds the first occurrence of the substring *needle* in the string *haystack*. The terminating null bytes ('\0') are not compared.

The **strcasestr**() function is like **strstr**(), but ignores the case of both arguments.

RETURN VALUE

These functions return a pointer to the beginning of the located substring, or NULL if the substring is not found.

If *needle* is the empty string, the return value is always *haystack* itself.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strstr()	Thread safety	MT-Safe
strcasestr()	Thread safety	MT-Safe locale

STANDARDS

strstr() C11, POSIX.1-2008.

strcasestr()
GNU.

HISTORY

strstr() POSIX.1-2001, C89.

strcasestr()
GNU.

SEE ALSO

memchr(3), *memmem*(3), *strcasecmp*(3), *strchr*(3), *string*(3), *strpbrk*(3), *strsep*(3), *strspn*(3), *strtok*(3), *wcsstr*(3)

NAME

strtod, strtof, strtold − convert ASCII string to floating-point number

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strtof(), **strtold**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **strtod**(), **strtof**(), and **strtold**() functions convert the initial portion of the string pointed to by *nptr* to *double*, *float*, and *long double* representation, respectively.

The expected form of the (initial portion of the) string is optional leading white space as recognized by *isspace*(3), an optional plus ('+') or minus sign ('−') and then either (i) a decimal number, or (ii) a hexadecimal number, or (iii) an infinity, or (iv) a NAN (not-a-number).

A *decimal number* consists of a nonempty sequence of decimal digits possibly containing a radix character (decimal point, locale-dependent, usually '.'), optionally followed by a decimal exponent. A decimal exponent consists of an 'E' or 'e', followed by an optional plus or minus sign, followed by a nonempty sequence of decimal digits, and indicates multiplication by a power of 10.

A *hexadecimal number* consists of a "0x" or "0X" followed by a nonempty sequence of hexadecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a 'P' or 'p', followed by an optional plus or minus sign, followed by a nonempty sequence of decimal digits, and indicates multiplication by a power of 2. At least one of radix character and binary exponent must be present.

An *infinity* is either "INF" or "INFINITY", disregarding case.

A *NAN* is "NAN" (disregarding case) optionally followed by a string, *(n-char-sequence)*, where *n-char-sequence* specifies in an implementation-dependent way the type of NAN (see NOTES).

RETURN VALUE

These functions return the converted value, if any.

If *endptr* is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and (unless *endptr* is null) the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and sign of the value), and **ERANGE** is stored in *errno*.

If the correct value would cause underflow, a value with magnitude no larger than **DBL_MIN**, **FLT_MIN**, or **LDBL_MIN** is returned and **ERANGE** is stored in *errno*.

ERRORS

ERANGE
Overflow or underflow occurred.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strtod(), **strtof**(), **strtold**()	Thread safety	MT-Safe locale

VERSIONS

In the glibc implementation, the *n-char-sequence* that optionally follows "NAN" is interpreted as an integer number (with an optional ’0’ or ’0x’ prefix to select base 8 or 16) that is to be placed in the

mantissa component of the returned value.

STANDARDS

C11, POSIX.1-2008.

HISTORY

strtod()

C89, POSIX.1-2001.

strtof()
strtold()

C99, POSIX.1-2001.

NOTES

Since 0 can legitimately be returned on both success and failure, the calling program should set *errno* to 0 before the call, and then determine if an error occurred by checking whether *errno* has a nonzero value after the call.

EXAMPLES

See the example on the *strtol*(3) manual page; the use of the functions described in this manual page is similar.

SEE ALSO

atof(3), *atoi*(3), *atol*(3), *nan*(3), *nanf*(3), *nanl*(3), *strfromd*(3), *strtol*(3), *strtoul*(3)

NAME

strtoimax, strtoumax − convert string to integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
                   int base);
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
                    int base);
```

DESCRIPTION

These functions are just like *strtol*(3) and *strtoul*(3), except that they return a value of type *intmax_t* and *uintmax_t*, respectively.

RETURN VALUE

On success, the converted value is returned. If nothing was found to convert, zero is returned. On overflow or underflow **INTMAX_MAX** or **INTMAX_MIN** or **UINTMAX_MAX** is returned, and *errno* is set to **ERANGE**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strtoimax(), **strtoumax**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

imaxabs(3), *imaxdiv*(3), *strtol*(3), *strtoul*(3), *wcstoimax*(3)

NAME

strtok, strtok_r − extract tokens from strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict str, const char *restrict delim);
char *strtok_r(char *restrict str, const char *restrict delim,
        char **restrict saveptr);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strtok_r():
  _POSIX_C_SOURCE
    || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok**(), the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok**() returns NULL.

A sequence of calls to **strtok**() that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to **strtok**() sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok**() returns NULL. (A string that is empty or that contains only delimiters will thus cause **strtok**() to return NULL on the first call.)

The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok**() saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok**() returns a pointer to the start of the found token.

From the above description, it follows that a sequence of two or more contiguous delimiter bytes in the parsed string is considered to be a single delimiter, and that delimiter bytes at the start or end of the string are ignored. Put another way: the tokens returned by **strtok**() are always nonempty strings. Thus, for example, given the string "*aaa;;bbb,*", successive calls to **strtok**() that specify the delimiter string "*;*," would return the strings "*aaa*" and "*bbb*", and then a null pointer.

The **strtok_r**() function is a reentrant version of **strtok**(). The *saveptr* argument is a pointer to a *char* * variable that is used internally by **strtok_r**() in order to maintain context between successive calls that parse the same string.

On the first call to **strtok_r**(), *str* should point to the string to be parsed, and the value of **saveptr* is ignored (but see NOTES). In subsequent calls, *str* should be NULL, and *saveptr* (and the buffer that it points to) should be unchanged since the previous call.

Different strings may be parsed concurrently using sequences of calls to **strtok_r**() that specify different *saveptr* arguments.

RETURN VALUE

The **strtok**() and **strtok_r**() functions return a pointer to the next token, or NULL if there are no more tokens.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strtok()	Thread safety	MT-Unsafe race:strtok
strtok_r()	Thread safety	MT-Safe

VERSIONS

On some implementations, **saveptr* is required to be NULL on the first call to **strtok_r**() that is being used to parse *str*.

STANDARDS

strtok()
: C11, POSIX.1-2008.

strtok_r()
: POSIX.1-2008.

HISTORY

strtok()
: POSIX.1-2001, C89, SVr4, 4.3BSD.

strtok_r()
: POSIX.1-2001.

BUGS

Be cautious when using these functions. If you do use them, note that:

- These functions modify their first argument.
- These functions cannot be used on constant strings.
- The identity of the delimiting byte is lost.
- The **strtok**() function uses a static buffer while parsing, so it's not thread safe. Use **strtok_r**() if this matters to you.

EXAMPLES

The program below uses nested loops that employ **strtok_r**() to break a string into a two-level hierarchy of tokens. The first command-line argument specifies the string to be parsed. The second argument specifies the delimiter byte(s) to be used to separate that string into "major" tokens. The third argument specifies the delimiter byte(s) to be used to separate the "major" tokens into subtokens.

An example of the output produced by this program is the following:

```
$ ./a.out 'a/bbb///cc;xxx:yyy:' ':;' '/'
1: a/bbb///cc
        --> a
        --> bbb
        --> cc
2: xxx
        --> xxx
3: yyy
        --> yyy
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    char *str1, *str2, *token, *subtoken;
    char *saveptr1, *saveptr2;
    int j;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s string delim subdelim\n",
```

```
                argv[0]);
        exit(EXIT_FAILURE);
    }

    for (j = 1, str1 = argv[1]; ; j++, str1 = NULL) {
        token = strtok_r(str1, argv[2], &saveptr1);
        if (token == NULL)
            break;
        printf("%d: %s\n", j, token);

        for (str2 = token; ; str2 = NULL) {
            subtoken = strtok_r(str2, argv[3], &saveptr2);
            if (subtoken == NULL)
                break;
            printf("\t --> %s\n", subtoken);
        }
    }

    exit(EXIT_SUCCESS);
}
```

Another example program using **strtok**() can be found in *getaddrinfo_a*(3).

SEE ALSO

memchr(3), *strchr*(3), *string*(3), *strpbrk*(3), *strsep*(3), *strspn*(3), *strstr*(3), *wcstok*(3)

NAME

strtol, strtoll, strtoq − convert a string to a long integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *restrict nptr,
        char **restrict endptr, int base);
long long strtoll(const char *restrict nptr,
        char **restrict endptr, int base);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strtoll():
_ISOC99_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

The **strtol**() function converts the initial part of the string in *nptr* to a long integer value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by *isspace*(3)) followed by a single optional '+' or '−' sign. If *base* is zero or 16, the string may then include a "0x" or "0X" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a *long* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either uppercase or lowercase represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, and the *base* is supported, **strtol**() stores the address of the first invalid character in **endptr*. If there were no digits at all, **strtol**() stores the original value of *nptr* in **endptr* (and returns 0). In particular, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string is valid.

The **strtoll**() function works just like the **strtol**() function but returns a *long long* integer value.

RETURN VALUE

The **strtol**() function returns the result of the conversion, unless the value would underflow or overflow. If an underflow occurs, **strtol**() returns **LONG_MIN**. If an overflow occurs, **strtol**() returns **LONG_MAX**. In both cases, *errno* is set to **ERANGE**. Precisely the same holds for **strtoll**() (with **LLONG_MIN** and **LLONG_MAX** instead of **LONG_MIN** and **LONG_MAX**).

ERRORS

This function does not modify *errno* on success.

EINVAL
(not in C99) The given *base* contains an unsupported value.

ERANGE
The resulting value was out of range.

The implementation may also set *errno* to **EINVAL** in case no conversion was performed (no digits seen, and 0 returned).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strtol(), **strtoll**(), **strtoq**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

strtol() POSIX.1-2001, C89, SVr4, 4.3BSD.

strtoll()
POSIX.1-2001, C99.

NOTES

Since **strtol**() can legitimately return 0, **LONG_MAX**, or **LONG_MIN** (**LLONG_MAX** or **LLONG_MIN** for *strtoll*()) on both success and failure, the calling program should set *errno* to 0 before the call, and then determine if an error occurred by checking whether *errno* == *ERANGE* after the call.

According to POSIX.1, in locales other than "C" and "POSIX", these functions may accept other, implementation-defined numeric strings.

BSD also has

```
quad_t strtoq(const char *nptr, char **endptr, int base);
```

with completely analogous definition. Depending on the wordsize of the current architecture, this may be equivalent to **strtoll**() or to **strtol**().

CAVEATS

If the *base* needs to be tested, it should be tested in a call where the string is known to succeed. Otherwise, it's impossible to portably differentiate the errors.

```
errno = 0;
strtol("0", NULL, base);
if (errno == EINVAL)
    goto unsupported_base;
```

EXAMPLES

The program shown below demonstrates the use of **strtol**(). The first command-line argument specifies a string from which **strtol**() should parse a number. The second (optional) argument specifies the base to be used for the conversion. (This argument is converted to numeric form using *atoi*(3), a function that performs no error checking and has a simpler interface than *strtol*().) Some examples of the results produced by this program are the following:

```
$ ./a.out 123
strtol() returned 123
$ ./a.out '    123'
strtol() returned 123
$ ./a.out 123abc
strtol() returned 123
Further characters after number: "abc"
$ ./a.out 123abc 55
strtol: Invalid argument
$ ./a.out ''
No digits were found
$ ./a.out 4000000000
strtol: Numerical result out of range
```

Program source

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int base;
    char *endptr, *str;
    long val;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s str [base]\n", argv[0]);
        exit(EXIT_FAILURE);
```

```
        }

        str = argv[1];
        base = (argc > 2) ? atoi(argv[2]) : 0;

        errno = 0;    /* To distinguish success/failure after call */
        strtol("0", NULL, base);
        if (errno == EINVAL) {
            perror("strtol");
            exit(EXIT_FAILURE);
        }

        errno = 0;    /* To distinguish success/failure after call */
        val = strtol(str, &endptr, base);

        /* Check for various possible errors. */

        if (errno == ERANGE) {
            perror("strtol");
            exit(EXIT_FAILURE);
        }

        if (endptr == str) {
            fprintf(stderr, "No digits were found\n");
            exit(EXIT_FAILURE);
        }

        /* If we got here, strtol() successfully parsed a number. */

        printf("strtol() returned %ld\n", val);

        if (*endptr != '\0')        /* Not necessarily an error... */
            printf("Further characters after number: \"%s\"\n", endptr);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

atof(3), *atoi*(3), *atol*(3), *strtod*(3), *strtoimax*(3), *strtoul*(3)

NAME

strtoul, strtoull, strtouq − convert a string to an unsigned long integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict nptr,
                      char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict nptr,
                      char **restrict endptr, int base);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

strtoull():
_ISOC99_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

The **strtoul**() function converts the initial part of the string in *nptr* to an *unsigned long* value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by *isspace*(3)) followed by a single optional '+' or '−' sign. If *base* is zero or 16, the string may then include a "0x" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to an *unsigned long* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either uppercase or lowercase represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, and the *base* is supported, **strtoul**() stores the address of the first invalid character in **endptr*. If there were no digits at all, **strtoul**() stores the original value of *nptr* in **endptr* (and returns 0). In particular, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string is valid.

The **strtoull**() function works just like the **strtoul**() function but returns an *unsigned long long* value.

RETURN VALUE

The **strtoul**() function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion represented as an unsigned value, unless the original (non-negated) value would overflow; in the latter case, **strtoul**() returns **ULONG_MAX** and sets *errno* to **ERANGE**. Precisely the same holds for **strtoull**() (with **ULLONG_MAX** instead of **ULONG_MAX**).

ERRORS

This function does not modify *errno* on success.

EINVAL
(not in C99) The given *base* contains an unsupported value.

ERANGE
The resulting value was out of range.

The implementation may also set *errno* to **EINVAL** in case no conversion was performed (no digits seen, and 0 returned).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strtoul(), **strtoull**(), **strtouq**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

strtoul()
POSIX.1-2001, C89, SVr4.

strtoull()
POSIX.1-2001, C99.

NOTES

Since **strtoul**() can legitimately return 0 or **ULONG_MAX** (**ULLONG_MAX** for *strtoull*()) on both success and failure, the calling program should set *errno* to 0 before the call, and then determine if an error occurred by checking whether *errno* has a nonzero value after the call.

In locales other than the "C" locale, other strings may be accepted. (For example, the thousands separator of the current locale may be supported.)

BSD also has

```
u_quad_t strtouq(const char *nptr, char **endptr, int base);
```

with completely analogous definition. Depending on the wordsize of the current architecture, this may be equivalent to **strtoull**() or to **strtoul**().

Negative values are considered valid input and are silently converted to the equivalent *unsigned long* value.

EXAMPLES

See the example on the *strtol*(3) manual page; the use of the functions described in this manual page is similar.

SEE ALSO

a64l(3), *atof*(3), *atoi*(3), *atol*(3), *strtod*(3), *strtol*(3), *strtoumax*(3)

NAME

strverscmp − compare two version strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <string.h>

int strverscmp(const char *_s1_, const char *_s2_);

DESCRIPTION

Often one has files *jan1*, *jan2*, ..., *jan9*, *jan10*, ... and it feels wrong when *ls*(1) orders them *jan1*, *jan10*, ..., *jan2*, ..., *jan9*. In order to rectify this, GNU introduced the *−v* option to *ls*(1), which is implemented using *versionsort*(3), which again uses **strverscmp**().

Thus, the task of **strverscmp**() is to compare two strings and find the "right" order, while *strcmp*(3) finds only the lexicographic order. This function does not use the locale category **LC_COLLATE**, so is meant mostly for situations where the strings are expected to be in ASCII.

What this function does is the following. If both strings are equal, return 0. Otherwise, find the position between two bytes with the property that before it both strings are equal, while directly after it there is a difference. Find the largest consecutive digit strings containing (or starting at, or ending at) this position. If one or both of these is empty, then return what *strcmp*(3) would have returned (numerical ordering of byte values). Otherwise, compare both digit strings numerically, where digit strings with one or more leading zeros are interpreted as if they have a decimal point in front (so that in particular digit strings with more leading zeros come before digit strings with fewer leading zeros). Thus, the ordering is *000*, *00*, *01*, *010*, *09*, *0*, *1*, *9*, *10*.

RETURN VALUE

The **strverscmp**() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be earlier than, equal to, or later than *s2*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strverscmp()	Thread safety	MT-Safe

STANDARDS

GNU.

EXAMPLES

The program below can be used to demonstrate the behavior of **strverscmp**(). It uses **strverscmp**() to compare the two strings given as its command-line arguments. An example of its use is the following:

```
$ ./a.out jan1 jan10
jan1 < jan10
```

Program source

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int res;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <string1> <string2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
        res = strverscmp(argv[1], argv[2]);

        printf("%s %s %s\n", argv[1],
               (res < 0) ? "<" : (res == 0) ? "==" : ">", argv[2]);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

rename(1), *strcasecmp*(3), *strcmp*(3), *strcoll*(3)

NAME

strxfrm − string transformation

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <string.h>

size_t strxfrm(char *dest***[restrict .***n***], const char** *src***[restrict .***n***],**
size_t *n***);**

DESCRIPTION

The **strxfrm**() function transforms the *src* string into a form such that the result of *strcmp*(3) on two strings that have been transformed with **strxfrm**() is the same as the result of *strcoll*(3) on the two strings before their transformation. The first *n* bytes of the transformed string are placed in *dest*. The transformation is based on the program's current locale for category **LC_COLLATE**. (See *setlocale*(3)).

RETURN VALUE

The **strxfrm**() function returns the number of bytes required to store the transformed string in *dest* excluding the terminating null byte ('\0'). If the value returned is *n* or more, the contents of *dest* are indeterminate.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
strxfrm()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD.

SEE ALSO

memcmp(3), *setlocale*(3), *strcasecmp*(3), *strcmp*(3), *strcoll*(3), *string*(3)

NAME

swab − swap adjacent bytes

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE       /* See feature_test_macros(7) */
#include <unistd.h>

void swab(const void *from[restrict .n], void *to[restrict .n],
          ssize_t n);
```

DESCRIPTION

The **swab**() function copies *n* bytes from the array pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. This function is used to exchange data between machines that have different low/high byte ordering.

This function does nothing when *n* is negative. When *n* is positive and odd, it handles *n−1* bytes as above, and does something unspecified with the last byte. (In other words, *n* should be even.)

RETURN VALUE

The **swab**() function returns no value.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
swab()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

SEE ALSO

bstring(3)

NAME

sysconf − get configuration information at run time

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

long sysconf(int *name***);**

DESCRIPTION

POSIX allows an application to test at compile or run time whether certain options are supported, or what the value is of certain configurable constants or limits.

At compile time this is done by including <*unistd.h*> and/or <*limits.h*> and testing the value of certain macros.

At run time, one can ask for numerical values using the present function **sysconf**(). One can ask for numerical values that may depend on the filesystem in which a file resides using *fpathconf*(3) and *pathconf*(3). One can ask for string values using *confstr*(3).

The values obtained from these functions are system configuration constants. They do not change during the lifetime of a process.

For options, typically, there is a constant **_POSIX_FOO** that may be defined in <*unistd.h*>. If it is undefined, one should ask at run time. If it is defined to −1, then the option is not supported. If it is defined to 0, then relevant functions and headers exist, but one has to ask at run time what degree of support is available. If it is defined to a value other than −1 or 0, then the option is supported. Usually the value (such as 200112L) indicates the year and month of the POSIX revision describing the option. glibc uses the value 1 to indicate support as long as the POSIX revision has not been published yet. The **sysconf**() argument will be **_SC_FOO**. For a list of options, see *posixoptions*(7).

For variables or limits, typically, there is a constant **_FOO**, maybe defined in <*limits.h*>, or **_POSIX_FOO**, maybe defined in <*unistd.h*>. The constant will not be defined if the limit is unspecified. If the constant is defined, it gives a guaranteed value, and a greater value might actually be supported. If an application wants to take advantage of values which may change between systems, a call to **sysconf**() can be made. The **sysconf**() argument will be **_SC_FOO**.

POSIX.1 variables

We give the name of the variable, the name of the **sysconf**() argument used to inquire about its value, and a short description.

First, the POSIX.1 compatible values.

ARG_MAX - _SC_ARG_MAX
: The maximum length of the arguments to the *exec*(3) family of functions. Must not be less than **_POSIX_ARG_MAX** (4096).

CHILD_MAX - _SC_CHILD_MAX
: The maximum number of simultaneous processes per user ID. Must not be less than **_POSIX_CHILD_MAX** (25).

HOST_NAME_MAX - _SC_HOST_NAME_MAX
: Maximum length of a hostname, not including the terminating null byte, as returned by *gethostname*(2). Must not be less than **_POSIX_HOST_NAME_MAX** (255).

LOGIN_NAME_MAX - _SC_LOGIN_NAME_MAX
: Maximum length of a login name, including the terminating null byte. Must not be less than **_POSIX_LOGIN_NAME_MAX** (9).

NGROUPS_MAX - _SC_NGROUPS_MAX
: Maximum number of supplementary group IDs.

clock ticks - **_SC_CLK_TCK**
: The number of clock ticks per second. The corresponding variable is obsolete. It was of course called **CLK_TCK**. (Note: the macro **CLOCKS_PER_SEC** does not give information: it must equal 1000000.)

OPEN_MAX - **_SC_OPEN_MAX**
The maximum number of files that a process can have open at any time. Must not be less than **_POSIX_OPEN_MAX** (20).

PAGESIZE - **_SC_PAGESIZE**
Size of a page in bytes. Must not be less than 1.

PAGE_SIZE - **_SC_PAGE_SIZE**
A synonym for **PAGESIZE**/**_SC_PAGESIZE**. (Both **PAGESIZE** and **PAGE_SIZE** are specified in POSIX.)

RE_DUP_MAX - **_SC_RE_DUP_MAX**
The number of repeated occurrences of a BRE permitted by *regexec*(3) and *regcomp*(3). Must not be less than **_POSIX2_RE_DUP_MAX** (255).

STREAM_MAX - **_SC_STREAM_MAX**
The maximum number of streams that a process can have open at any time. If defined, it has the same value as the standard C macro **FOPEN_MAX**. Must not be less than **_POSIX_STREAM_MAX** (8).

SYMLOOP_MAX - **_SC_SYMLOOP_MAX**
The maximum number of symbolic links seen in a pathname before resolution returns **ELOOP**. Must not be less than **_POSIX_SYMLOOP_MAX** (8).

TTY_NAME_MAX - **_SC_TTY_NAME_MAX**
The maximum length of terminal device name, including the terminating null byte. Must not be less than **_POSIX_TTY_NAME_MAX** (9).

TZNAME_MAX - **_SC_TZNAME_MAX**
The maximum number of bytes in a timezone name. Must not be less than **_POSIX_TZNAME_MAX** (6).

_POSIX_VERSION - **_SC_VERSION**
indicates the year and month the POSIX.1 standard was approved in the format **YYYYMML**; the value **199009L** indicates the Sept. 1990 revision.

POSIX.2 variables
Next, the POSIX.2 values, giving limits for utilities.

BC_BASE_MAX - **_SC_BC_BASE_MAX**
indicates the maximum *obase* value accepted by the *bc*(1) utility.

BC_DIM_MAX - **_SC_BC_DIM_MAX**
indicates the maximum value of elements permitted in an array by *bc*(1)

BC_SCALE_MAX - **_SC_BC_SCALE_MAX**
indicates the maximum *scale* value allowed by *bc*(1)

BC_STRING_MAX - **_SC_BC_STRING_MAX**
indicates the maximum length of a string accepted by *bc*(1)

COLL_WEIGHTS_MAX - **_SC_COLL_WEIGHTS_MAX**
indicates the maximum numbers of weights that can be assigned to an entry of the **LC_COLLATE order** keyword in the locale definition file.

EXPR_NEST_MAX - **_SC_EXPR_NEST_MAX**
is the maximum number of expressions which can be nested within parentheses by *expr*(1)

LINE_MAX - **_SC_LINE_MAX**
The maximum length of a utility's input line, either from standard input or from a file. This includes space for a trailing newline.

RE_DUP_MAX - **_SC_RE_DUP_MAX**
The maximum number of repeated occurrences of a regular expression when the interval notation **\{m,n\}** is used.

POSIX2_VERSION - **_SC_2_VERSION**
indicates the version of the POSIX.2 standard in the format of YYYYMML.

POSIX2_C_DEV - _SC_2_C_DEV
indicates whether the POSIX.2 C language development facilities are supported.

POSIX2_FORT_DEV - _SC_2_FORT_DEV
indicates whether the POSIX.2 FORTRAN development utilities are supported.

POSIX2_FORT_RUN - _SC_2_FORT_RUN
indicates whether the POSIX.2 FORTRAN run-time utilities are supported.

_POSIX2_LOCALEDEF - _SC_2_LOCALEDEF
indicates whether the POSIX.2 creation of locales via *localedef*(1) is supported.

POSIX2_SW_DEV - _SC_2_SW_DEV
indicates whether the POSIX.2 software development utilities option is supported.

These values also exist, but may not be standard.

- **_SC_PHYS_PAGES**
The number of pages of physical memory. Note that it is possible for the product of this value and the value of **_SC_PAGESIZE** to overflow.

- **_SC_AVPHYS_PAGES**
The number of currently available pages of physical memory.

- **_SC_NPROCESSORS_CONF**
The number of processors configured. See also *get_nprocs_conf*(3).

- **_SC_NPROCESSORS_ONLN**
The number of processors currently online (available). See also *get_nprocs_conf*(3).

RETURN VALUE

The return value of **sysconf**() is one of the following:

- On error, −1 is returned and *errno* is set to indicate the error (for example, **EINVAL**, indicating that *name* is invalid).
- If *name* corresponds to a maximum or minimum limit, and that limit is indeterminate, −1 is returned and *errno* is not changed. (To distinguish an indeterminate limit from an error, set *errno* to zero before the call, and then check whether *errno* is nonzero when −1 is returned.)
- If *name* corresponds to an option, a positive value is returned if the option is supported, and −1 is returned if the option is not supported.
- Otherwise, the current value of the option or limit is returned. This value will not be more restrictive than the corresponding value that was described to the application in <*unistd.h*> or <*limits.h*> when the application was compiled.

ERRORS

EINVAL
name is invalid.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sysconf()	Thread safety	MT-Safe env

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

BUGS

It is difficult to use **ARG_MAX** because it is not specified how much of the argument space for *exec*(3) is consumed by the user's environment variables.

Some returned values may be huge; they are not suitable for allocating memory.

SEE ALSO

bc(1), *expr*(1), *getconf* (1), *locale*(1), *confstr*(3), *fpathconf*(3), *pathconf*(3), *posixoptions*(7)

NAME

closelog, openlog, syslog, vsyslog − send messages to the system logger

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char * format, ...);
void closelog(void);

void vsyslog(int priority, const char * format, va_list ap);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

vsyslog():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

openlog()

openlog() opens a connection to the system logger for a program.

The string pointed to by *ident* is prepended to every message, and is typically set to the program name. If *ident* is NULL, the program name is used. (POSIX.1-2008 does not specify the behavior when *ident* is NULL.)

The *option* argument specifies flags which control the operation of **openlog**() and subsequent calls to **syslog**(). The *facility* argument establishes a default to be used if none is specified in subsequent calls to **syslog**(). The values that may be specified for *option* and *facility* are described below.

The use of **openlog**() is optional; it will automatically be called by **syslog**() if necessary, in which case *ident* will default to NULL.

syslog() and vsyslog()

syslog() generates a log message, which will be distributed by *syslogd*(8)

The *priority* argument is formed by ORing together a *facility* value and a *level* value (described below). If no *facility* value is ORed into *priority*, then the default value set by **openlog**() is used, or, if there was no preceding **openlog**() call, a default of **LOG_USER** is employed.

The remaining arguments are a *format*, as in *printf*(3), and any arguments required by the *format*, except that the two-character sequence **%m** will be replaced by the error message string *strerror*(*errno*). The format string need not include a terminating newline character.

The function **vsyslog**() performs the same task as **syslog**() with the difference that it takes a set of arguments which have been obtained using the *stdarg*(3) variable argument list macros.

closelog()

closelog() closes the file descriptor being used to write to the system logger. The use of **closelog**() is optional.

Values for *option*

The *option* argument to **openlog**() is a bit mask constructed by ORing together any of the following values:

LOG_CONS Write directly to the system console if there is an error while sending to the system logger.

LOG_NDELAY Open the connection immediately (normally, the connection is opened when the first message is logged). This may be useful, for example, if a subsequent *chroot*(2) would make the pathname used internally by the logging facility unreachable.

LOG_NOWAIT Don't wait for child processes that may have been created while logging the message. (The GNU C library does not create a child process, so this option has no effect on Linux.)

LOG_ODELAY The converse of **LOG_NDELAY**; opening of the connection is delayed until **syslog**() is called. (This is the default, and need not be specified.)

LOG_PERROR (Not in POSIX.1-2001 or POSIX.1-2008.) Also log the message to *stderr*.

LOG_PID Include the caller's PID with each message.

Values for *facility*

The *facility* argument is used to specify what type of program is logging the message. This lets the configuration file specify that messages from different facilities will be handled differently.

LOG_AUTH security/authorization messages

LOG_AUTHPRIV
security/authorization messages (private)

LOG_CRON clock daemon (**cron** and **at**)

LOG_DAEMON
system daemons without separate facility value

LOG_FTP ftp daemon

LOG_KERN kernel messages (these can't be generated from user processes)

LOG_LOCAL0 through **LOG_LOCAL7**
reserved for local use

LOG_LPR line printer subsystem

LOG_MAIL mail subsystem

LOG_NEWS USENET news subsystem

LOG_SYSLOG messages generated internally by *syslogd*(8)

LOG_USER (default)
generic user-level messages

LOG_UUCP UUCP subsystem

Values for *level*

This determines the importance of the message. The levels are, in order of decreasing importance:

LOG_EMERG system is unusable

LOG_ALERT action must be taken immediately

LOG_CRIT critical conditions

LOG_ERR error conditions

LOG_WARNING
warning conditions

LOG_NOTICE normal, but significant, condition

LOG_INFO informational message

LOG_DEBUG debug-level message

The function *setlogmask*(3) can be used to restrict logging to specified levels only.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
openlog(), **closelog**()	Thread safety	MT-Safe
syslog(), **vsyslog**()	Thread safety	MT-Safe env locale

STANDARDS

syslog()
openlog()
closelog()
POSIX.1-2008.

vsyslog()
None.

HISTORY

syslog()
4.2BSD, SUSv2, POSIX.1-2001.

openlog()
closelog()
4.3BSD, SUSv2, POSIX.1-2001.

vsyslog()
4.3BSD-Reno.

POSIX.1-2001 specifies only the **LOG_USER** and **LOG_LOCAL*** values for *facility*. However, with the exception of **LOG_AUTHPRIV** and **LOG_FTP**, the other *facility* values appear on most UNIX systems.

The **LOG_PERROR** value for *option* is not specified by POSIX.1-2001 or POSIX.1-2008, but is available in most versions of UNIX.

NOTES

The argument *ident* in the call of **openlog**() is probably stored as-is. Thus, if the string it points to is changed, **syslog**() may start prepending the changed string, and if the string it points to ceases to exist, the results are undefined. Most portable is to use a string constant.

Never pass a string with user-supplied data as a format, use the following instead:

```
syslog(priority, "%s", string);
```

SEE ALSO

journalctl(1), *logger*(1), *setlogmask*(3), *syslog.conf*(5), *syslogd*(8)

NAME

system − execute a shell command

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int system(const char *****command***);**

DESCRIPTION

The **system**() library function behaves as if it used *fork*(2) to create a child process that executed the shell command specified in *command* using *execl*(3) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

system() returns after the command has been completed.

During execution of the command, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system**(). (These signals will be handled according to their defaults inside the child process that executes *command*.)

If *command* is NULL, then **system**() returns a status indicating whether a shell is available on the system.

RETURN VALUE

The return value of **system**() is one of the following:

- If *command* is NULL, then a nonzero value if a shell is available, or 0 if no shell is available.
- If a child process could not be created, or its status could not be retrieved, the return value is −1 and *errno* is set to indicate the error.
- If a shell could not be executed in the child process, then the return value is as though the child shell terminated by calling *_exit*(2) with the status 127.
- If all system calls succeed, then the return value is the termination status of the child shell used to execute *command*. (The termination status of a shell is the termination status of the last command it executes.)

In the last two cases, the return value is a "wait status" that can be examined using the macros described in *waitpid*(2). (i.e., **WIFEXITED**(), **WEXITSTATUS**(), and so on).

system() does not affect the wait status of any other children.

ERRORS

system() can fail with any of the same errors as *fork*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
system()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

NOTES

system() provides simplicity and convenience: it handles all of the details of calling *fork*(2), *execl*(3), and *waitpid*(2), as well as the necessary manipulations of signals; in addition, the shell performs the usual substitutions and I/O redirections for *command*. The main cost of **system**() is inefficiency: additional system calls are required to create the process that runs the shell and to execute the shell.

If the **_XOPEN_SOURCE** feature test macro is defined (before including *any* header files), then the macros described in *waitpid*(2) (**WEXITSTATUS**(), etc.) are made available when including *<stdlib.h>*.

As mentioned, **system**() ignores **SIGINT** and **SIGQUIT**. This may make programs that call it from a loop uninterruptible, unless they take care themselves to check the exit status of the child. For

example:

```
while (something) {
    int ret = system("foo");

    if (WIFSIGNALED(ret) &&
        (WTERMSIG(ret) == SIGINT || WTERMSIG(ret) == SIGQUIT))
            break;
}
```

According to POSIX.1, it is unspecified whether handlers registered using *pthread_atfork*(3) are called during the execution of **system**(). In the glibc implementation, such handlers are not called.

Before glibc 2.1.3, the check for the availability of */bin/sh* was not actually performed if *command* was NULL; instead it was always assumed to be available, and **system**() always returned 1 in this case. Since glibc 2.1.3, this check is performed because, even though POSIX.1-2001 requires a conforming implementation to provide a shell, that shell may not be available or executable if the calling program has previously called *chroot*(2) (which is not specified by POSIX.1-2001).

It is possible for the shell command to terminate with a status of 127, which yields a **system**() return value that is indistinguishable from the case where a shell could not be executed in the child process.

Caveats

Do not use **system**() from a privileged program (a set-user-ID or set-group-ID program, or a program with capabilities) because strange values for some environment variables might be used to subvert system integrity. For example, **PATH** could be manipulated so that an arbitrary program is executed with privilege. Use the *exec*(3) family of functions instead, but not *execlp*(3) or *execvp*(3) (which also use the **PATH** environment variable to search for an executable).

system() will not, in fact, work properly from programs with set-user-ID or set-group-ID privileges on systems on which */bin/sh* is bash version 2: as a security measure, bash 2 drops privileges on startup. (Debian uses a different shell, *dash*(1), which does not do this when invoked as **sh**.)

Any user input that is employed as part of *command* should be *carefully* sanitized, to ensure that unexpected shell commands or command options are not executed. Such risks are especially grave when using **system**() from a privileged program.

BUGS

If the command name starts with a hyphen, *sh*(1) interprets the command name as an option, and the behavior is undefined. (See the **-c** option to *sh*(1)To work around this problem, prepend the command with a space as in the following call:

```
system(" -unfortunate-command-name");
```

SEE ALSO

sh(1), *execve*(2), *fork*(2), *sigaction*(2), *sigprocmask*(2), *wait*(2), *exec*(3), *signal*(7)

NAME

sysv_signal − signal handling with System V semantics

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t sysv_signal(int signum, sighandler_t handler);
```

DESCRIPTION

The **sysv_signal**() function takes the same arguments, and performs the same task, as *signal*(2).

However **sysv_signal**() provides the System V unreliable signal semantics, that is: a) the disposition of the signal is reset to the default when the handler is invoked; b) delivery of further instances of the signal is not blocked while the signal handler is executing; and c) if the handler interrupts (certain) blocking system calls, then the system call is not automatically restarted.

RETURN VALUE

The **sysv_signal**() function returns the previous value of the signal handler, or **SIG_ERR** on error.

ERRORS

As for *signal*(2).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
sysv_signal()	Thread safety	MT-Safe

VERSIONS

Use of **sysv_signal**() should be avoided; use *sigaction*(2) instead.

On older Linux systems, **sysv_signal**() and *signal*(2) were equivalent. But on newer systems, *signal*(2) provides reliable signal semantics; see *signal*(2) for details.

The use of *sighandler_t* is a GNU extension; this type is defined only if the **_GNU_SOURCE** feature test macro is defined.

STANDARDS

None.

SEE ALSO

sigaction(2), *signal*(2), *bsd_signal*(3), *signal*(7)

NAME

TAILQ_CONCAT, TAILQ_EMPTY, TAILQ_ENTRY, TAILQ_FIRST, TAILQ_FOREACH, TAILQ_FOREACH_REVERSE, TAILQ_HEAD, TAILQ_HEAD_INITIALIZER, TAILQ_INIT, TAILQ_INSERT_AFTER, TAILQ_INSERT_BEFORE, TAILQ_INSERT_HEAD, TAILQ_INSERT_TAIL, TAILQ_LAST, TAILQ_NEXT, TAILQ_PREV, TAILQ_REMOVE − implementation of a doubly linked tail queue

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/queue.h>

TAILQ_ENTRY(TYPE);

TAILQ_HEAD(HEADNAME, TYPE);
TAILQ_HEAD TAILQ_HEAD_INITIALIZER(TAILQ_HEAD *head***);**
void TAILQ_INIT(TAILQ_HEAD ***_head_);**

int TAILQ_EMPTY(TAILQ_HEAD ***_head_);**

void TAILQ_INSERT_HEAD(TAILQ_HEAD ***_head_,**
struct TYPE ***_elm_, TAILQ_ENTRY** *NAME***);**
void TAILQ_INSERT_TAIL(TAILQ_HEAD ***_head_,**
struct TYPE ***_elm_, TAILQ_ENTRY** *NAME***);**
void TAILQ_INSERT_BEFORE(struct TYPE ***_listelm_,**
struct TYPE ***_elm_, TAILQ_ENTRY** *NAME***);**
void TAILQ_INSERT_AFTER(TAILQ_HEAD ***_head_, struct TYPE ***_listelm_**,**
struct TYPE ***_elm_, TAILQ_ENTRY** *NAME***);**

struct TYPE *TAILQ_FIRST(TAILQ_HEAD ***_head_);**
struct TYPE *TAILQ_LAST(TAILQ_HEAD ***_head_, HEADNAME);**
struct TYPE *TAILQ_PREV(struct TYPE ***_elm_, HEADNAME, TAILQ_ENTRY** *NAME***);**
struct TYPE *TAILQ_NEXT(struct TYPE ***_elm_, TAILQ_ENTRY** *NAME***);**

TAILQ_FOREACH(struct TYPE ***_var_, TAILQ_HEAD ***_head_**,**
TAILQ_ENTRY *NAME***);**
TAILQ_FOREACH_REVERSE(struct TYPE ***_var_, TAILQ_HEAD ***_head_**, HEADNAME,**
TAILQ_ENTRY *NAME***);**

void TAILQ_REMOVE(TAILQ_HEAD ***_head_, struct TYPE ***_elm_**,**
TAILQ_ENTRY *NAME***);**

void TAILQ_CONCAT(TAILQ_HEAD ***_head1_, TAILQ_HEAD ***_head2_**,**
TAILQ_ENTRY *NAME***);**

DESCRIPTION

These macros define and operate on doubly linked tail queues.

In the macro definitions, *TYPE* is the name of a user defined structure, that must contain a field of type *TAILQ_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user defined structure that must be declared using the macro **TAILQ_HEAD**().

Creation

A tail queue is headed by a structure defined by the **TAILQ_HEAD**() macro. This structure contains a pair of pointers, one to the first element in the queue and the other to the last element in the queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A *TAILQ_HEAD* structure is declared as follows:

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

where *struct HEADNAME* is the structure to be defined, and *struct TYPE* is the type of the elements to be linked into the queue. A pointer to the head of the queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

TAILQ_ENTRY() declares a structure that connects the elements in the queue.

TAILQ_HEAD_INITIALIZER() evaluates to an initializer for the queue *head*.

TAILQ_INIT() initializes the queue referenced by

TAILQ_EMPTY() evaluates to true if there are no items on the queue. *head*.

Insertion

TAILQ_INSERT_HEAD() inserts the new element *elm* at the head of the queue.

TAILQ_INSERT_TAIL() inserts the new element *elm* at the end of the queue.

TAILQ_INSERT_BEFORE() inserts the new element *elm* before the element *listelm*.

TAILQ_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

Traversal

TAILQ_FIRST() returns the first item on the queue, or NULL if the queue is empty.

TAILQ_LAST() returns the last item on the queue. If the queue is empty the return value is NULL.

TAILQ_PREV() returns the previous item on the queue, or NULL if this item is the first.

TAILQ_NEXT() returns the next item on the queue, or NULL if this item is the last.

TAILQ_FOREACH() traverses the queue referenced by *head* in the forward direction, assigning each element in turn to *var*. *var* is set to NULL if the loop completes normally, or if there were no elements.

TAILQ_FOREACH_REVERSE() traverses the queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

Removal

TAILQ_REMOVE() removes the element *elm* from the queue.

Other features

TAILQ_CONCAT() concatenates the queue headed by *head2* onto the end of the one headed by *head1* removing all entries from the former.

RETURN VALUE

TAILQ_EMPTY() returns nonzero if the queue is empty, and zero if the queue contains at least one entry.

TAILQ_FIRST(), **TAILQ_LAST**(), **TAILQ_PREV**(), and **TAILQ_NEXT**() return a pointer to the first, last, previous, or next *TYPE* structure, respectively.

TAILQ_HEAD_INITIALIZER() returns an initializer that can be assigned to the queue *head*.

STANDARDS

BSD.

HISTORY

4.4BSD.

CAVEATS

TAILQ_FOREACH() and **TAILQ_FOREACH_REVERSE**() don't allow *var* to be removed or freed within the loop, as it would interfere with the traversal. **TAILQ_FOREACH_SAFE**() and **TAILQ_FOREACH_REVERSE_SAFE**(), which are present on the BSDs but are not present in glibc, fix this limitation by allowing *var* to safely be removed from the list and freed from within the loop without interfering with the traversal.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    TAILQ_ENTRY(entry) entries;             /* Tail queue */
};
```

```
TAILQ_HEAD(tailhead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct tailhead head;                   /* Tail queue head */
    int i;

    TAILQ_INIT(&head);                      /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));      /* Insert at the head */
    TAILQ_INSERT_HEAD(&head, n1, entries);

    n1 = malloc(sizeof(struct entry));      /* Insert at the tail */
    TAILQ_INSERT_TAIL(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));      /* Insert after */
    TAILQ_INSERT_AFTER(&head, n1, n2, entries);

    n3 = malloc(sizeof(struct entry));      /* Insert before */
    TAILQ_INSERT_BEFORE(n2, n3, entries);

    TAILQ_REMOVE(&head, n2, entries);       /* Deletion */
    free(n2);
                                            /* Forward traversal */
    i = 0;
    TAILQ_FOREACH(np, &head, entries)
        np->data = i++;
                                            /* Reverse traversal */
    TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
        printf("%i\n", np->data);
                                            /* TailQ deletion */
    n1 = TAILQ_FIRST(&head);
    while (n1 != NULL) {
        n2 = TAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    TAILQ_INIT(&head);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

insque(3), *queue*(7)

NAME

tan, tanf, tanl − tangent function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tanf(), **tanl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the tangent of *x*, where *x* is given in radians.

RETURN VALUE

On success, these functions return the tangent of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity or negative infinity, a domain error occurs, and a NaN is returned.

If the correct result would overflow, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the mathematically correct sign.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is an infinity
: *errno* is set to **EDOM** (but see BUGS). An invalid floating-point exception (**FE_INVALID**) is raised.

Range error: result overflow
: An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tan(), **tanf**(), **tanl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

BUGS

Before glibc 2.10, the glibc implementation did not set *errno* to **EDOM** when a domain error occurred.

SEE ALSO

acos(3), *asin*(3), *atan*(3), *atan2*(3), *cos*(3), *ctan*(3), *sin*(3)

NAME

tanh, tanhf, tanhl − hyperbolic tangent function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tanhf(), **tanhl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

These functions return the hyperbolic tangent of *x*, which is defined mathematically as:

tanh(x) = sinh(x) / cosh(x)

RETURN VALUE

On success, these functions return the hyperbolic tangent of *x*.

If *x* is a NaN, a NaN is returned.

If *x* is +0 (−0), +0 (−0) is returned.

If *x* is positive infinity (negative infinity), +1 (−1) is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tanh(), **tanhf**(), **tanhl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The variant returning *double* also conforms to SVr4, 4.3BSD, C89.

SEE ALSO

acosh(3), *asinh*(3), *atanh*(3), *cosh*(3), *ctanh*(3), *sinh*(3)

NAME

tcgetpgrp, tcsetpgrp − get and set terminal foreground process group

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
int tcsetpgrp(int fd, pid_t pgrp);
```

DESCRIPTION

The function **tcgetpgrp**() returns the process group ID of the foreground process group on the terminal associated to *fd*, which must be the controlling terminal of the calling process.

The function **tcsetpgrp**() makes the process group with process group ID *pgrp* the foreground process group on the terminal associated to *fd*, which must be the controlling terminal of the calling process, and still be associated with its session. Moreover, *pgrp* must be a (nonempty) process group belonging to the same session as the calling process.

If **tcsetpgrp**() is called by a member of a background process group in its session, and the calling process is not blocking or ignoring **SIGTTOU**, a **SIGTTOU** signal is sent to all members of this background process group.

RETURN VALUE

When *fd* refers to the controlling terminal of the calling process, the function **tcgetpgrp**() will return the foreground process group ID of that terminal if there is one, and some value larger than 1 that is not presently a process group ID otherwise. When *fd* does not refer to the controlling terminal of the calling process, −1 is returned, and *errno* is set to indicate the error.

When successful, **tcsetpgrp**() returns 0. Otherwise, it returns −1, and *errno* is set to indicate the error.

ERRORS

EBADF
: *fd* is not a valid file descriptor.

EINVAL
: *pgrp* has an unsupported value.

ENOTTY
: The calling process does not have a controlling terminal, or it has one but it is not described by *fd*, or, for **tcsetpgrp**(), this controlling terminal is no longer associated with the session of the calling process.

EPERM
: *pgrp* has a supported value, but is not the process group ID of a process in the same session as the calling process.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tcgetpgrp(), **tcsetpgrp**()	Thread safety	MT-Safe

VERSIONS

These functions are implemented via the **TIOCGPGRP** and **TIOCSPGRP** ioctls.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

The ioctls appeared in 4.2BSD. The functions are POSIX inventions.

SEE ALSO

setpgid(2), *setsid*(2), *credentials*(7)

NAME

tcgetsid − get session ID

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE 500       /* See feature_test_macros(7) */
#include <termios.h>

pid_t tcgetsid(int fd);
```

DESCRIPTION

The function **tcgetsid**() returns the session ID of the current session that has the terminal associated to *fd* as controlling terminal. This terminal must be the controlling terminal of the calling process.

RETURN VALUE

When *fd* refers to the controlling terminal of our session, the function **tcgetsid**() will return the session ID of this session. Otherwise, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
fd is not a valid file descriptor.

ENOTTY
The calling process does not have a controlling terminal, or it has one but it is not described by *fd*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tcgetsid()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

This function is implemented via the **TIOCGSID** *ioctl*(2), present since Linux 2.1.71.

SEE ALSO

getsid(2)

NAME

telldir − return current location in directory stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <dirent.h>

long telldir(DIR *dirp);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

telldir():
_XOPEN_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The **telldir**() function returns the current location associated with the directory stream *dirp*.

RETURN VALUE

On success, the **telldir**() function returns the current location in the directory stream. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
Invalid directory stream descriptor *dirp*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
telldir()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.3BSD.

Up to glibc 2.1.1, the return type of **telldir**() was *off_t*. POSIX.1-2001 specifies *long*, and this is the type used since glibc 2.1.2.

In early filesystems, the value returned by **telldir**() was a simple file offset within a directory. Modern filesystems use tree or hash structures, rather than flat tables, to represent directories. On such filesystems, the value returned by **telldir**() (and used internally by *readdir*(3)) is a "cookie" that is used by the implementation to derive a position within a directory. Application programs should treat this strictly as an opaque value, making *no* assumptions about its contents.

SEE ALSO

closedir(3), *opendir*(3), *readdir*(3), *rewinddir*(3), *scandir*(3), *seekdir*(3)

NAME

tempnam − create a name for a temporary file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

char *tempnam(const char * *dir***, const char *** *pfx***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tempnam():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Never use this function. Use *mkstemp*(3) or *tmpfile*(3) instead.

The **tempnam**() function returns a pointer to a string that is a valid filename, and such that a file with this name did not exist when **tempnam**() checked. The filename suffix of the pathname generated will start with *pfx* in case *pfx* is a non-NULL string of at most five bytes. The directory prefix part of the pathname generated is required to be "appropriate" (often that at least implies writable).

Attempts to find an appropriate directory go through the following steps:

a) In case the environment variable **TMPDIR** exists and contains the name of an appropriate directory, that is used.

b) Otherwise, if the *dir* argument is non-NULL and appropriate, it is used.

c) Otherwise, *P_tmpdir* (as defined in <*stdio.h*>) is used when appropriate.

d) Finally an implementation-defined directory may be used.

The string returned by **tempnam**() is allocated using *malloc*(3) and hence should be freed by *free*(3).

RETURN VALUE

On success, the **tempnam**() function returns a pointer to a unique temporary filename. It returns NULL if a unique name cannot be generated, with *errno* set to indicate the error.

ERRORS

ENOMEM
Allocation of storage failed.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tempnam()	Thread safety	MT-Safe env

STANDARDS

POSIX.1-2008.

HISTORY

SVr4, 4.3BSD, POSIX.1-2001. Obsoleted in POSIX.1-2008.

NOTES

Although **tempnam**() generates names that are difficult to guess, it is nevertheless possible that between the time that **tempnam**() returns a pathname, and the time that the program opens it, another program might create that pathname using *open*(2), or create it as a symbolic link. This can lead to security holes. To avoid such possibilities, use the *open*(2) **O_EXCL** flag to open the pathname. Or better yet, use *mkstemp*(3) or *tmpfile*(3).

SUSv2 does not mention the use of **TMPDIR**; glibc will use it only when the program is not set-user-ID. On SVr4, the directory used under **d)** is */tmp* (and this is what glibc does).

Because it dynamically allocates memory used to return the pathname, **tempnam**() is reentrant, and thus thread safe, unlike *tmpnam*(3).

The **tempnam**() function generates a different string each time it is called, up to **TMP_MAX** (defined in <*stdio.h*>) times. If it is called more than **TMP_MAX** times, the behavior is implementation defined.

tempnam() uses at most the first five bytes from *pfx*.

The glibc implementation of **tempnam**() fails with the error **EEXIST** upon failure to find a unique name.

BUGS

The precise meaning of "appropriate" is undefined; it is unspecified how accessibility of a directory is determined.

SEE ALSO

mkstemp(3), *mktemp*(3), *tmpfile*(3), *tmpnam*(3)

NAME

termios, tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfmakeraw, cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed, cfsetspeed − get and set terminal attributes, line control, get and set baud rate

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);

int tcsendbreak(int fd, int duration);
int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcflow(int fd, int action);

void cfmakeraw(struct termios *termios_p);

speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);

int cfsetispeed(struct termios *termios_p, speed_t speed);
int cfsetospeed(struct termios *termios_p, speed_t speed);
int cfsetspeed(struct termios *termios_p, speed_t speed);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

cfsetspeed(), **cfmakeraw**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The termios functions describe a general terminal interface that is provided to control asynchronous communications ports.

The termios structure

Many of the functions described here have a *termios_p* argument that is a pointer to a *termios* structure. This structure contains at least the following members:

```
tcflag_t c_iflag;      /* input modes */
tcflag_t c_oflag;      /* output modes */
tcflag_t c_cflag;      /* control modes */
tcflag_t c_lflag;      /* local modes */
cc_t     c_cc[NCCS];   /* special characters */
```

The values that may be assigned to these fields are described below. In the case of the first four bit-mask fields, the definitions of some of the associated flags that may be set are exposed only if a specific feature test macro (see *feature_test_macros*(7)) is defined, as noted in brackets ("[]").

In the descriptions below, "not in POSIX" means that the value is not specified in POSIX.1-2001, and "XSI" means that the value is specified in POSIX.1-2001 as part of the XSI extension.

c_iflag flag constants:

IGNBRK
Ignore BREAK condition on input.

BRKINT
If **IGNBRK** is set, a BREAK is ignored. If it is not set but **BRKINT** is set, then a BREAK causes the input and output queues to be flushed, and if the terminal is the controlling terminal of a foreground process group, it will cause a **SIGINT** to be sent to this foreground process group. When neither **IGNBRK** nor **BRKINT** are set, a BREAK reads as a null byte ('\0'), except when **PARMRK** is set, in which case it reads as the sequence \377 \0 \0.

IGNPAR
Ignore framing errors and parity errors.

PARMRK
If this bit is set, input bytes with parity or framing errors are marked when passed to the program. This bit is meaningful only when **INPCK** is set and **IGNPAR** is not set. The way erroneous bytes are marked is with two preceding bytes, \377 and \0. Thus, the program actually reads three bytes for one erroneous byte received from the terminal. If a valid byte has the value \377, and **ISTRIP** (see below) is not set, the program might confuse it with the prefix that marks a parity error. Therefore, a valid byte \377 is passed to the program as two bytes, \377 \377, in this case.

If neither **IGNPAR** nor **PARMRK** is set, read a character with a parity error or framing error as \0.

INPCK
Enable input parity checking.

ISTRIP
Strip off eighth bit.

INLCR
Translate NL to CR on input.

IGNCR
Ignore carriage return on input.

ICRNL
Translate carriage return to newline on input (unless **IGNCR** is set).

IUCLC
(not in POSIX) Map uppercase characters to lowercase on input.

IXON Enable XON/XOFF flow control on output.

IXANY
(XSI) Typing any character will restart stopped output. (The default is to allow just the START character to restart output.)

IXOFF
Enable XON/XOFF flow control on input.

IMAXBEL
(not in POSIX) Ring bell when input queue is full. Linux does not implement this bit, and acts as if it is always set.

IUTF8 (since Linux 2.6.4)
(not in POSIX) Input is UTF8; this allows character-erase to be correctly performed in cooked mode.

c_oflag flag constants:

OPOST
Enable implementation-defined output processing.

OLCUC
(not in POSIX) Map lowercase characters to uppercase on output.

ONLCR
(XSI) Map NL to CR-NL on output.

OCRNL
Map CR to NL on output.

ONOCR
Don’t output CR at column 0.

ONLRET
The NL character is assumed to do the carriage-return function; the kernel’s idea of the current column is set to 0 after both NL and CR.

OFILL
Send fill characters for a delay, rather than using a timed delay.

OFDEL
Fill character is ASCII DEL (0177). If unset, fill character is ASCII NUL ('\0'). (Not implemented on Linux.)

NLDLY
Newline delay mask. Values are **NL0** and **NL1**. [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

CRDLY
Carriage return delay mask. Values are **CR0**, **CR1**, **CR2**, or **CR3**. [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

TABDLY
Horizontal tab delay mask. Values are **TAB0**, **TAB1**, **TAB2**, **TAB3** (or **XTABS**, but see the **BUGS** section). A value of TAB3, that is, XTABS, expands tabs to spaces (with tab stops every eight columns). [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

BSDLY
Backspace delay mask. Values are **BS0** or **BS1**. (Has never been implemented.) [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

VTDLY
Vertical tab delay mask. Values are **VT0** or **VT1**.

FFDLY
Form feed delay mask. Values are **FF0** or **FF1**. [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

c_cflag flag constants:

CBAUD
(not in POSIX) Baud speed mask (4+1 bits). [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

CBAUDEX
(not in POSIX) Extra baud speed mask (1 bit), included in **CBAUD**. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

(POSIX says that the baud speed is stored in the *termios* structure without specifying where precisely, and provides **cfgetispeed**() and **cfsetispeed**() for getting at it. Some systems use bits selected by **CBAUD** in *c_cflag*, other systems use separate fields, for example, *sg_ispeed* and *sg_ospeed*.)

CSIZE Character size mask. Values are **CS5**, **CS6**, **CS7**, or **CS8**.

CSTOPB
Set two stop bits, rather than one.

CREAD
Enable receiver.

PARENB
Enable parity generation on output and parity checking for input.

PARODD
If set, then parity for input and output is odd; otherwise even parity is used.

HUPCL
Lower modem control lines after last process closes the device (hang up).

CLOCAL
Ignore modem control lines.

LOBLK
(not in POSIX) Block output from a noncurrent shell layer. For use by **shl** (shell layers). (Not implemented on Linux.)

CIBAUD
(not in POSIX) Mask for input speeds. The values for the **CIBAUD** bits are the same as the values for the **CBAUD** bits, shifted left **IBSHIFT** bits. [requires **_BSD_SOURCE** or **_SVID_SOURCE**] (Not implemented in glibc, supported on Linux via **TCGET*** and **TCSET*** ioctls; see *ioctl_tty*(2))

CMSPAR
(not in POSIX) Use "stick" (mark/space) parity (supported on certain serial devices): if **PARODD** is set, the parity bit is always 1; if **PARODD** is not set, then the parity bit is always 0. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

CRTSCTS
(not in POSIX) Enable RTS/CTS (hardware) flow control. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

c_lflag flag constants:

ISIG When any of the characters INTR, QUIT, SUSP, or DSUSP are received, generate the corresponding signal.

ICANON
Enable canonical mode (described below).

XCASE
(not in POSIX; not supported under Linux) If **ICANON** is also set, terminal is uppercase only. Input is converted to lowercase, except for characters preceded by \. On output, uppercase characters are preceded by \ and lowercase characters are converted to uppercase. [requires **_BSD_SOURCE** or **_SVID_SOURCE** or **_XOPEN_SOURCE**]

ECHO Echo input characters.

ECHOE
If **ICANON** is also set, the ERASE character erases the preceding input character, and WERASE erases the preceding word.

ECHOK
If **ICANON** is also set, the KILL character erases the current line.

ECHONL
If **ICANON** is also set, echo the NL character even if ECHO is not set.

ECHOCTL
(not in POSIX) If **ECHO** is also set, terminal special characters other than TAB, NL, START, and STOP are echoed as **^X**, where X is the character with ASCII code 0x40 greater than the special character. For example, character 0x08 (BS) is echoed as **^H**. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

ECHOPRT
(not in POSIX) If **ICANON** and **ECHO** are also set, characters are printed as they are being erased. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

ECHOKE
(not in POSIX) If **ICANON** is also set, KILL is echoed by erasing each character on the line, as specified by **ECHOE** and **ECHOPRT**. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

DEFECHO
(not in POSIX) Echo only when a process is reading. (Not implemented on Linux.)

FLUSHO
(not in POSIX; not supported under Linux) Output is being flushed. This flag is toggled by typing the DISCARD character. [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

NOFLSH
Disable flushing the input and output queues when generating signals for the INT, QUIT, and SUSP characters.

TOSTOP
Send the **SIGTTOU** signal to the process group of a background process which tries to write to its controlling terminal.

PENDIN
(not in POSIX; not supported under Linux) All characters in the input queue are reprinted when the next character is read. (**bash**(1) handles typeahead this way.) [requires **_BSD_SOURCE** or **_SVID_SOURCE**]

IEXTEN
Enable implementation-defined input processing. This flag, as well as **ICANON** must be enabled for the special characters EOL2, LNEXT, REPRINT, WERASE to be interpreted, and for the **IUCLC** flag to be effective.

The *c_cc* array defines the terminal special characters. The symbolic indices (initial values) and meaning are:

VDISCARD
(not in POSIX; not supported under Linux; 017, SI, Ctrl-O) Toggle: start/stop discarding pending output. Recognized when **IEXTEN** is set, and then not passed as input.

VDSUSP
(not in POSIX; not supported under Linux; 031, EM, Ctrl-Y) Delayed suspend character (DSUSP): send **SIGTSTP** signal when the character is read by the user program. Recognized when **IEXTEN** and **ISIG** are set, and the system supports job control, and then not passed as input.

VEOF (004, EOT, Ctrl-D) End-of-file character (EOF). More precisely: this character causes the pending tty buffer to be sent to the waiting user program without waiting for end-of-line. If it is the first character of the line, the *read*(2) in the user program returns 0, which signifies end-of-file. Recognized when **ICANON** is set, and then not passed as input.

VEOL (0, NUL) Additional end-of-line character (EOL). Recognized when **ICANON** is set.

VEOL2
(not in POSIX; 0, NUL) Yet another end-of-line character (EOL2). Recognized when **ICANON** is set.

VERASE
(0177, DEL, rubout, or 010, BS, Ctrl-H, or also #) Erase character (ERASE). This erases the previous not-yet-erased character, but does not erase past EOF or beginning-of-line. Recognized when **ICANON** is set, and then not passed as input.

VINTR
(003, ETX, Ctrl-C, or also 0177, DEL, rubout) Interrupt character (INTR). Send a **SIGINT** signal. Recognized when **ISIG** is set, and then not passed as input.

VKILL
(025, NAK, Ctrl-U, or Ctrl-X, or also @) Kill character (KILL). This erases the input since the last EOF or beginning-of-line. Recognized when **ICANON** is set, and then not passed as input.

VLNEXT
(not in POSIX; 026, SYN, Ctrl-V) Literal next (LNEXT). Quotes the next input character, depriving it of a possible special meaning. Recognized when **IEXTEN** is set, and then not passed as input.

VMIN Minimum number of characters for noncanonical read (MIN).

VQUIT
(034, FS, Ctrl-\) Quit character (QUIT). Send **SIGQUIT** signal. Recognized when **ISIG** is set, and then not passed as input.

VREPRINT
(not in POSIX; 022, DC2, Ctrl-R) Reprint unread characters (REPRINT). Recognized when **ICANON** and **IEXTEN** are set, and then not passed as input.

VSTART
(021, DC1, Ctrl-Q) Start character (START). Restarts output stopped by the Stop character. Recognized when **IXON** is set, and then not passed as input.

VSTATUS
(not in POSIX; not supported under Linux; status request: 024, DC4, Ctrl-T). Status character (STATUS). Display status information at terminal, including state of foreground process and amount of CPU time it has consumed. Also sends a **SIGINFO** signal (not supported on Linux) to the foreground process group.

VSTOP
(023, DC3, Ctrl-S) Stop character (STOP). Stop output until Start character typed. Recognized when **IXON** is set, and then not passed as input.

VSUSP
(032, SUB, Ctrl-Z) Suspend character (SUSP). Send **SIGTSTP** signal. Recognized when **ISIG** is set, and then not passed as input.

VSWTCH
(not in POSIX; not supported under Linux; 0, NUL) Switch character (SWTCH). Used in System V to switch shells in *shell layers*, a predecessor to shell job control.

VTIME
Timeout in deciseconds for noncanonical read (TIME).

VWERASE
(not in POSIX; 027, ETB, Ctrl-W) Word erase (WERASE). Recognized when **ICANON** and **IEXTEN** are set, and then not passed as input.

An individual terminal special character can be disabled by setting the value of the corresponding *c_cc* element to **_POSIX_VDISABLE**.

The above symbolic subscript values are all different, except that **VTIME**, **VMIN** may have the same value as **VEOL**, **VEOF**, respectively. In noncanonical mode the special character meaning is replaced by the timeout meaning. For an explanation of **VMIN** and **VTIME**, see the description of noncanonical mode below.

Retrieving and changing terminal settings

tcgetattr() gets the parameters associated with the object referred by *fd* and stores them in the *termios* structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

tcsetattr() sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the *termios* structure referred to by *termios_p*. *optional_actions* specifies when the changes take effect:

TCSANOW
the change occurs immediately.

TCSADRAIN
the change occurs after all output written to *fd* has been transmitted. This option should be used when changing parameters that affect output.

TCSAFLUSH
the change occurs after all output written to the object referred by *fd* has been transmitted, and all input that has been received but not read will be discarded before the change is made.

Canonical and noncanonical mode

The setting of the **ICANON** canon flag in *c_lflag* determines whether the terminal is operating in canonical mode (**ICANON** set) or noncanonical mode (**ICANON** unset). By default, **ICANON** is set.

In canonical mode:

- Input is made available line by line. An input line is available when one of the line delimiters is typed (NL, EOL, EOL2; or EOF at the start of line). Except in the case of EOF, the line delimiter is included in the buffer returned by *read*(2).
- Line editing is enabled (ERASE, KILL; and if the **IEXTEN** flag is set: WERASE, REPRINT, LNEXT). A *read*(2) returns at most one line of input; if the *read*(2) requested fewer bytes than are available in the current line of input, then only as many bytes as requested are read, and the remaining characters will be available for a future *read*(2).

- The maximum line length is 4096 chars (including the terminating newline character); lines longer than 4096 chars are truncated. After 4095 characters, input processing (e.g., **ISIG** and **ECHO*** processing) continues, but any input data after 4095 characters up to (but not including) any terminating newline is discarded. This ensures that the terminal can always receive more input until at least one line can be read.

In noncanonical mode input is available immediately (without the user having to type a line-delimiter character), no input processing is performed, and line editing is disabled. The read buffer will only accept 4095 chars; this provides the necessary space for a newline char if the input mode is switched to canonical. The settings of MIN (*c_cc[VMIN]*) and TIME (*c_cc[VTIME]*) determine the circumstances in which a *read*(2) completes; there are four distinct cases:

MIN == 0, TIME == 0 (polling read)
: If data is available, *read*(2) returns immediately, with the lesser of the number of bytes available, or the number of bytes requested. If no data is available, *read*(2) returns 0.

MIN > 0, TIME == 0 (blocking read)
: *read*(2) blocks until MIN bytes are available, and returns up to the number of bytes requested.

MIN == 0, TIME > 0 (read with timeout)
: TIME specifies the limit for a timer in tenths of a second. The timer is started when *read*(2) is called. *read*(2) returns either when at least one byte of data is available, or when the timer expires. If the timer expires without any input becoming available, *read*(2) returns 0. If data is already available at the time of the call to *read*(2), the call behaves as though the data was received immediately after the call.

MIN > 0, TIME > 0 (read with interbyte timeout)
: TIME specifies the limit for a timer in tenths of a second. Once an initial byte of input becomes available, the timer is restarted after each further byte is received. *read*(2) returns when any of the following conditions is met:

  - MIN bytes have been received.
  - The interbyte timer expires.
  - The number of bytes requested by *read*(2) has been received. (POSIX does not specify this termination condition, and on some other implementations *read*(2) does not return in this case.)

  Because the timer is started only after the initial byte becomes available, at least one byte will be read. If data is already available at the time of the call to *read*(2), the call behaves as though the data was received immediately after the call.

POSIX does not specify whether the setting of the **O_NONBLOCK** file status flag takes precedence over the MIN and TIME settings. If **O_NONBLOCK** is set, a *read*(2) in noncanonical mode may return immediately, regardless of the setting of MIN or TIME. Furthermore, if no data is available, POSIX permits a *read*(2) in noncanonical mode to return either 0, or −1 with *errno* set to **EAGAIN**.

Raw mode

cfmakeraw() sets the terminal to something like the "raw" mode of the old Version 7 terminal driver: input is available character by character, echoing is disabled, and all special processing of terminal input and output characters is disabled. The terminal attributes are set as follows:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                 | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

Line control

tcsendbreak() transmits a continuous stream of zero-valued bits for a specific duration, if the terminal is using asynchronous serial data transmission. If *duration* is zero, it transmits zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not zero, it sends zero-valued bits for some implementation-defined length of time.

If the terminal is not using asynchronous serial data transmission, **tcsendbreak**() returns without taking

any action.

tcdrain() waits until all output written to the object referred to by *fd* has been transmitted.

tcflush() discards data written to the object referred to by *fd* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

TCIFLUSH
flushes data received but not read.

TCOFLUSH
flushes data written but not transmitted.

TCIOFLUSH
flushes both data received but not read, and data written but not transmitted.

tcflow() suspends transmission or reception of data on the object referred to by *fd*, depending on the value of *action*:

TCOOFF
suspends output.

TCOON
restarts suspended output.

TCIOFF
transmits a STOP character, which stops the terminal device from transmitting data to the system.

TCION
transmits a START character, which starts the terminal device transmitting data to the system.

The default on open of a terminal file is that neither its input nor its output is suspended.

Line speed

The baud rate functions are provided for getting and setting the values of the input and output baud rates in the *termios* structure. The new values do not take effect until **tcsetattr**() is successfully called.

Setting the speed to **B0** instructs the modem to "hang up". The actual bit rate corresponding to **B38400** may be altered with *setserial*(8)

The input and output baud rates are stored in the *termios* structure.

cfgetospeed() returns the output baud rate stored in the *termios* structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the *termios* structure pointed to by *termios_p* to *speed*, which must be one of these constants:

B0
B50
B75
B110
B134
B150
B200
B300
B600
B1200
B1800
B2400
B4800
B9600
B19200
B38400
B57600
B115200
B230400

B460800
B500000
B576000
B921600
B1000000
B1152000
B1500000
B2000000

These constants are additionally supported on the SPARC architecture:

B76800
B153600
B307200
B614400

These constants are additionally supported on non-SPARC architectures:

B2500000
B3000000
B3500000
B4000000

Due to differences between architectures, portable applications should check if a particular **B***nnn* constant is defined prior to using it.

The zero baud rate, **B0**, is used to terminate the connection. If **B0** is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line. **CBAUDEX** is a mask for the speeds beyond those defined in POSIX.1 (57600 and above). Thus, **B57600** & **CBAUDEX** is nonzero.

Setting the baud rate to a value other than those defined by **B***nnn* constants is possible via the **TCSETS2** ioctl; see *ioctl_tty*(2).

cfgetispeed() returns the input baud rate stored in the *termios* structure.

cfsetispeed() sets the input baud rate stored in the *termios* structure to *speed*, which must be specified as one of the **B***nnn* constants listed above for **cfsetospeed**(). If the input baud rate is set to the literal constant **0** (not the symbolic constant **B0**), the input baud rate will be equal to the output baud rate.

cfsetspeed() is a 4.4BSD extension. It takes the same arguments as **cfsetispeed**(), and sets both input and output speed.

RETURN VALUE

cfgetispeed() returns the input baud rate stored in the *termios* structure.

cfgetospeed() returns the output baud rate stored in the *termios* structure.

All other functions return:

0 on success.

−1 on failure and set *errno* to indicate the error.

Note that **tcsetattr**() returns success if *any* of the requested changes could be successfully carried out. Therefore, when making multiple changes it may be necessary to follow this call with a further call to **tcgetattr**() to check that all changes have been performed successfully.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tcgetattr(), **tcsetattr**(), **tcdrain**(), **tcflush**(), **tcflow**(), **tcsendbreak**(), **cfmakeraw**(), **cfgetispeed**(), **cfgetospeed**(), **cfsetispeed**(), **cfsetospeed**(), **cfsetspeed**()	Thread safety	MT-Safe

STANDARDS

tcgetattr()
tcsetattr()

tcsendbreak()
tcdrain()
tcflush()
tcflow()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
POSIX.1-2008.

cfmakeraw()
cfsetspeed()
BSD.

HISTORY

tcgetattr()
tcsetattr()
tcsendbreak()
tcdrain()
tcflush()
tcflow()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
POSIX.1-2001.

cfmakeraw()
cfsetspeed()
BSD.

NOTES

UNIX V7 and several later systems have a list of baud rates where after the values **B0** through **B9600** one finds the two constants **EXTA**, **EXTB** ("External A" and "External B"). Many systems extend the list with much higher baud rates.

The effect of a nonzero *duration* with **tcsendbreak**() varies. SunOS specifies a break of *duration* * *N* seconds, where *N* is at least 0.25, and not more than 0.5. Linux, AIX, DU, Tru64 send a break of *duration* milliseconds. FreeBSD and NetBSD and HP-UX and MacOS ignore the value of *duration*. Under Solaris and UnixWare, **tcsendbreak**() with nonzero *duration* behaves like **tcdrain**().

BUGS

On the Alpha architecture before Linux 4.16 (and glibc before glibc 2.28), the **XTABS** value was different from **TAB3** and it was ignored by the **N_TTY** line discipline code of the terminal driver as a result (because as it wasn't part of the **TABDLY** mask).

SEE ALSO

reset(1), *setterm*(1), *stty*(1), *tput*(1), *tset*(1), *tty*(1), *ioctl_console*(2), *ioctl_tty*(2), *cc_t*(3type), *speed_t*(3type), *tcflag_t*(3type), *setserial*(8)

NAME

tgamma, tgammaf, tgammal − true gamma function

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <math.h>

double tgamma(double *x***);**
float tgammaf(float *x***);**
long double tgammal(long double *x***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tgamma(), **tgammaf**(), **tgammal**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions calculate the Gamma function of *x*.

The Gamma function is defined by

Gamma(x) = integral from 0 to infinity of t^(x−1) e^−t dt

It is defined for every real number except for nonpositive integers. For nonnegative integral *m* one has

Gamma(m+1) = m!

and, more generally, for all *x*:

Gamma(x+1) = x * Gamma(x)

Furthermore, the following is valid for all values of *x* outside the poles:

Gamma(x) * Gamma(1 − x) = PI / sin(PI * x)

RETURN VALUE

On success, these functions return Gamma(x).

If *x* is a NaN, a NaN is returned.

If *x* is positive infinity, positive infinity is returned.

If *x* is a negative integer, or is negative infinity, a domain error occurs, and a NaN is returned.

If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the correct mathematical sign.

If the result underflows, a range error occurs, and the functions return 0, with the correct mathematical sign.

If *x* is −0 or +0, a pole error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, respectively, with the same sign as the 0.

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is a negative integer, or negative infinity
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised (but see BUGS).

Pole error: *x* is +0 or −0
: *errno* is set to **ERANGE**. A divide-by-zero floating-point exception (**FE_DIVBYZERO**) is raised.

Range error: result overflow
: *errno* is set to **ERANGE**. An overflow floating-point exception (**FE_OVERFLOW**) is raised.

glibc also gives the following error which is not specified in C99 or POSIX.1-2001.

Range error: result underflow
: An underflow floating-point exception (**FE_UNDERFLOW**) is raised, and *errno* is set to **ERANGE**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tgamma(), **tgammaf**(), **tgammal**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

NOTES

This function had to be called "true gamma function" since there is already a function *gamma*(3) that returns something else (see *gamma*(3) for details).

BUGS

Before glibc 2.18, the glibc implementation of these functions did not set *errno* to **EDOM** when *x* is negative infinity.

Before glibc 2.19, the glibc implementation of these functions did not set *errno* to **ERANGE** on an underflow range error.

In glibc versions 2.3.3 and earlier, an argument of +0 or −0 incorrectly produced a domain error (*errno* set to **EDOM** and an **FE_INVALID** exception raised), rather than a pole error.

SEE ALSO

gamma(3), *lgamma*(3)

NAME

timegm, timelocal − inverses of gmtime and localtime

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

[[deprecated]] time_t timelocal(struct tm *tm);
time_t timegm(struct tm *tm);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

timelocal(), **timegm**():
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

The functions **timelocal**() and **timegm**() are the inverses of *localtime*(3) and *gmtime*(3). Both functions take a broken-down time and convert it to calendar time (seconds since the Epoch, 1970-01-01 00:00:00 +0000, UTC). The difference between the two functions is that **timelocal**() takes the local timezone into account when doing the conversion, while **timegm**() takes the input value to be Coordinated Universal Time (UTC).

RETURN VALUE

On success, these functions return the calendar time (seconds since the Epoch), expressed as a value of type *time_t*. On error, they return the value *(time_t) −1* and set *errno* to indicate the error.

ERRORS

EOVERFLOW
The result cannot be represented.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
timelocal(), **timegm**()	Thread safety	MT-Safe env locale

STANDARDS

BSD.

HISTORY

GNU, BSD.

The **timelocal**() function is equivalent to the POSIX standard function *mktime*(3). There is no reason to ever use it.

SEE ALSO

gmtime(3), *localtime*(3), *mktime*(3), *tzset*(3)

NAME

timeradd, timersub, timercmp, timerclear, timerisset − timeval operations

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <sys/time.h>

void timeradd(struct timeval *a, struct timeval *b,
        struct timeval *res);
void timersub(struct timeval *a, struct timeval *b,
        struct timeval *res);

void timerclear(struct timeval *tvp);
int timerisset(struct timeval *tvp);

int timercmp(struct timeval *a, struct timeval *b, CMP);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
Since glibc 2.19:
_DEFAULT_SOURCE
glibc 2.19 and earlier:
_BSD_SOURCE

DESCRIPTION

The macros are provided to operate on *timeval* structures, defined in <*sys/time.h*> as:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

timeradd() adds the time values in *a* and *b*, and places the sum in the *timeval* pointed to by *res*. The result is normalized such that *res−>tv_usec* has a value in the range 0 to 999,999.

timersub() subtracts the time value in *b* from the time value in *a*, and places the result in the *timeval* pointed to by *res*. The result is normalized such that *res−>tv_usec* has a value in the range 0 to 999,999.

timerclear() zeros out the *timeval* structure pointed to by *tvp*, so that it represents the Epoch: 1970-01-01 00:00:00 +0000 (UTC).

timerisset() returns true (nonzero) if either field of the *timeval* structure pointed to by *tvp* contains a nonzero value.

timercmp() compares the timer values in *a* and *b* using the comparison operator *CMP*, and returns true (nonzero) or false (0) depending on the result of the comparison. Some systems (but not Linux/glibc), have a broken **timercmp**() implementation, in which *CMP* of >=, <=, and == do not work; portable applications can instead use

```
!timercmp(..., <)
!timercmp(..., >)
!timercmp(..., !=)
```

RETURN VALUE

timerisset() and **timercmp**() return true (nonzero) or false (0).

ERRORS

No errors are defined.

STANDARDS

None.

HISTORY

BSD.

SEE ALSO

gettimeofday(2), *time*(7)

NAME

TIMEVAL_TO_TIMESPEC, TIMESPEC_TO_TIMEVAL − convert between time structures

SYNOPSIS

```
#define _GNU_SOURCE
#include <sys/time.h>

void TIMEVAL_TO_TIMESPEC(const struct timeval *tv, struct timespec *ts);
void TIMESPEC_TO_TIMEVAL(struct timeval *tv, const struct timespec *ts);
```

DESCRIPTION

These macros convert from a *timeval*(3type) to a *timespec*(3type) structure, and vice versa, respectively.

This is especially useful for writing interfaces that receive a type, but are implemented with calls to functions that receive the other one.

STANDARDS

GNU, BSD.

NAME

tmpfile − create a temporary file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdio.h>

FILE *tmpfile(void);

DESCRIPTION

The **tmpfile**() function opens a unique temporary file in binary read/write (w+b) mode. The file will be automatically deleted when it is closed or the program terminates.

RETURN VALUE

The **tmpfile**() function returns a stream descriptor, or NULL if a unique filename cannot be generated or the unique file cannot be opened. In the latter case, *errno* is set to indicate the error.

ERRORS

EACCES
Search permission denied for directory in file's path prefix.

EEXIST
Unable to generate a unique filename.

EINTR
The call was interrupted by a signal; see *signal*(7).

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOSPC
There was no room in the directory to add the new filename.

EROFS
Read-only filesystem.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tmpfile()	Thread safety	MT-Safe

VERSIONS

The standard does not specify the directory that **tmpfile**() will use. glibc will try the path prefix *P_tmpdir* defined in <*stdio.h*>, and if that fails, then the directory */tmp*.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89, SVr4, 4.3BSD, SUSv2.

NOTES

POSIX.1-2001 specifies: an error message may be written to *stdout* if the stream cannot be opened.

SEE ALSO

exit(3), *mkstemp*(3), *mktemp*(3), *tempnam*(3), *tmpnam*(3)

NAME

tmpnam, tmpnam_r − create a name for a temporary file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

[[deprecated]] char *tmpnam(char *s);
[[deprecated]] char *tmpnam_r(char *s);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tmpnam_r()
Since glibc 2.19:
_DEFAULT_SOURCE
Up to and including glibc 2.19:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Note: avoid using these functions; use *mkstemp*(3) or *tmpfile*(3) instead.

The **tmpnam**() function returns a pointer to a string that is a valid filename, and such that a file with this name did not exist at some point in time, so that naive programmers may think it a suitable name for a temporary file. If the argument *s* is NULL, this name is generated in an internal static buffer and may be overwritten by the next call to **tmpnam**(). If *s* is not NULL, the name is copied to the character array (of length at least *L_tmpnam*) pointed to by *s* and the value *s* is returned in case of success.

The created pathname has a directory prefix *P_tmpdir*. (Both *L_tmpnam* and *P_tmpdir* are defined in *<stdio.h>*, just like the **TMP_MAX** mentioned below.)

The **tmpnam_r**() function performs the same task as **tmpnam**(), but returns NULL (to indicate an error) if *s* is NULL.

RETURN VALUE

These functions return a pointer to a unique temporary filename, or NULL if a unique name cannot be generated.

ERRORS

No errors are defined.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tmpnam()	Thread safety	MT-Unsafe race:tmpnam/!s
tmpnam_r()	Thread safety	MT-Safe

STANDARDS

tmpnam()
C11, POSIX.1-2008.

tmpnam_r()
None.

HISTORY

tmpnam()
SVr4, 4.3BSD, C89, POSIX.1-2001. Obsolete in POSIX.1-2008.

tmpnam_r()
Solaris.

NOTES

The **tmpnam**() function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, the behavior is implementation defined.

Although these functions generate names that are difficult to guess, it is nevertheless possible that between the time that the pathname is returned and the time that the program opens it, another program might create that pathname using *open*(2), or create it as a symbolic link. This can lead to security

holes. To avoid such possibilities, use the *open*(2) **O_EXCL** flag to open the pathname. Or better yet, use *mkstemp*(3) or *tmpfile*(3).

Portable applications that use threads cannot call **tmpnam**() with a NULL argument if either **_POSIX_THREADS** or **_POSIX_THREAD_SAFE_FUNCTIONS** is defined.

BUGS

Never use these functions. Use *mkstemp*(3) or *tmpfile*(3) instead.

SEE ALSO

mkstemp(3), *mktemp*(3), *tempnam*(3), *tmpfile*(3)

NAME

toascii − convert character to ASCII

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <ctype.h>

[[deprecated]] int toascii(int c);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

toascii():
_XOPEN_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

toascii() converts *c* to a 7-bit *unsigned char* value that fits into the ASCII character set, by clearing the high-order bits.

RETURN VALUE

The value returned is that of the converted character.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
toascii()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

SVr4, BSD, POSIX.1-2001. Obsolete in POSIX.1-2008, noting that it cannot be used portably in a localized application.

BUGS

Many people will be unhappy if you use this function. This function will convert accented letters into random characters.

SEE ALSO

isascii(3), *tolower*(3), *toupper*(3)

NAME

toupper, tolower, toupper_l, tolower_l − convert uppercase or lowercase

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <ctype.h>

int toupper(int c);
int tolower(int c);

int toupper_l(int c, locale_t locale);
int tolower_l(int c, locale_t locale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

toupper_l(), **tolower_l**():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

These functions convert lowercase letters to uppercase, and vice versa.

If *c* is a lowercase letter, **toupper**() returns its uppercase equivalent, if an uppercase representation exists in the current locale. Otherwise, it returns *c*. The **toupper_l**() function performs the same task, but uses the locale referred to by the locale handle *locale*.

If *c* is an uppercase letter, **tolower**() returns its lowercase equivalent, if a lowercase representation exists in the current locale. Otherwise, it returns *c*. The **tolower_l**() function performs the same task, but uses the locale referred to by the locale handle *locale*.

If *c* is neither an *unsigned char* value nor **EOF**, the behavior of these functions is undefined.

The behavior of **toupper_l**() and **tolower_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)) or is not a valid locale object handle.

RETURN VALUE

The value returned is that of the converted letter, or *c* if the conversion was not possible.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
toupper(), **tolower**(), **toupper_l**(), **tolower_l**()	Thread safety	MT-Safe

STANDARDS

toupper()
tolower()
C11, POSIX.1-2008.

toupper_l()
tolower_l()
POSIX.1-2008.

HISTORY

toupper()
tolower()
C89, 4.3BSD, POSIX.1-2001.

toupper_l()
tolower_l()
POSIX.1-2008.

NOTES

The standards require that the argument *c* for these functions is either **EOF** or a value that is representable in the type *unsigned char*. If the argument *c* is of type *char*, it must be cast to *unsigned char*, as in the following example:

```
char c;
...
res = toupper((unsigned char) c);
```

This is necessary because *char* may be the equivalent *signed char*, in which case a byte where the top bit is set would be sign extended when converting to *int*, yielding a value that is outside the range of *unsigned char*.

The details of what constitutes an uppercase or lowercase letter depend on the locale. For example, the default "**C**" locale does not know about umlauts, so no conversion is done for them.

In some non-English locales, there are lowercase letters with no corresponding uppercase equivalent; the German sharp s is one example.

SEE ALSO

isalpha(3), *newlocale*(3), *setlocale*(3), *towlower*(3), *towupper*(3), *uselocale*(3), *locale*(7)

NAME

towctrans − wide-character transliteration

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

wint_t towctrans(wint_t *wc***, wctrans_t** *desc***);**

DESCRIPTION

If *wc* is a wide character, then the **towctrans**() function translates it according to the transliteration descriptor *desc*. If *wc* is **WEOF**, **WEOF** is returned.

desc must be a transliteration descriptor returned by the *wctrans*(3) function.

RETURN VALUE

The **towctrans**() function returns the translated wide character, or **WEOF** if *wc* is **WEOF**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
towctrans()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **towctrans**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

towlower(3), *towupper*(3), *wctrans*(3)

NAME

towlower, towlower_l − convert a wide character to lowercase

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wctype.h>

wint_t towlower(wint_t wc);
wint_t towlower_l(wint_t wc, locale_t locale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

towlower_l():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **towlower**() function is the wide-character equivalent of the *tolower*(3) function. If *wc* is an uppercase wide character, and there exists a lowercase equivalent in the current locale, it returns the lowercase equivalent of *wc*. In all other cases, *wc* is returned unchanged.

The **towlower_l**() function performs the same task, but performs the conversion based on the character type information in the locale specified by *locale*. The behavior of **towlower_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)) or is not a valid locale object handle.

The argument *wc* must be representable as a *wchar_t* and be a valid character in the locale or be the value **WEOF**.

RETURN VALUE

If *wc* was convertible to lowercase, **towlower**() returns its lowercase equivalent; otherwise it returns *wc*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
towlower()	Thread safety	MT-Safe locale
towlower_l()	Thread safety	MT-Safe

STANDARDS

towlower()
C11, POSIX.1-2008 (XSI).

towlower_l()
POSIX.1-2008.

STANDARDS

towlower()
C99, POSIX.1-2001 (XSI). Obsolete in POSIX.1-2008 (XSI).

towlower_l()
glibc 2.3. POSIX.1-2008.

NOTES

The behavior of these functions depends on the **LC_CTYPE** category of the locale.

These functions are not very appropriate for dealing with Unicode characters, because Unicode knows about three cases: upper, lower, and title case.

SEE ALSO

iswlower(3), *towctrans*(3), *towupper*(3), *locale*(7)

NAME

towupper, towupper_l − convert a wide character to uppercase

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wctype.h>

wint_t towupper(wint_t wc);
wint_t towupper_l(wint_t wc, locale_t locale);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

towupper_l():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **towupper**() function is the wide-character equivalent of the *toupper*(3) function. If *wc* is a lowercase wide character, and there exists an uppercase equivalent in the current locale, it returns the uppercase equivalent of *wc*. In all other cases, *wc* is returned unchanged.

The **towupper_l**() function performs the same task, but performs the conversion based on the character type information in the locale specified by *locale*. The behavior of **towupper_l**() is undefined if *locale* is the special locale object **LC_GLOBAL_LOCALE** (see *duplocale*(3)) or is not a valid locale object handle.

The argument *wc* must be representable as a *wchar_t* and be a valid character in the locale or be the value **WEOF**.

RETURN VALUE

If *wc* was convertible to uppercase, **towupper**() returns its uppercase equivalent; otherwise it returns *wc*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
towupper()	Thread safety	MT-Safe locale
towupper_l()	Thread safety	MT-Safe

STANDARDS

towupper()
C11, POSIX.1-2008 (XSI).

towupper_l()
POSIX.1-2008.

HISTORY

towupper()
C99, POSIX.1-2001 (XSI). Obsolete in POSIX.1-2008 (XSI).

towupper_l()
POSIX.1-2008. glibc 2.3.

NOTES

The behavior of these functions depends on the **LC_CTYPE** category of the locale.

These functions are not very appropriate for dealing with Unicode characters, because Unicode knows about three cases: upper, lower, and title case.

SEE ALSO

iswupper(3), *towctrans*(3), *towlower*(3), *locale*(7)

NAME

trunc, truncf, truncl − round to integer, toward zero

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

```
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

trunc(), **truncf**(), **truncl**():
_ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

These functions round *x* to the nearest integer value that is not larger in magnitude than *x*.

RETURN VALUE

These functions return the rounded integer value, in floating format.

If *x* is integral, infinite, or NaN, *x* itself is returned.

ERRORS

No errors occur.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
trunc(), **truncf**(), **truncl**()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

glibc 2.1. C99, POSIX.1-2001.

NOTES

The integral value returned by these functions may be too large to store in an integer type (*int*, *long*, etc.). To avoid an overflow, which will produce undefined results, an application should perform a range check on the returned value before assigning it to an integer type.

SEE ALSO

ceil(3), *floor*(3), *lrint*(3), *nearbyint*(3), *rint*(3), *round*(3)

NAME

tsearch, tfind, tdelete, twalk, twalk_r, tdestroy − manage a binary search tree

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <search.h>

typedef enum { preorder, postorder, endorder, leaf } VISIT;

void *tsearch(const void *key, void **rootp,
        int (*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
        int (*compar)(const void *, const void *));
void *tdelete(const void *restrict key, void **restrict rootp,
        int (*compar)(const void *, const void *));
void twalk(const void *root,
        void (*action)(const void *nodep, VISIT which,
                int depth));

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <search.h>

void twalk_r(const void *root,
        void (*action)(const void *nodep, VISIT which,
                void *closure),
        void *closure);
void tdestroy(void *root, void (*free_node)(void *nodep));
```

DESCRIPTION

tsearch(), **tfind**(), **twalk**(), and **tdelete**() manage a binary search tree. They are generalized from Knuth (6.2.2) Algorithm T. The first field in each node of the tree is a pointer to the corresponding data item. (The calling program must store the actual data.) *compar* points to a comparison routine, which takes pointers to two items. It should return an integer which is negative, zero, or positive, depending on whether the first item is less than, equal to, or greater than the second.

tsearch() searches the tree for an item. *key* points to the item to be searched for. *rootp* points to a variable which points to the root of the tree. If the tree is empty, then the variable that *rootp* points to should be set to NULL. If the item is found in the tree, then **tsearch**() returns a pointer to the corresponding tree node. (In other words, **tsearch**() returns a pointer to a pointer to the data item.) If the item is not found, then **tsearch**() adds it, and returns a pointer to the corresponding tree node.

tfind() is like **tsearch**(), except that if the item is not found, then **tfind**() returns NULL.

tdelete() deletes an item from the tree. Its arguments are the same as for **tsearch**().

twalk() performs depth-first, left-to-right traversal of a binary tree. *root* points to the starting node for the traversal. If that node is not the root, then only part of the tree will be visited. **twalk**() calls the user function *action* each time a node is visited (that is, three times for an internal node, and once for a leaf). *action*, in turn, takes three arguments. The first argument is a pointer to the node being visited. The structure of the node is unspecified, but it is possible to cast the pointer to a pointer-to-pointer-to-element in order to access the element stored within the node. The application must not modify the structure pointed to by this argument. The second argument is an integer which takes one of the values **preorder**, **postorder**, or **endorder** depending on whether this is the first, second, or third visit to the internal node, or the value **leaf** if this is the single visit to a leaf node. (These symbols are defined in *<search.h>*.) The third argument is the depth of the node; the root node has depth zero.

(More commonly, **preorder**, **postorder**, and **endorder** are known as **preorder**, **inorder**, and **postorder**: before visiting the children, after the first and before the second, and after visiting the children. Thus, the choice of name **postorder** is rather confusing.)

twalk_r() is similar to **twalk**(), but instead of the *depth* argument, the *closure* argument pointer is passed to each invocation of the action callback, unchanged. This pointer can be used to pass information to and from the callback function in a thread-safe fashion, without resorting to global variables.

tdestroy() removes the whole tree pointed to by *root*, freeing all resources allocated by the **tsearch**()

function. For the data in each tree node the function *free_node* is called. The pointer to the data is passed as the argument to the function. If no such work is necessary, *free_node* must point to a function doing nothing.

RETURN VALUE

tsearch() returns a pointer to a matching node in the tree, or to the newly added node, or NULL if there was insufficient memory to add the item. **tfind**() returns a pointer to the node, or NULL if no match is found. If there are multiple items that match the key, the item whose node is returned is unspecified.

tdelete() returns a pointer to the parent of the node deleted, or NULL if the item was not found. If the deleted node was the root node, **tdelete**() returns a dangling pointer that must not be accessed.

tsearch(), **tfind**(), and **tdelete**() also return NULL if *rootp* was NULL on entry.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tsearch(), **tfind**(), **tdelete**()	Thread safety	MT-Safe race:rootp
twalk()	Thread safety	MT-Safe race:root
twalk_r()	Thread safety	MT-Safe race:root
tdestroy()	Thread safety	MT-Safe

STANDARDS

tsearch()
tfind()
tdelete()
twalk() POSIX.1-2008.

tdestroy()
twalk_r()
GNU.

HISTORY

tsearch()
tfind()
tdelete()
twalk() POSIX.1-2001, POSIX.1-2008, SVr4.

twalk_r()
glibc 2.30.

NOTES

twalk() takes a pointer to the root, while the other functions take a pointer to a variable which points to the root.

tdelete() frees the memory required for the node in the tree. The user is responsible for freeing the memory for the corresponding data.

The example program depends on the fact that **twalk**() makes no further reference to a node after calling the user function with argument "endorder" or "leaf". This works with the GNU library implementation, but is not in the System V documentation.

EXAMPLES

The following program inserts twelve random numbers into a binary tree, where duplicate numbers are collapsed, then prints the numbers in order.

```
#define _GNU_SOURCE     /* Expose declaration of tdestroy() */
#include <search.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

static void *root = NULL;

static void *
```

```
xmalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p)
        return p;
    fprintf(stderr, "insufficient memory\n");
    exit(EXIT_FAILURE);
}

static int
compare(const void *pa, const void *pb)
{
    if (*(int *) pa < *(int *) pb)
        return -1;
    if (*(int *) pa > *(int *) pb)
        return 1;
    return 0;
}

static void
action(const void *nodep, VISIT which, int depth)
{
    int *datap;

    switch (which) {
    case preorder:
        break;
    case postorder:
        datap = *(int **) nodep;
        printf("%6d\n", *datap);
        break;
    case endorder:
        break;
    case leaf:
        datap = *(int **) nodep;
        printf("%6d\n", *datap);
        break;
    }
}

int
main(void)
{
    int  *ptr;
    int  **val;

    srand(time(NULL));
    for (unsigned int i = 0; i < 12; i++) {
        ptr = xmalloc(sizeof(*ptr));
        *ptr = rand() & 0xff;
        val = tsearch(ptr, &root, compare);
        if (val == NULL)
            exit(EXIT_FAILURE);
        if (*val != ptr)
            free(ptr);
    }
    twalk(root, action);
```

```
        tdestroy(root, free);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

bsearch(3), *hsearch*(3), *lsearch*(3), *qsort*(3)

NAME

ttyname, ttyname_r − return name of a terminal

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

char *ttyname(int fd);
int ttyname_r(int fd, char buf[.buflen], size_t buflen);
```

DESCRIPTION

The function **ttyname**() returns a pointer to the null-terminated pathname of the terminal device that is open on the file descriptor *fd*, or NULL on error (for example, if *fd* is not connected to a terminal). The return value may point to static data, possibly overwritten by the next call. The function **ttyname_r**() stores this pathname in the buffer *buf* of length *buflen*.

RETURN VALUE

The function **ttyname**() returns a pointer to a pathname on success. On error, NULL is returned, and *errno* is set to indicate the error. The function **ttyname_r**() returns 0 on success, and an error number upon error.

ERRORS

EBADF
: Bad file descriptor.

ENODEV
: *fd* refers to a slave pseudoterminal device but the corresponding pathname could not be found (see NOTES).

ENOTTY
: *fd* does not refer to a terminal device.

ERANGE
: (**ttyname_r**()) *buflen* was too small to allow storing the pathname.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ttyname()	Thread safety	MT-Unsafe race:ttyname
ttyname_r()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, 4.2BSD.

NOTES

A process that keeps a file descriptor that refers to a *pts*(4) device open when switching to another mount namespace that uses a different */dev/ptmx* instance may still accidentally find that a device path of the same name for that file descriptor exists. However, this device path refers to a different device and thus can't be used to access the device that the file descriptor refers to. Calling **ttyname**() or **ttyname_r**() on the file descriptor in the new mount namespace will cause these functions to return NULL and set *errno* to **ENODEV**.

SEE ALSO

tty(1), *fstat*(2), *ctermid*(3), *isatty*(3), *pts*(4)

NAME

ttyslot − find the slot of the current user's terminal in some file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h> /* See NOTES */

int ttyslot(void);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ttyslot():
Since glibc 2.24:
_DEFAULT_SOURCE
From glibc 2.20 to glibc 2.23:
_DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
glibc 2.19 and earlier:
_BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION

The legacy function **ttyslot**() returns the index of the current user's entry in some file.

Now "What file?" you ask. Well, let's first look at some history.

Ancient history

There used to be a file */etc/ttys* in UNIX V6, that was read by the *init*(1) program to find out what to do with each terminal line. Each line consisted of three characters. The first character was either '0' or '1', where '0' meant "ignore". The second character denoted the terminal: '8' stood for "/dev/tty8". The third character was an argument to *getty*(8) indicating the sequence of line speeds to try ('−' was: start trying 110 baud). Thus a typical line was "18−". A hang on some line was solved by changing the '1' to a '0', signaling init, changing back again, and signaling init again.

In UNIX V7 the format was changed: here the second character was the argument to *getty*(8) indicating the sequence of line speeds to try ('0' was: cycle through 300-1200-150-110 baud; '4' was for the on-line console DECwriter) while the rest of the line contained the name of the tty. Thus a typical line was "14console".

Later systems have more elaborate syntax. System V-like systems have */etc/inittab* instead.

Ancient history (2)

On the other hand, there is the file */etc/utmp* listing the people currently logged in. It is maintained by *login*(1)It has a fixed size, and the appropriate index in the file was determined by *login*(1) using the **ttyslot**() call to find the number of the line in */etc/ttys* (counting from 1).

The semantics of ttyslot

Thus, the function **ttyslot**() returns the index of the controlling terminal of the calling process in the file */etc/ttys*, and that is (usually) the same as the index of the entry for the current user in the file */etc/utmp*. BSD still has the */etc/ttys* file, but System V-like systems do not, and hence cannot refer to it. Thus, on such systems the documentation says that **ttyslot**() returns the current user's index in the user accounting data base.

RETURN VALUE

If successful, this function returns the slot number. On error (e.g., if none of the file descriptors 0, 1, or 2 is associated with a terminal that occurs in this data base) it returns 0 on UNIX V6 and V7 and BSD-like systems, but −1 on System V-like systems.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ttyslot()	Thread safety	MT-Unsafe

VERSIONS

The utmp file is found in various places on various systems, such as */etc/utmp*, */var/adm/utmp*, */var/run/utmp*.

STANDARDS

None.

HISTORY

SUSv1; marked as LEGACY in SUSv2; removed in POSIX.1-2001. SUSv2 requires −1 on error.

The glibc2 implementation of this function reads the file **_PATH_TTYS**, defined in <*ttyent.h*> as "/etc/ttys". It returns 0 on error. Since Linux systems do not usually have "/etc/ttys", it will always return 0.

On BSD-like systems and Linux, the declaration of **ttyslot**() is provided by <*unistd.h*>. On System V-like systems, the declaration is provided by <*stdlib.h*>. Since glibc 2.24, <*stdlib.h*> also provides the declaration with the following feature test macro definitions:

```
(_XOPEN_SOURCE >= 500 ||
        (_XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED))
    && ! (_POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600)
```

Minix also has *fttyslot*(*fd*).

SEE ALSO

getttyent(3), *ttyname*(3), *utmp*(5)

NAME

tzset, tzname, timezone, daylight − initialize time conversion information

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <time.h>

void tzset(void);

extern char *tzname[2];
extern long timezone;
extern int daylight;
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tzset():
_POSIX_C_SOURCE

tzname:
_POSIX_C_SOURCE

timezone, *daylight*:
_XOPEN_SOURCE
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE

DESCRIPTION

The **tzset**() function initializes the *tzname* variable from the **TZ** environment variable. This function is automatically called by the other time conversion functions that depend on the timezone. In a System-V-like environment, it will also set the variables *timezone* (seconds West of UTC) and *daylight* (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present, or future when daylight saving time applies).

If the **TZ** variable does not appear in the environment, the system timezone is used. The system timezone is configured by copying, or linking, a file in the *tzfile*(5) format to */etc/localtime*. A timezone database of these files may be located in the system timezone directory (see the **FILES** section below).

If the **TZ** variable does appear in the environment, but its value is empty, or its value cannot be interpreted using any of the formats specified below, then Coordinated Universal Time (UTC) is used.

The value of **TZ** can be one of two formats. The first format is a string of characters that directly represent the timezone to be used:

```
std offset[dst[offset][,start[/time],end[/time]]]
```

There are no spaces in the specification. The *std* string specifies an abbreviation for the timezone and must be three or more alphabetic characters. When enclosed between the less-than (<) and greater-than (>) signs, the character set is expanded to include the plus (+) sign, the minus (−) sign, and digits. The *offset* string immediately follows *std* and specifies the time value to be added to the local time to get Coordinated Universal Time (UTC). The *offset* is positive if the local timezone is west of the Prime Meridian and negative if it is east. The hour must be between 0 and 24, and the minutes and seconds 00 and 59:

```
[+|-]hh[:mm[:ss]]
```

The *dst* string and *offset* specify the name and offset for the corresponding daylight saving timezone. If the offset is omitted, it defaults to one hour ahead of standard time.

The *start* field specifies when daylight saving time goes into effect and the *end* field specifies when the change is made back to standard time. These fields may have the following formats:

Jn This specifies the Julian day with *n* between 1 and 365. Leap days are not counted. In this format, February 29 can't be represented; February 28 is day 59, and March 1 is always day 60.

n This specifies the zero-based Julian day with *n* between 0 and 365. February 29 is counted in leap years.

M*m.w.d*
: This specifies day *d* (0 <= *d* <= 6) of week *w* (1 <= *w* <= 5) of month *m* (1 <= *m* <= 12). Week 1 is the first week in which day *d* occurs and week 5 is the last week in which day *d* occurs. Day 0 is a Sunday.

The *time* fields specify when, in the local time currently in effect, the change to the other time occurs. If omitted, the default is 02:00:00.

Here is an example for New Zealand, where the standard time (NZST) is 12 hours ahead of UTC, and daylight saving time (NZDT), 13 hours ahead of UTC, runs from the first Sunday in October to the third Sunday in March, and the changeovers happen at the default time of 02:00:00:

```
TZ="NZST-12:00:00NZDT-13:00:00,M10.1.0,M3.3.0"
```

The second format specifies that the timezone information should be read from a file:

```
:[filespec]
```

If the file specification *filespec* is omitted, or its value cannot be interpreted, then Coordinated Universal Time (UTC) is used. If *filespec* is given, it specifies another *tzfile*(5)-format file to read the timezone information from. If *filespec* does not begin with a '/', the file specification is relative to the system timezone directory. If the colon is omitted each of the above **TZ** formats will be tried.

Here's an example, once more for New Zealand:

```
TZ=":Pacific/Auckland"
```

ENVIRONMENT

TZ If this variable is set its value takes precedence over the system configured timezone.

TZDIR
: If this variable is set its value takes precedence over the system configured timezone database directory path.

FILES

/etc/localtime
: The system timezone file.

/usr/share/zoneinfo/
: The system timezone database directory.

/usr/share/zoneinfo/posixrules
: When a TZ string includes a dst timezone without anything following it, then this file is used for the start/end rules. It is in the *tzfile*(5) format. By default, the zoneinfo Makefile hard links it to the *America/New_York* tzfile.

Above are the current standard file locations, but they are configurable when glibc is compiled.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
tzset()	Thread safety	MT-Safe env locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001, SVr4, 4.3BSD.

4.3BSD had a function **char *timezone(***zone***,** *dst***)** that returned the name of the timezone corresponding to its first argument (minutes West of UTC). If the second argument was 0, the standard name was used, otherwise the daylight saving time version.

SEE ALSO

date(1), *gettimeofday*(2), *time*(2), *ctime*(3), *getenv*(3), *tzfile*(5)

NAME

ualarm − schedule signal after given number of microseconds

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

useconds_t ualarm(useconds_t *usecs***, useconds_t** *interval***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

ualarm():
Since glibc 2.12:
(_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
|| /* glibc >= 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _BSD_SOURCE
Before glibc 2.12:
_BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

The **ualarm**() function causes the signal **SIGALRM** to be sent to the invoking process after (not less than) *usecs* microseconds. The delay may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.

Unless caught or ignored, the **SIGALRM** signal will terminate the process.

If the *interval* argument is nonzero, further **SIGALRM** signals will be sent every *interval* microseconds after the first.

RETURN VALUE

This function returns the number of microseconds remaining for any alarm that was previously set, or 0 if no alarm was pending.

ERRORS

EINTR
Interrupted by a signal; see *signal*(7).

EINVAL
usecs or *interval* is not smaller than 1000000. (On systems where that is considered an error.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ualarm()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD, POSIX.1-2001. POSIX.1-2001 marks it as obsolete. Removed in POSIX.1-2008.

4.3BSD, SUSv2, and POSIX do not define any errors.

POSIX.1-2001 does not specify what happens if the *usecs* argument is 0. On Linux (and probably most other systems), the effect is to cancel any pending alarm.

The type *useconds_t* is an unsigned integer type capable of holding integers in the range [0,1000000]. On the original BSD implementation, and in glibc before glibc 2.1, the arguments to **ualarm**() were instead typed as *unsigned int*. Programs will be more portable if they never mention *useconds_t* explicitly.

The interaction of this function with other timer functions such as *alarm*(2), *sleep*(3), *nanosleep*(2), *setitimer*(2), *timer_create*(2), *timer_delete*(2), *timer_getoverrun*(2), *timer_gettime*(2), *timer_settime*(2), *usleep*(3) is unspecified.

This function is obsolete. Use *setitimer*(2) or POSIX interval timers (**timer_create**(2), etc.) instead.

SEE ALSO

alarm(2), *getitimer*(2), *nanosleep*(2), *select*(2), *setitimer*(2), *usleep*(3), *time*(7)

NAME

ulimit − get and set user limits

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <ulimit.h>

[[deprecated]] long ulimit(int *cmd***, long** *newlimit***);**

DESCRIPTION

Warning: this routine is obsolete. Use *getrlimit*(2), *setrlimit*(2), and *sysconf*(3) instead. For the shell command **ulimit**, see *bash*(1)

The **ulimit**() call will get or set some limit for the calling process. The *cmd* argument can have one of the following values.

UL_GETFSIZE
: Return the limit on the size of a file, in units of 512 bytes.

UL_SETFSIZE
: Set the limit on the size of a file.

3 (Not implemented for Linux.) Return the maximum possible address of the data segment.

4 (Implemented but no symbolic constant provided.) Return the maximum number of files that the calling process can open.

RETURN VALUE

On success, **ulimit**() returns a nonnegative value. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EPERM
: An unprivileged process tried to increase a limit.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ulimit()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

SVr4, POSIX.1-2001. POSIX.1-2008 marks it as obsolete.

SEE ALSO

bash(1), *getrlimit*(2), *setrlimit*(2), *sysconf*(3)

NAME

undocumented − undocumented library functions

SYNOPSIS

Undocumented library functions

DESCRIPTION

This man page mentions those library functions which are implemented in the standard libraries but not yet documented in man pages.

Solicitation

If you have information about these functions, please look in the source code, write a man page (using a style similar to that of the other Linux section 3 man pages), and send it to **mtk.man-pages@gmail.com** for inclusion in the next man page release.

The list

authdes_create(3), *authdes_getucred*(3), *authdes_pk_create*(3), *clntunix_create*(3), *creat64*(3), *dn_skipname*(3), *fcrypt*(3), *fp_nquery*(3), *fp_query*(3), *fp_resstat*(3), *freading*(3), *freopen64*(3), *fseeko64*(3), *ftello64*(3), *ftw64*(3), *fwscanf*(3), *get_avphys_pages*(3), *getdirentries64*(3), *getmsg*(3), *getnetname*(3), *get_phys_pages*(3), *getpublickey*(3), *getsecretkey*(3), *h_errlist*(3), *host2netname*(3), *hostalias*(3), *inet_nsap_addr*(3), *inet_nsap_ntoa*(3), *init_des*(3), *libc_nls_init*(3), *mstats*(3), *netname2host*(3), *netname2user*(3), *nlist*(3), *obstack_free*(3), *parse_printf_format*(3), *p_cdname*(3), *p_cdnname*(3), *p_class*(3), *p_fqname*(3), *p_option*(3), *p_query*(3), *printf_size*(3), *printf_size_info*(3), *p_rr*(3), *p_time*(3), *p_type*(3), *putlong*(3), *putshort*(3), *re_compile_fastmap*(3), *re_compile_pattern*(3), *register_printf_function*(3), *re_match*(3), *re_match_2*(3), *re_rx_search*(3), *re_search*(3), *re_search_2*(3), *re_set_registers*(3), *re_set_syntax*(3), *res_send_setqhook*(3), *res_send_setrhook*(3), *ruserpass*(3), *setfileno*(3), *sethostfile*(3), *svc_exit*(3), *svcudp_enablecache*(3), *tell*(3), *thrd_create*(3), *thrd_current*(3), *thrd_equal*(3), *thrd_sleep*(3), *thrd_yield*(3), *tr_break*(3), *tzsetwall*(3), *ufc_dofinalperm*(3), *ufc_doit*(3), *user2netname*(3), *wcschrnul*(3), *wcsftime*(3), *wscanf*(3), *xdr_authdes_cred*(3), *xdr_authdes_verf*(3), *xdr_cryptkeyarg*(3), *xdr_cryptkeyres*(3), *xdr_datum*(3), *xdr_des_block*(3), *xdr_domainname*(3), *xdr_getcredres*(3), *xdr_keybuf*(3), *xdr_keystatus*(3), *xdr_mapname*(3), *xdr_netnamestr*(3), *xdr_netobj*(3), *xdr_passwd*(3), *xdr_peername*(3), *xdr_rmtcall_args*(3), *xdr_rmtcallres*(3), *xdr_unixcred*(3), *xdr_yp_buf*(3), *xdr_yp_inaddr*(3), *xdr_ypbind_binding*(3), *xdr_ypbind_resp*(3), *xdr_ypbind_resptype*(3), *xdr_ypbind_setdom*(3), *xdr_ypdelete_args*(3), *xdr_ypmaplist*(3), *xdr_ypmaplist_str*(3), *xdr_yppasswd*(3), *xdr_ypreq_key*(3), *xdr_ypreq_nokey*(3), *xdr_ypresp_all*(3), *xdr_ypresp_all_seq*(3), *xdr_ypresp_key_val*(3), *xdr_ypresp_maplist*(3), *xdr_ypresp_master*(3), *xdr_ypresp_order*(3), *xdr_ypresp_val*(3), *xdr_ypstat*(3), *xdr_ypupdate_args*(3), *yp_all*(3), *yp_bind*(3), *yperr_string*(3), *yp_first*(3), *yp_get_default_domain*(3), *yp_maplist*(3), *yp_master*(3), *yp_match*(3), *yp_next*(3), *yp_order*(3), *ypprot_err*(3), *yp_unbind*(3), *yp_update*(3)

NAME

ungetwc − push back a wide character onto a FILE stream

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wint_t ungetwc(wint_t *wc***, FILE ***stream***);**

DESCRIPTION

The **ungetwc**() function is the wide-character equivalent of the *ungetc*(3) function. It pushes back a wide character onto *stream* and returns it.

If *wc* is **WEOF**, it returns **WEOF**. If *wc* is an invalid wide character, it sets *errno* to **EILSEQ** and returns **WEOF**.

If *wc* is a valid wide character, it is pushed back onto the stream and thus becomes available for future wide-character read operations. The file-position indicator is decremented by one or more. The end-of-file indicator is cleared. The backing storage of the file is not affected.

Note: *wc* need not be the last wide-character read from the stream; it can be any other valid wide character.

If the implementation supports multiple push-back operations in a row, the pushed-back wide characters will be read in reverse order; however, only one level of push-back is guaranteed.

RETURN VALUE

The **ungetwc**() function returns *wc* when successful, or **WEOF** upon failure.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
ungetwc()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **ungetwc**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

fgetwc(3)

NAME

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked − nonlocking stdio functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);

void clearerr_unlocked(FILE *stream);
int feof_unlocked(FILE *stream);
int ferror_unlocked(FILE *stream);
int fileno_unlocked(FILE *stream);
int fflush_unlocked(FILE *_Nullable stream);

int fgetc_unlocked(FILE *stream);
int fputc_unlocked(int c, FILE *stream);

size_t fread_unlocked(void ptr[restrict .size * .n],
                      size_t size, size_t n,
                      FILE *restrict stream);
size_t fwrite_unlocked(const void ptr[restrict .size * .n],
                       size_t size, size_t n,
                       FILE *restrict stream);

char *fgets_unlocked(char s[restrict .n], int n, FILE *restrict stream);
int fputs_unlocked(const char *restrict s, FILE *restrict stream);

#include <wchar.h>

wint_t getwc_unlocked(FILE *stream);
wint_t getwchar_unlocked(void);
wint_t fgetwc_unlocked(FILE *stream);

wint_t fputwc_unlocked(wchar_t wc, FILE *stream);
wint_t putwc_unlocked(wchar_t wc, FILE *stream);
wint_t putwchar_unlocked(wchar_t wc);

wchar_t *fgetws_unlocked(wchar_t ws[restrict .n], int n,
                         FILE *restrict stream);
int fputws_unlocked(const wchar_t *restrict ws,
                    FILE *restrict stream);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

getc_unlocked(), **getchar_unlocked**(), **putc_unlocked**(), **putchar_unlocked**():

```
/* glibc >= 2.24: */ _POSIX_C_SOURCE >= 199309L
   || /* glibc <= 2.23: */ _POSIX_C_SOURCE
   || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

clearerr_unlocked(), **feof_unlocked**(), **ferror_unlocked**(), **fileno_unlocked**(), **fflush_unlocked**(), **fgetc_unlocked**(), **fputc_unlocked**(), **fread_unlocked**(), **fwrite_unlocked**():

```
/* glibc >= 2.19: */ _DEFAULT_SOURCE
   || /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
```

fgets_unlocked(), **fputs_unlocked**(), **getwc_unlocked**(), **getwchar_unlocked**(), **fgetwc_unlocked**(), **fputwc_unlocked**(), **putwchar_unlocked**(), **fgetws_unlocked**(), **fputws_unlocked**():

```
_GNU_SOURCE
```

DESCRIPTION

Each of these functions has the same behavior as its counterpart without the "_unlocked" suffix, except that they do not use locking (they do not set locks themselves, and do not test for the presence of locks set by others) and hence are thread-unsafe. See *flockfile*(3).

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
getc_unlocked(), **putc_unlocked**(), **clearerr_unlocked**(), **fflush_unlocked**(), **fgetc_unlocked**(), **fputc_unlocked**(), **fread_unlocked**(), **fwrite_unlocked**(), **fgets_unlocked**(), **fputs_unlocked**(), **getwc_unlocked**(), **fgetwc_unlocked**(), **fputwc_unlocked**(), **putwc_unlocked**(), **fgetws_unlocked**(), **fputws_unlocked**()	Thread safety	MT-Safe race:stream
getchar_unlocked(), **getwchar_unlocked**()	Thread safety	MT-Unsafe race:stdin
putchar_unlocked(), **putwchar_unlocked**()	Thread safety	MT-Unsafe race:stdout
feof_unlocked(), **ferror_unlocked**(), **fileno_unlocked**()	Thread safety	MT-Safe

STANDARDS

getc_unlocked()
getchar_unlocked()
putc_unlocked()
putchar_unlocked()
POSIX.1-2008.

Others: None.

HISTORY

getc_unlocked()
getchar_unlocked()
putc_unlocked()
putchar_unlocked()
POSIX.1-2001.

SEE ALSO

flockfile(3), *stdio*(3)

NAME

unlockpt − unlock a pseudoterminal master/slave pair

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE
#include <stdlib.h>

int unlockpt(int fd);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

unlockpt():
Since glibc 2.24:
_XOPEN_SOURCE >= 500
glibc 2.23 and earlier:
_XOPEN_SOURCE

DESCRIPTION

The **unlockpt**() function unlocks the slave pseudoterminal device corresponding to the master pseudoterminal referred to by the file descriptor *fd*.

unlockpt() should be called before opening the slave side of a pseudoterminal.

RETURN VALUE

When successful, **unlockpt**() returns 0. Otherwise, it returns −1 and sets *errno* to indicate the error.

ERRORS

EBADF
The *fd* argument is not a file descriptor open for writing.

EINVAL
The *fd* argument is not associated with a master pseudoterminal.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
unlockpt()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1. POSIX.1-2001.

SEE ALSO

grantpt(3), *posix_openpt*(3), *ptsname*(3), *pts*(4), *pty*(7)

NAME

updwtmp, logwtmp − append an entry to the wtmp file

LIBRARY

System utilities library (*libutil*, *−lutil*)

SYNOPSIS

#include <utmp.h>

void updwtmp(const char ****wtmp_file*, const struct utmp ****ut***);**
void logwtmp(const char ****line*, const char ****name***, const char ****host***);**

DESCRIPTION

updwtmp() appends the utmp structure *ut* to the wtmp file.

logwtmp() constructs a utmp structure using *line*, *name*, *host*, current time, and current process ID. Then it calls **updwtmp**() to append the structure to the wtmp file.

FILES

/var/log/wtmp
database of past user logins

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
updwtmp(), **logwtmp**()	Thread safety	MT-Unsafe sig:ALRM timer

VERSIONS

For consistency with the other "utmpx" functions (see *getutxent*(3)), glibc provides (since glibc 2.1):

```
#define _GNU_SOURCE          /* See feature_test_macros(7) */
#include <utmpx.h>
void updwtmpx (const char *wtmpx_file, const struct utmpx *utx);
```

This function performs the same task as **updwtmp**(), but differs in that it takes a *utmpx* structure as its last argument.

STANDARDS

None.

HISTORY

Solaris, NetBSD.

SEE ALSO

getutxent(3), *wtmp*(5)

NAME

uselocale − set/get the locale for the calling thread

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <locale.h>

locale_t uselocale(locale_t *newloc***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

uselocale():
Since glibc 2.10:
_XOPEN_SOURCE >= 700
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **uselocale**() function sets the current locale for the calling thread, and returns the thread's previously current locale. After a successful call to **uselocale**(), any calls by this thread to functions that depend on the locale will operate as though the locale has been set to *newloc*.

The *newloc* argument can have one of the following values:

A handle returned by a call to **newlocale**(3) or **duplocale**(3)
: The calling thread's current locale is set to the specified locale.

The special locale object handle **LC_GLOBAL_LOCALE**
: The calling thread's current locale is set to the global locale determined by *setlocale*(3).

(locale_t) 0
: The calling thread's current locale is left unchanged (and the current locale is returned as the function result).

RETURN VALUE

On success, **uselocale**() returns the locale handle that was set by the previous call to **uselocale**() in this thread, or **LC_GLOBAL_LOCALE** if there was no such previous call. On error, it returns *(locale_t) 0*, and sets *errno* to indicate the error.

ERRORS

EINVAL
: *newloc* does not refer to a valid locale object.

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.3. POSIX.1-2008.

NOTES

Unlike *setlocale*(3), **uselocale**() does not allow selective replacement of individual locale categories. To employ a locale that differs in only a few categories from the current locale, use calls to *duplocale*(3) and *newlocale*(3) to obtain a locale object equivalent to the current locale and modify the desired categories in that object.

EXAMPLES

See *newlocale*(3) and *duplocale*(3).

SEE ALSO

locale(1), *duplocale*(3), *freelocale*(3), *newlocale*(3), *setlocale*(3), *locale*(5), *locale*(7)

NAME

usleep − suspend execution for microsecond intervals

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <unistd.h>

int usleep(useconds_t usec);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

usleep():
  Since glibc 2.12:
    (_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
      || /* glibc >= 2.19: */ _DEFAULT_SOURCE
      || /* glibc <= 2.19: */ _BSD_SOURCE
  Before glibc 2.12:
    _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

The **usleep**() function suspends execution of the calling thread for (at least) *usec* microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.

RETURN VALUE

The **usleep**() function returns 0 on success. On error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EINTR
: Interrupted by a signal; see *signal*(7).

EINVAL
: *usec* is greater than or equal to 1000000. (On systems where that is considered an error.)

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
usleep()	Thread safety	MT-Safe

STANDARDS

None.

HISTORY

4.3BSD, POSIX.1-2001. POSIX.1-2001 declares it obsolete, suggesting *nanosleep*(2) instead. Removed in POSIX.1-2008.

On the original BSD implementation, and before glibc 2.2.2, the return type of this function is *void*. The POSIX version returns *int*, and this is also the prototype used since glibc 2.2.2.

Only the **EINVAL** error return is documented by SUSv2 and POSIX.1-2001.

CAVEATS

The interaction of this function with the **SIGALRM** signal, and with other timer functions such as *alarm*(2), *sleep*(3), *nanosleep*(2), *setitimer*(2), *timer_create*(2), *timer_delete*(2), *timer_getoverrun*(2), *timer_gettime*(2), *timer_settime*(2), *ualarm*(3) is unspecified.

SEE ALSO

alarm(2), *getitimer*(2), *nanosleep*(2), *select*(2), *setitimer*(2), *sleep*(3), *ualarm*(3), *useconds_t*(3type), *time*(7)

NAME

wcpcpy − copy a wide-character string, returning a pointer to its end

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcpcpy(wchar_t *restrict *dest***, const wchar_t *restrict** *src***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcpcpy():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcpcpy**() function is the wide-character equivalent of the *stpcpy*(3) function. It copies the wide-character string pointed to by *src*, including the terminating null wide character (L'\0'), to the array pointed to by *dest*.

The strings may not overlap.

The programmer must ensure that there is room for at least *wcslen(src)+1* wide characters at *dest*.

RETURN VALUE

wcpcpy() returns a pointer to the end of the wide-character string *dest*, that is, a pointer to the terminating null wide character.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcpcpy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

SEE ALSO

strcpy(3), *wcscpy*(3)

NAME

wcpncpy − copy a fixed-size string of wide characters, returning a pointer to its end

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcpncpy(wchar_t dest[restrict .n],
                 const wchar_t src[restrict .n],
                 size_t n);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcpncpy():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcpncpy**() function is the wide-character equivalent of the *stpncpy*(3) function. It copies at most *n* wide characters from the wide-character string pointed to by *src*, including the terminating null wide (L'\0'), to the array pointed to by *dest*. Exactly *n* wide characters are written at *dest*. If the length *wcslen(src)* is smaller than *n*, the remaining wide characters in the array pointed to by *dest* are filled with L'\0' characters. If the length *wcslen(src)* is greater than or equal to *n*, the string pointed to by *dest* will not be L'\0' terminated.

The strings may not overlap.

The programmer must ensure that there is room for at least *n* wide characters at *dest*.

RETURN VALUE

wcpncpy() returns a pointer to the last wide character written, that is, *dest*+*n*−1.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcpncpy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

SEE ALSO

stpncpy(3), *wcsncpy*(3)

NAME

wcrtomb − convert a wide character to a multibyte sequence

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

size_t wcrtomb(char *restrict *s***, wchar_t** *wc***, mbstate_t *restrict** *ps***);**

DESCRIPTION

The main case for this function is when *s* is not NULL and *wc* is not a null wide character (L'\0'). In this case, the **wcrtomb**() function converts the wide character *wc* to its multibyte representation and stores it at the beginning of the character array pointed to by *s*. It updates the shift state **ps*, and returns the length of said multibyte representation, that is, the number of bytes written at *s*.

A different case is when *s* is not NULL, but *wc* is a null wide character (L'\0'). In this case, the **wcrtomb**() function stores at the character array pointed to by *s* the shift sequence needed to bring **ps* back to the initial state, followed by a '\0' byte. It updates the shift state **ps* (i.e., brings it into the initial state), and returns the length of the shift sequence plus one, that is, the number of bytes written at *s*.

A third case is when *s* is NULL. In this case, *wc* is ignored, and the function effectively returns

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal anonymous buffer.

In all of the above cases, if *ps* is NULL, a static anonymous state known only to the **wcrtomb**() function is used instead.

RETURN VALUE

The **wcrtomb**() function returns the number of bytes that have been or would have been written to the byte array at *s*. If *wc* can not be represented as a multibyte sequence (according to the current locale), *(size_t) −1* is returned, and *errno* set to **EILSEQ**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcrtomb()	Thread safety	MT-Unsafe race:wcrtomb/!ps

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wcrtomb**() depends on the **LC_CTYPE** category of the current locale.

Passing NULL as *ps* is not multithread safe.

SEE ALSO

mbsinit(3), *wcsrtombs*(3)

NAME

wcscasecmp − compare two wide-character strings, ignoring case

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

int wcscasecmp(const wchar_t *s1, const wchar_t *s2);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcscasecmp():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcscasecmp**() function is the wide-character equivalent of the *strcasecmp*(3) function. It compares the wide-character string pointed to by *s1* and the wide-character string pointed to by *s2*, ignoring case differences (**towupper**(3), *towlower*(3)).

RETURN VALUE

The **wcscasecmp**() function returns zero if the wide-character strings at *s1* and *s2* are equal except for case distinctions. It returns a positive integer if *s1* is greater than *s2*, ignoring case. It returns a negative integer if *s1* is smaller than *s2*, ignoring case.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcscasecmp()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1.

NOTES

The behavior of **wcscasecmp**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

strcasecmp(3), *wcscmp*(3)

NAME

wcscat − concatenate two wide-character strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict *dest***, const wchar_t *restrict** *src***);**

DESCRIPTION

The **wcscat**() function is the wide-character equivalent of the *strcat*(3) function. It copies the wide-character string pointed to by *src*, including the terminating null wide character (L'\0'), to the end of the wide-character string pointed to by *dest*.

The strings may not overlap.

The programmer must ensure that there is room for at least *wcslen(dest)*+*wcslen(src)*+1 wide characters at *dest*.

RETURN VALUE

wcscat() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcscat()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strcat(3), *wcpcpy*(3), *wcscpy*(3), *wcsncat*(3)

NAME

wcschr − search a wide character in a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcschr(const wchar_t *_wcs_, wchar_t _wc_);

DESCRIPTION

The **wcschr**() function is the wide-character equivalent of the *strchr*(3) function. It searches the first occurrence of *wc* in the wide-character string pointed to by *wcs*.

RETURN VALUE

The **wcschr**() function returns a pointer to the first occurrence of *wc* in the wide-character string pointed to by *wcs*, or NULL if *wc* does not occur in the string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcschr()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strchr(3), *wcspbrk*(3), *wcsrchr*(3), *wcsstr*(3), *wmemchr*(3)

NAME

wcscmp − compare two wide-character strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int wcscmp(const wchar_t *_s1_, const wchar_t *_s2_);

DESCRIPTION

The **wcscmp**() function is the wide-character equivalent of the *strcmp*(3) function. It compares the wide-character string pointed to by *s1* and the wide-character string pointed to by *s2*.

RETURN VALUE

The **wcscmp**() function returns zero if the wide-character strings at *s1* and *s2* are equal. It returns an integer greater than zero if at the first differing position *i*, the corresponding wide-character *s1[i]* is greater than *s2[i]*. It returns an integer less than zero if at the first differing position *i*, the corresponding wide-character *s1[i]* is less than *s2[i]*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcscmp()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strcmp(3), *wcscasecmp*(3), *wmemcmp*(3)

NAME

wcscpy − copy a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict *dest***, const wchar_t *restrict** *src***);**

DESCRIPTION

The **wcscpy**() function is the wide-character equivalent of the *strcpy*(3) function. It copies the wide-character string pointed to by *src*, including the terminating null wide character (L'\0'), to the array pointed to by *dest*.

The strings may not overlap.

The programmer must ensure that there is room for at least *wcslen(src)+1* wide characters at *dest*.

RETURN VALUE

wcscpy() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcscpy()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strcpy(3), *wcpcpy*(3), *wcscat*(3), *wcsdup*(3), *wmemcpy*(3)

NAME

wcscspn − search a wide-character string for any of a set of wide characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

size_t wcscspn(const wchar_t **wcs*, const wchar_t **reject*);

DESCRIPTION

The **wcscspn**() function is the wide-character equivalent of the *strcspn*(3) function. It determines the length of the longest initial segment of *wcs* which consists entirely of wide-characters not listed in *reject*. In other words, it searches for the first occurrence in the wide-character string *wcs* of any of the characters in the wide-character string *reject*.

RETURN VALUE

The **wcscspn**() function returns the number of wide characters in the longest initial segment of *wcs* which consists entirely of wide-characters not listed in *reject*. In other words, it returns the position of the first occurrence in the wide-character string *wcs* of any of the characters in the wide-character string *reject*, or *wcslen(wcs)* if there is none.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcscspn()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strcspn(3), *wcspbrk*(3), *wcsspn*(3)

NAME

wcsdup − duplicate a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcsdup(const wchar_t *s);

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcsdup():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcsdup**() function is the wide-character equivalent of the *strdup*(3) function. It allocates and returns a new wide-character string whose initial contents is a duplicate of the wide-character string pointed to by *s*.

Memory for the new wide-character string is obtained with *malloc*(3), and should be freed with *free*(3).

RETURN VALUE

On success, **wcsdup**() returns a pointer to the new wide-character string. On error, it returns NULL, with *errno* set to indicate the error.

ERRORS

ENOMEM
Insufficient memory available to allocate duplicate string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsdup()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

libc5, glibc 2.0.

SEE ALSO

strdup(3), *wcscpy*(3)

NAME

wcslen − determine the length of a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

size_t wcslen(const wchar_t *s);

DESCRIPTION

The **wcslen**() function is the wide-character equivalent of the *strlen*(3) function. It determines the length of the wide-character string pointed to by *s*, excluding the terminating null wide character (L'\0').

RETURN VALUE

The **wcslen**() function returns the number of wide characters in *s*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcslen()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

In cases where the input buffer may not contain a terminating null wide character, *wcsnlen*(3) should be used instead.

SEE ALSO

strlen(3)

NAME

wcsncasecmp − compare two fixed-size wide-character strings, ignoring case

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int wcsncasecmp(const wchar_t *s1***[.***n***], const wchar_t** *s2***[.***n***], size_t** *n***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcsncasecmp():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcsncasecmp**() function is the wide-character equivalent of the *strncasecmp*(3) function. It compares the wide-character string pointed to by *s1* and the wide-character string pointed to by *s2*, but at most *n* wide characters from each string, ignoring case differences (**towupper**(3), *towlower*(3)).

RETURN VALUE

The **wcsncasecmp**() function returns zero if the wide-character strings at *s1* and *s2*, truncated to at most length *n*, are equal except for case distinctions. It returns a positive integer if truncated *s1* is greater than truncated *s2*, ignoring case. It returns a negative integer if truncated *s1* is smaller than truncated *s2*, ignoring case.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsncasecmp()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1.

NOTES

The behavior of **wcsncasecmp**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

strncasecmp(3), *wcsncmp*(3)

NAME

wcsncat − concatenate two wide-character strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t dest[restrict .n],
                 const wchar_t src[restrict .n],
                 size_t n);
```

DESCRIPTION

The **wcsncat**() function is the wide-character equivalent of the *strncat*(3) function. It copies at most *n* wide characters from the wide-character string pointed to by *src* to the end of the wide-character string pointed to by *dest*, and adds a terminating null wide character (L'\0').

The strings may not overlap.

The programmer must ensure that there is room for at least *wcslen(dest)*+*n*+1 wide characters at *dest*.

RETURN VALUE

wcsncat() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsncat()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strncat(3), *wcscat*(3)

NAME

wcsncmp − compare two fixed-size wide-character strings

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int wcsncmp(const wchar_t *s1***[.***n***], const wchar_t** *s2***[.***n***], size_t** *n***);**

DESCRIPTION

The **wcsncmp**() function is the wide-character equivalent of the *strncmp*(3) function. It compares the wide-character string pointed to by *s1* and the wide-character string pointed to by *s2*, but at most *n* wide characters from each string. In each string, the comparison extends only up to the first occurrence of a null wide character (L'\0'), if any.

RETURN VALUE

The **wcsncmp**() function returns zero if the wide-character strings at *s1* and *s2*, truncated to at most length *n*, are equal. It returns an integer greater than zero if at the first differing position *i* (*i* < *n*), the corresponding wide-character *s1[i]* is greater than *s2[i]*. It returns an integer less than zero if at the first differing position *i* (*i* < *n*), the corresponding wide-character *s1[i]* is less than *s2[i]*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsncmp()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strncmp(3), *wcsncasecmp*(3)

NAME

wcsncpy − copy a fixed-size string of wide characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t dest[restrict .n],
                 const wchar_t src[restrict .n],
                 size_t n);
```

DESCRIPTION

The **wcsncpy**() function is the wide-character equivalent of the *strncpy*(3) function. It copies at most *n* wide characters from the wide-character string pointed to by *src*, including the terminating null wide character (L'\0'), to the array pointed to by *dest*. Exactly *n* wide characters are written at *dest*. If the length *wcslen(src)* is smaller than *n*, the remaining wide characters in the array pointed to by *dest* are filled with null wide characters. If the length *wcslen(src)* is greater than or equal to *n*, the string pointed to by *dest* will not be terminated by a null wide character.

The strings may not overlap.

The programmer must ensure that there is room for at least *n* wide characters at *dest*.

RETURN VALUE

wcsncpy() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsncpy()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strncpy(3)

NAME

wcsnlen − determine the length of a fixed-size wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

size_t wcsnlen(const wchar_t *s***[.***maxlen***], size_t** *maxlen***);**

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcsnlen():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcsnlen**() function is the wide-character equivalent of the *strnlen*(3) function. It returns the number of wide-characters in the string pointed to by *s*, not including the terminating null wide character (L'\0'), but at most *maxlen* wide characters (note: this parameter is not a byte count). In doing this, **wcsnlen**() looks at only the first *maxlen* wide characters at *s* and never beyond *s[maxlen−1]*.

RETURN VALUE

The **wcsnlen**() function returns *wcslen(s)*, if that is less than *maxlen*, or *maxlen* if there is no null wide character among the first *maxlen* wide characters pointed to by *s*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsnlen()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

glibc 2.1.

SEE ALSO

strnlen(3), *wcslen*(3)

NAME

wcsnrtombs − convert a wide-character string to a multibyte string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t wcsnrtombs(char dest[restrict .len], const wchar_t **restrict src,
                  size_t nwc, size_t len, mbstate_t *restrict ps);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wcsnrtombs():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The **wcsnrtombs**() function is like the *wcsrtombs*(3) function, except that the number of wide characters to be converted, starting at **src*, is limited to *nwc*.

If *dest* is not NULL, the **wcsnrtombs**() function converts at most *nwc* wide characters from the wide-character string **src* to a multibyte string starting at *dest*. At most *len* bytes are written to *dest*. The shift state **ps* is updated. The conversion is effectively performed by repeatedly calling *wcrtomb(dest, *src, ps)*, as long as this call succeeds, and then incrementing *dest* by the number of bytes written and **src* by one. The conversion can stop for three reasons:

- A wide character has been encountered that can not be represented as a multibyte sequence (according to the current locale). In this case, **src* is left pointing to the invalid wide character, *(size_t) −1* is returned, and *errno* is set to **EILSEQ**.
- *nwc* wide characters have been converted without encountering a null wide character (L'\0'), or the length limit forces a stop. In this case, **src* is left pointing to the next wide character to be converted, and the number of bytes written to *dest* is returned.
- The wide-character string has been completely converted, including the terminating null wide character (which has the side effect of bringing back **ps* to the initial state). In this case, **src* is set to NULL, and the number of bytes written to *dest*, excluding the terminating null byte ('\0'), is returned.

If *dest* is NULL, *len* is ignored, and the conversion proceeds as above, except that the converted bytes are not written out to memory, and that no destination length limit exists.

In both of the above cases, if *ps* is NULL, a static anonymous state known only to the **wcsnrtombs**() function is used instead.

The programmer must ensure that there is room for at least *len* bytes at *dest*.

RETURN VALUE

The **wcsnrtombs**() function returns the number of bytes that make up the converted part of multibyte sequence, not including the terminating null byte. If a wide character was encountered which could not be converted, *(size_t) −1* is returned, and *errno* set to **EILSEQ**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsnrtombs()	Thread safety	MT-Unsafe race:wcsnrtombs/!ps

STANDARDS

POSIX.1-2008.

NOTES

The behavior of **wcsnrtombs**() depends on the **LC_CTYPE** category of the current locale.

Passing NULL as *ps* is not multithread safe.

SEE ALSO

iconv(3), *mbsinit*(3), *wcsrtombs*(3)

NAME

wcspbrk − search a wide-character string for any of a set of wide characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *_wcs_, const wchar_t *_accept_);

DESCRIPTION

The **wcspbrk**() function is the wide-character equivalent of the *strpbrk*(3) function. It searches for the first occurrence in the wide-character string pointed to by *wcs* of any of the characters in the wide-character string pointed to by *accept*.

RETURN VALUE

The **wcspbrk**() function returns a pointer to the first occurrence in *wcs* of any of the characters listed in *accept*. If *wcs* contains none of these characters, NULL is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcspbrk()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strpbrk(3), *wcschr*(3), *wcscspn*(3)

NAME

wcsrchr − search a wide character in a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *_wcs_, wchar_t _wc_);

DESCRIPTION

The **wcsrchr**() function is the wide-character equivalent of the *strrchr*(3) function. It searches the last occurrence of *wc* in the wide-character string pointed to by *wcs*.

RETURN VALUE

The **wcsrchr**() function returns a pointer to the last occurrence of *wc* in the wide-character string pointed to by *wcs*, or NULL if *wc* does not occur in the string.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsrchr()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strrchr(3), *wcschr*(3)

NAME

wcsrtombs − convert a wide-character string to a multibyte string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

size_t wcsrtombs(char dest[restrict .len], const wchar_t **restrict src,
                 size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

If *dest* is not NULL, the **wcsrtombs**() function converts the wide-character string **src* to a multibyte string starting at *dest*. At most *len* bytes are written to *dest*. The shift state **ps* is updated. The conversion is effectively performed by repeatedly calling *wcrtomb(dest, *src, ps)*, as long as this call succeeds, and then incrementing *dest* by the number of bytes written and **src* by one. The conversion can stop for three reasons:

- A wide character has been encountered that can not be represented as a multibyte sequence (according to the current locale). In this case, **src* is left pointing to the invalid wide character, *(size_t) −1* is returned, and *errno* is set to **EILSEQ**.
- The length limit forces a stop. In this case, **src* is left pointing to the next wide character to be converted, and the number of bytes written to *dest* is returned.
- The wide-character string has been completely converted, including the terminating null wide character (L'\0'), which has the side effect of bringing back **ps* to the initial state. In this case, **src* is set to NULL, and the number of bytes written to *dest*, excluding the terminating null byte ('\0'), is returned.

If *dest* is NULL, *len* is ignored, and the conversion proceeds as above, except that the converted bytes are not written out to memory, and that no length limit exists.

In both of the above cases, if *ps* is NULL, a static anonymous state known only to the **wcsrtombs**() function is used instead.

The programmer must ensure that there is room for at least *len* bytes at *dest*.

RETURN VALUE

The **wcsrtombs**() function returns the number of bytes that make up the converted part of multibyte sequence, not including the terminating null byte. If a wide character was encountered which could not be converted, *(size_t) −1* is returned, and *errno* set to **EILSEQ**.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsrtombs()	Thread safety	MT-Unsafe race:wcsrtombs/!ps

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wcsrtombs**() depends on the **LC_CTYPE** category of the current locale.

Passing NULL as *ps* is not multithread safe.

SEE ALSO

iconv(3), *mbsinit*(3), *wcrtomb*(3), *wcsnrtombs*(3), *wcstombs*(3)

NAME

wcsspn − get length of a prefix wide-character substring

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

size_t wcsspn(const wchar_t **wcs*, const wchar_t **accept*);

DESCRIPTION

The **wcsspn**() function is the wide-character equivalent of the *strspn*(3) function. It determines the length of the longest initial segment of *wcs* which consists entirely of wide-characters listed in *accept*. In other words, it searches for the first occurrence in the wide-character string *wcs* of a wide-character not contained in the wide-character string *accept*.

RETURN VALUE

The **wcsspn**() function returns the number of wide characters in the longest initial segment of *wcs* which consists entirely of wide-characters listed in *accept*. In other words, it returns the position of the first occurrence in the wide-character string *wcs* of a wide-character not contained in the wide-character string *accept*, or *wcslen(wcs)* if there is none.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsspn()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strspn(3), *wcscspn*(3)

NAME

wcsstr − locate a substring in a wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wcsstr(const wchar_t ****haystack*, const wchar_t ****needle***);**

DESCRIPTION

The **wcsstr**() function is the wide-character equivalent of the *strstr*(3) function. It searches for the first occurrence of the wide-character string *needle* (without its terminating null wide character (L'\0')) as a substring in the wide-character string *haystack*.

RETURN VALUE

The **wcsstr**() function returns a pointer to the first occurrence of *needle* in *haystack*. It returns NULL if *needle* does not occur as a substring in *haystack*.

Note the special case: If *needle* is the empty wide-character string, the return value is always *haystack* itself.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcsstr()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

strstr(3), *wcschr*(3)

NAME

wcstoimax, wcstoumax − convert wide-character string to integer

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stddef.h>
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t *restrict nptr,
                   wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr,
                    wchar_t **restrict endptr, int base);
```

DESCRIPTION

These functions are just like *wcstol*(3) and *wcstoul*(3), except that they return a value of type *intmax_t* and *uintmax_t*, respectively.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcstoimax(), **wcstoumax**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

imaxabs(3), *imaxdiv*(3), *strtoimax*(3), *strtoumax*(3), *wcstol*(3), *wcstoul*(3)

NAME

wcstok − split wide-character string into tokens

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *restrict wcs, const wchar_t *restrict delim,
                wchar_t **restrict ptr);
```

DESCRIPTION

The **wcstok**() function is the wide-character equivalent of the *strtok*(3) function, with an added argument to make it multithread-safe. It can be used to split a wide-character string *wcs* into tokens, where a token is defined as a substring not containing any wide-characters from *delim*.

The search starts at *wcs*, if *wcs* is not NULL, or at **ptr*, if *wcs* is NULL. First, any delimiter wide-characters are skipped, that is, the pointer is advanced beyond any wide-characters which occur in *delim*. If the end of the wide-character string is now reached, **wcstok**() returns NULL, to indicate that no tokens were found, and stores an appropriate value in **ptr*, so that subsequent calls to **wcstok**() will continue to return NULL. Otherwise, the **wcstok**() function recognizes the beginning of a token and returns a pointer to it, but before doing that, it zero-terminates the token by replacing the next wide-character which occurs in *delim* with a null wide character (L'\0'), and it updates **ptr* so that subsequent calls will continue searching after the end of recognized token.

RETURN VALUE

The **wcstok**() function returns a pointer to the next token, or NULL if no further token was found.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcstok()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The original *wcs* wide-character string is destructively modified during the operation.

EXAMPLES

The following code loops over the tokens contained in a wide-character string.

```
wchar_t *wcs = ...;
wchar_t *token;
wchar_t *state;
for (token = wcstok(wcs, L" \t\n", &state);
    token != NULL;
    token = wcstok(NULL, L" \t\n", &state)) {
    ...
}
```

SEE ALSO

strtok(3), *wcschr*(3)

NAME

wcstombs − convert a wide-character string to a multibyte string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdlib.h>

size_t wcstombs(char dest[restrict .n], const wchar_t *restrict src,
                size_t n);
```

DESCRIPTION

If *dest* is not NULL, the **wcstombs**() function converts the wide-character string *src* to a multibyte string starting at *dest*. At most *n* bytes are written to *dest*. The sequence of characters placed in *dest* begins in the initial shift state. The conversion can stop for three reasons:

- A wide character has been encountered that can not be represented as a multibyte sequence (according to the current locale). In this case, *(size_t) −1* is returned.
- The length limit forces a stop. In this case, the number of bytes written to *dest* is returned, but the shift state at this point is lost.
- The wide-character string has been completely converted, including the terminating null wide character (L'\0'). In this case, the conversion ends in the initial shift state. The number of bytes written to *dest*, excluding the terminating null byte ('\0'), is returned.

The programmer must ensure that there is room for at least *n* bytes at *dest*.

If *dest* is NULL, *n* is ignored, and the conversion proceeds as above, except that the converted bytes are not written out to memory, and no length limit exists.

In order to avoid the case 2 above, the programmer should make sure *n* is greater than or equal to *wcstombs(NULL,src,0)+1*.

RETURN VALUE

The **wcstombs**() function returns the number of bytes that make up the converted part of a multibyte sequence, not including the terminating null byte. If a wide character was encountered which could not be converted, *(size_t) −1* is returned.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcstombs()	Thread safety	MT-Safe

VERSIONS

The function *wcsrtombs*(3) provides a better interface to the same functionality.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wcstombs**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

mblen(3), *mbstowcs*(3), *mbtowc*(3), *wcsrtombs*(3), *wctomb*(3)

NAME

wcswidth − determine columns needed for a fixed-size wide-character string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE         /* See feature_test_macros(7) */
#include <wchar.h>

int wcswidth(const wchar_t *s, size_t n);
```

DESCRIPTION

The **wcswidth**() function returns the number of columns needed to represent the wide-character string pointed to by *s*, but at most *n* wide characters. If a nonprintable wide character occurs among these characters, −1 is returned.

RETURN VALUE

The **wcswidth**() function returns the number of column positions for the wide-character string *s*, truncated to at most length *n*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcswidth()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The behavior of **wcswidth**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iswprint(3), *wcwidth*(3)

NAME

wctob − try to represent a wide character as a single byte

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int wctob(wint_t *c***);**

DESCRIPTION

The **wctob**() function tests whether the multibyte representation of the wide character *c*, starting in the initial state, consists of a single byte. If so, it is returned as an *unsigned char*.

Never use this function. It cannot help you in writing internationalized programs. Internationalized programs must never distinguish single-byte and multibyte characters.

RETURN VALUE

The **wctob**() function returns the single-byte representation of *c*, if it exists, or **EOF** otherwise.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wctob()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wctob**() depends on the **LC_CTYPE** category of the current locale.

This function should never be used. Internationalized programs must never distinguish single-byte and multibyte characters. Use either *wctomb*(3) or the thread-safe *wcrtomb*(3) instead.

SEE ALSO

btowc(3), *wcrtomb*(3), *wctomb*(3)

NAME

wctomb − convert a wide character to a multibyte sequence

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <stdlib.h>

int wctomb(char ****s*, wchar_t** *wc***);**

DESCRIPTION

If *s* is not NULL, the **wctomb**() function converts the wide character *wc* to its multibyte representation and stores it at the beginning of the character array pointed to by *s*. It updates the shift state, which is stored in a static anonymous variable known only to the **wctomb**() function, and returns the length of said multibyte representation, that is, the number of bytes written at *s*.

The programmer must ensure that there is room for at least **MB_CUR_MAX** bytes at *s*.

If *s* is NULL, the **wctomb**() function resets the shift state, known only to this function, to the initial state, and returns nonzero if the encoding has nontrivial shift state, or zero if the encoding is stateless.

RETURN VALUE

If *s* is not NULL, the **wctomb**() function returns the number of bytes that have been written to the byte array at *s*. If *wc* can not be represented as a multibyte sequence (according to the current locale), −1 is returned.

If *s* is NULL, the **wctomb**() function returns nonzero if the encoding has nontrivial shift state, or zero if the encoding is stateless.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wctomb()	Thread safety	MT-Unsafe race

VERSIONS

The function *wcrtomb*(3) provides a better interface to the same functionality.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wctomb**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

MB_CUR_MAX(3), *mblen*(3), *mbstowcs*(3), *mbtowc*(3), *wcrtomb*(3), *wcstombs*(3)

NAME

wctrans − wide-character translation mapping

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

wctrans_t wctrans(const char *_name_);

DESCRIPTION

The *wctrans_t* type represents a mapping which can map a wide character to another wide character. Its nature is implementation-dependent, but the special value *(wctrans_t) 0* denotes an invalid mapping. Nonzero *wctrans_t* values can be passed to the *towctrans*(3) function to actually perform the wide-character mapping.

The **wctrans**() function returns a mapping, given by its name. The set of valid names depends on the **LC_CTYPE** category of the current locale, but the following names are valid in all locales.

"tolower" − realizes the **tolower**(3) mapping
"toupper" − realizes the **toupper**(3) mapping

RETURN VALUE

The **wctrans**() function returns a mapping descriptor if the *name* is valid. Otherwise, it returns *(wctrans_t) 0*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wctrans()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wctrans**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

towctrans(3)

NAME

wctype − wide-character classification

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wctype.h>

wctype_t wctype(const char *****name***);**

DESCRIPTION

The *wctype_t* type represents a property which a wide character may or may not have. In other words, it represents a class of wide characters. This type's nature is implementation-dependent, but the special value *(wctype_t) 0* denotes an invalid property. Nonzero *wctype_t* values can be passed to the *iswctype*(3) function to actually test whether a given wide character has the property.

The **wctype**() function returns a property, given by its name. The set of valid names depends on the **LC_CTYPE** category of the current locale, but the following names are valid in all locales.

"alnum" − realizes the **isalnum**(3) classification function
"alpha" − realizes the **isalpha**(3) classification function
"blank" − realizes the **isblank**(3) classification function
"cntrl" − realizes the **iscntrl**(3) classification function
"digit" − realizes the **isdigit**(3) classification function
"graph" − realizes the **isgraph**(3) classification function
"lower" − realizes the **islower**(3) classification function
"print" − realizes the **isprint**(3) classification function
"punct" − realizes the **ispunct**(3) classification function
"space" − realizes the **isspace**(3) classification function
"upper" − realizes the **isupper**(3) classification function
"xdigit" − realizes the **isxdigit**(3) classification function

RETURN VALUE

The **wctype**() function returns a property descriptor if the *name* is valid. Otherwise, it returns *(wctype_t) 0*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wctype()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wctype**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iswctype(3)

NAME

wcwidth − determine columns needed for a wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#define _XOPEN_SOURCE       /* See feature_test_macros(7) */
#include <wchar.h>

int wcwidth(wchar_t c);
```

DESCRIPTION

The **wcwidth**() function returns the number of columns needed to represent the wide character *c*. If *c* is a printable wide character, the value is at least 0. If *c* is null wide character (L'\0'), the value is 0. Otherwise, −1 is returned.

RETURN VALUE

The **wcwidth**() function returns the number of column positions for *c*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wcwidth()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Note that before glibc 2.2.5, glibc used the prototype

```
int wcwidth(wint_t c);
```

NOTES

The behavior of **wcwidth**() depends on the **LC_CTYPE** category of the current locale.

SEE ALSO

iswprint(3), *wcswidth*(3)

NAME

wmemchr − search a wide character in a wide-character array

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wmemchr(const wchar_t *s***[.***n***], wchar_t** *c***, size_t** *n***);**

DESCRIPTION

The **wmemchr**() function is the wide-character equivalent of the *memchr*(3) function. It searches the *n* wide characters starting at *s* for the first occurrence of the wide character *c*.

RETURN VALUE

The **wmemchr**() function returns a pointer to the first occurrence of *c* among the *n* wide characters starting at *s*, or NULL if *c* does not occur among these.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wmemchr()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

memchr(3), *wcschr*(3)

NAME

wmemcmp − compare two arrays of wide-characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

int wmemcmp(const wchar_t *s1***[.***n***], const wchar_t** *s2***[.***n***], size_t** *n***);**

DESCRIPTION

The **wmemcmp**() function is the wide-character equivalent of the *memcmp*(3) function. It compares the *n* wide-characters starting at *s1* and the *n* wide-characters starting at *s2*.

RETURN VALUE

The **wmemcmp**() function returns zero if the wide-character arrays of size *n* at *s1* and *s2* are equal. It returns an integer greater than zero if at the first differing position *i* ($i < n$), the corresponding wide-character *s1[i]* is greater than *s2[i]*. It returns an integer less than zero if at the first differing position *i* ($i < n$), the corresponding wide-character *s1[i]* is less than *s2[i]*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wmemcmp()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

memcmp(3), *wcscmp*(3)

NAME

wmemcpy − copy an array of wide-characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t dest[restrict .n],
                 const wchar_t src[restrict .n],
                 size_t n);
```

DESCRIPTION

The **wmemcpy**() function is the wide-character equivalent of the *memcpy*(3) function. It copies *n* wide characters from the array starting at *src* to the array starting at *dest*.

The arrays may not overlap; use *wmemmove*(3) to copy between overlapping arrays.

The programmer must ensure that there is room for at least *n* wide characters at *dest*.

RETURN VALUE

wmemcpy() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wmemcpy()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

memcpy(3), *wcscpy*(3), *wmemmove*(3), *wmempcpy*(3)

NAME

wmemmove − copy an array of wide-characters

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wmemmove(wchar_t *dest***[.***n***], const wchar_t** *src***[.***n***], size_t** *n***);**

DESCRIPTION

The **wmemmove**() function is the wide-character equivalent of the *memmove*(3) function. It copies *n* wide characters from the array starting at *src* to the array starting at *dest*. The arrays may overlap.

The programmer must ensure that there is room for at least *n* wide characters at *dest*.

RETURN VALUE

wmemmove() returns *dest*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wmemmove()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

memmove(3), *wmemcpy*(3)

NAME

wmemset − fill an array of wide-characters with a constant wide character

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <wchar.h>

wchar_t *wmemset(wchar_t *wcs***[.***n***], wchar_t** *wc***, size_t** *n***);**

DESCRIPTION

The **wmemset**() function is the wide-character equivalent of the *memset*(3) function. It fills the array of *n* wide-characters starting at *wcs* with *n* copies of the wide character *wc*.

RETURN VALUE

wmemset() returns *wcs*.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wmemset()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

SEE ALSO

memset(3)

NAME

wordexp, wordfree − perform word expansion like a posix-shell

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <wordexp.h>

int wordexp(const char *restrict s, wordexp_t *restrict p, int flags);
void wordfree(wordexp_t *p);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

wordexp(), **wordfree**():
_XOPEN_SOURCE

DESCRIPTION

The function **wordexp**() performs a shell-like expansion of the string *s* and returns the result in the structure pointed to by *p*. The data type *wordexp_t* is a structure that at least has the fields *we_wordc*, *we_wordv*, and *we_offs*. The field *we_wordc* is a *size_t* that gives the number of words in the expansion of *s*. The field *we_wordv* is a *char* ** that points to the array of words found. The field *we_offs* of type *size_t* is sometimes (depending on *flags*, see below) used to indicate the number of initial elements in the *we_wordv* array that should be filled with NULLs.

The function **wordfree**() frees the allocated memory again. More precisely, it does not free its argument, but it frees the array *we_wordv* and the strings that points to.

The string argument

Since the expansion is the same as the expansion by the shell (see *sh*(1)) of the parameters to a command, the string *s* must not contain characters that would be illegal in shell command parameters. In particular, there must not be any unescaped newline or |, &, ;, <, >, (,), {, } characters outside a command substitution or parameter substitution context.

If the argument *s* contains a word that starts with an unquoted comment character #, then it is unspecified whether that word and all following words are ignored, or the # is treated as a non-comment character.

The expansion

The expansion done consists of the following stages: tilde expansion (replacing ~user by user's home directory), variable substitution (replacing $FOO by the value of the environment variable FOO), command substitution (replacing $(command) or `command` by the output of command), arithmetic expansion, field splitting, wildcard expansion, quote removal.

The result of expansion of special parameters ($@, $*, $#, $?, $−, $$, $!, $0) is unspecified.

Field splitting is done using the environment variable $IFS. If it is not set, the field separators are space, tab, and newline.

The output array

The array *we_wordv* contains the words found, followed by a NULL.

The flags argument

The *flag* argument is a bitwise inclusive OR of the following values:

WRDE_APPEND
Append the words found to the array resulting from a previous call.

WRDE_DOOFFS
Insert *we_offs* initial NULLs in the array *we_wordv*. (These are not counted in the returned *we_wordc*.)

WRDE_NOCMD
Don't do command substitution.

WRDE_REUSE
The argument *p* resulted from a previous call to **wordexp**(), and **wordfree**() was not called. Reuse the allocated storage.

WRDE_SHOWERR
Normally during command substitution *stderr* is redirected to */dev/null*. This flag specifies that *stderr* is not to be redirected.

WRDE_UNDEF
Consider it an error if an undefined shell variable is expanded.

RETURN VALUE

On success, **wordexp**() returns 0. On failure, **wordexp**() returns one of the following nonzero values:

WRDE_BADCHAR
Illegal occurrence of newline or one of |, &, ;, <, >, (,), {, }.

WRDE_BADVAL
An undefined shell variable was referenced, and the **WRDE_UNDEF** flag told us to consider this an error.

WRDE_CMDSUB
Command substitution requested, but the **WRDE_NOCMD** flag told us to consider this an error.

WRDE_NOSPACE
Out of memory.

WRDE_SYNTAX
Shell syntax error, such as unbalanced parentheses or unmatched quotes.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wordexp()	Thread safety	MT-Unsafe race:utent const:env env sig:ALRM timer locale
wordfree()	Thread safety	MT-Safe

In the above table, *utent* in *race:utent* signifies that if any of the functions *setutent*(3), *getutent*(3), or *endutent*(3) are used in parallel in different threads of a program, then data races could occur. **wordexp**() calls those functions, so we use race:utent to remind users.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. glibc 2.1.

EXAMPLES

The output of the following example program is approximately that of "ls [a-c]*.c".

```
#include <stdio.h>
#include <stdlib.h>
#include <wordexp.h>

int
main(void)
{
    wordexp_t p;
    char **w;

    wordexp("[a-c]*.c", &p, 0);
    w = p.we_wordv;
    for (size_t i = 0; i < p.we_wordc; i++)
        printf("%s\n", w[i]);
    wordfree(&p);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

fnmatch(3), *glob*(3)

NAME

wprintf, fwprintf, swprintf, vwprintf, vfwprintf, vswprintf − formatted wide-character output conversion

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int wprintf(const wchar_t *restrict format, ...);
int fwprintf(FILE *restrict stream,
        const wchar_t *restrict format, ...);
int swprintf(wchar_t wcs[restrict .maxlen], size_t maxlen,
        const wchar_t *restrict format, ...);

int vwprintf(const wchar_t *restrict format, va_list args);
int vfwprintf(FILE *restrict stream,
        const wchar_t *restrict format, va_list args);
int vswprintf(wchar_t wcs[restrict .maxlen], size_t maxlen,
        const wchar_t *restrict format, va_list args);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

All functions shown above:
_XOPEN_SOURCE >= 500 || _ISOC99_SOURCE
|| _POSIX_C_SOURCE >= 200112L

DESCRIPTION

The **wprintf**() family of functions is the wide-character equivalent of the *printf*(3) family of functions. It performs formatted output of wide characters.

The **wprintf**() and **vwprintf**() functions perform wide-character output to *stdout*. *stdout* must not be byte oriented; see *fwide*(3) for more information.

The **fwprintf**() and **vfwprintf**() functions perform wide-character output to *stream*. *stream* must not be byte oriented; see *fwide*(3) for more information.

The **swprintf**() and **vswprintf**() functions perform wide-character output to an array of wide characters. The programmer must ensure that there is room for at least *maxlen* wide characters at *wcs*.

These functions are like the *printf*(3), *vprintf*(3), *fprintf*(3), *vfprintf*(3), *sprintf*(3), *vsprintf*(3) functions except for the following differences:

- The *format* string is a wide-character string.
- The output consists of wide characters, not bytes.
- **swprintf**() and **vswprintf**() take a *maxlen* argument, *sprintf*(3) and *vsprintf*(3) do not. (**snprintf**(3) and *vsnprintf*(3) take a *maxlen* argument, but these functions do not return −1 upon buffer overflow on Linux.)

The treatment of the conversion characters **c** and **s** is different:

c If no **l** modifier is present, the *int* argument is converted to a wide character by a call to the *btowc*(3) function, and the resulting wide character is written. If an **l** modifier is present, the *wint_t* (wide character) argument is written.

s If no **l** modifier is present: the *const char* * argument is expected to be a pointer to an array of character type (pointer to a string) containing a multibyte character sequence beginning in the initial shift state. Characters from the array are converted to wide characters (each by a call to the *mbrtowc*(3) function with a conversion state starting in the initial state before the first byte). The resulting wide characters are written up to (but not including) the terminating null wide character (L'\0'). If a precision is specified, no more wide characters than the number specified are written. Note that the precision determines the number of *wide characters* written, not the number of *bytes* or *screen positions*. The array must contain a terminating null byte ('\0'), unless a precision is given and it is so small that the number of converted wide characters reaches it before the end of the array is reached. If an **l** modifier is present: the

*const wchar_t ** argument is expected to be a pointer to an array of wide characters. Wide characters from the array are written up to (but not including) a terminating null wide character. If a precision is specified, no more than the number specified are written. The array must contain a terminating null wide character, unless a precision is given and it is smaller than or equal to the number of wide characters in the array.

RETURN VALUE

The functions return the number of wide characters written, excluding the terminating null wide character in case of the functions **swprintf**() and **vswprintf**(). They return −1 when an error occurs.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
wprintf(), **fwprintf**(), **swprintf**(), **vwprintf**(), **vfwprintf**(), **vswprintf**()	Thread safety	MT-Safe locale

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C99.

NOTES

The behavior of **wprintf**() et al. depends on the **LC_CTYPE** category of the current locale.

If the *format* string contains non-ASCII wide characters, the program will work correctly only if the **LC_CTYPE** category of the current locale at run time is the same as the **LC_CTYPE** category of the current locale at compile time. This is because the *wchar_t* representation is platform- and locale-dependent. (The glibc represents wide characters using their Unicode (ISO/IEC 10646) code point, but other platforms don't do this. Also, the use of C99 universal character names of the form \unnnn does not solve this problem.) Therefore, in internationalized programs, the *format* string should consist of ASCII wide characters only, or should be constructed at run time in an internationalized way (e.g., using *gettext*(3) or *iconv*(3), followed by *mbstowcs*(3)).

SEE ALSO

fprintf(3), *fputwc*(3), *fwide*(3), *printf*(3), *snprintf*(3)

NAME

xencrypt, xdecrypt, passwd2des − RFS password encryption

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <rpc/des_crypt.h>

void passwd2des(char **passwd***, char ***_key_**);**

int xencrypt(char ***_secret_, char ***_passwd_**);**
int xdecrypt(char ***_secret_, char ***_passwd_**);**

DESCRIPTION

WARNING: Do not use these functions in new code. They do not achieve any type of acceptable cryptographic security guarantees.

The function **passwd2des**() takes a character string *passwd* of arbitrary length and fills a character array *key* of length 8. The array *key* is suitable for use as DES key. It has odd parity set in bit 0 of each byte. Both other functions described here use this function to turn their argument *passwd* into a DES key.

The **xencrypt**() function takes the ASCII character string *secret* given in hex, which must have a length that is a multiple of 16, encrypts it using the DES key derived from *passwd* by **passwd2des**(), and outputs the result again in *secret* as a hex string of the same length.

The **xdecrypt**() function performs the converse operation.

RETURN VALUE

The functions **xencrypt**() and **xdecrypt**() return 1 on success and 0 on error.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
passwd2des(), **xencrypt**(), **xdecrypt**()	Thread safety	MT-Safe

VERSIONS

These functions are available since glibc 2.1.

BUGS

The prototypes are missing from the abovementioned include file.

SEE ALSO

cbc_crypt(3)

NAME

xdr − library routines for external data representation

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS AND DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

The prototypes below are declared in <*rpc/xdr.h*> and make use of the following types:

```
typedef int bool_t;

typedef bool_t (*xdrproc_t)(XDR *, void *,...);
```

For the declaration of the *XDR* type, see <*rpc/xdr.h*>.

bool_t xdr_array(XDR ****xdrs*, char *****arrp***, unsigned int ****sizep***,**
unsigned int *maxsize***, unsigned int** *elsize***,**
xdrproc_t *elproc***);**

A filter primitive that translates between variable-length arrays and their corresponding external representations. The argument *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The argument *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_bool(XDR ****xdrs*, bool_t ****bp***);**

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_bytes(XDR ****xdrs*, char *****sp***, unsigned int ****sizep***,**
unsigned int *maxsize***);**

A filter primitive that translates between counted byte strings and their external representations. The argument *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_char(XDR ****xdrs*, char ****cp***);**

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes**(), **xdr_opaque**(), or **xdr_string**().

void xdr_destroy(XDR ****xdrs*);**

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy**() is undefined.

bool_t xdr_double(XDR ****xdrs*, double ****dp***);**

A filter primitive that translates between C *double* precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_enum(XDR ****xdrs*, enum_t ****ep***);**

A filter primitive that translates between C *enum*s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_float(XDR ****xdrs*, float *** *fp***);**

A filter primitive that translates between C *float*s and their external representations. This routine returns one if it succeeds, zero otherwise.

void xdr_free(xdrproc_t *proc***, char ****objp***);**

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

unsigned int xdr_getpos(XDR *****xdrs*);**

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

long *xdr_inline(XDR *****xdrs*, int** *len***);**

A macro that invokes the inline routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to *long **.

Warning: **xdr_inline**() may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

bool_t xdr_int(XDR *****xdrs*, int *****ip***);**

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_long(XDR *****xdrs*, long *****lp***);**

A filter primitive that translates between C *long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

void xdrmem_create(XDR *****xdrs*, char *****addr***, unsigned int** *size***,**
enum xdr_op *op***);**

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

bool_t xdr_opaque(XDR *****xdrs*, char *****cp***, unsigned int** *cnt***);**

A filter primitive that translates between fixed size opaque data and its external representation. The argument *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_pointer(XDR *****xdrs*, char ******objpp***,**
unsigned int *objsize***, xdrproc_t** *xdrobj***);**

Like **xdr_reference**() except that it serializes null pointers, whereas **xdr_reference**() does not. Thus, **xdr_pointer**() can represent recursive data structures, such as binary trees or linked lists.

void xdrrec_create(XDR *****xdrs*, unsigned int** *sendsize***,**
unsigned int *recvsize***, char *****handle***,**
int (***readit***)(char *, char *, int),**
int (***writeit***)(char *, char *, int));**

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the system calls *read*(2) and *write*(2), except that *handle* is passed to the former routines as the first argument. Note: the XDR stream's *op* field must be set by the caller.

Warning: to read from an XDR stream created by this API, you'll need to call **xdrrec_skiprecord**() first before calling any other XDR APIs. This inserts additional bytes in the stream to provide record boundary information. Also, XDR streams created with different **xdr*_create** APIs are not compatible for the same reason.

bool_t xdrrec_endofrecord(XDR ****xdrs*, int** *sendnow***);**

This routine can be invoked only on streams created by **xdrrec_create**(). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is nonzero. This routine returns one if it succeeds, zero otherwise.

bool_t xdrrec_eof(XDR ****xdrs*);**

This routine can be invoked only on streams created by **xdrrec_create**(). After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

bool_t xdrrec_skiprecord(XDR ****xdrs*);**

This routine can be invoked only on streams created by **xdrrec_create**(). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_reference(XDR ****xdrs*, char *****pp***, unsigned int** *size***,**
xdrproc_t *proc***);**

A primitive that provides pointer chasing within structures. The argument *pp* is the address of the pointer; *size* is the *sizeof* the structure that **pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

Warning: this routine does not understand null pointers. Use **xdr_pointer**() instead.

xdr_setpos(XDR ****xdrs*, unsigned int** *pos***);**

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The argument *pos* is a position value obtained from **xdr_getpos**(). This routine returns one if the XDR stream could be repositioned, and zero otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

bool_t xdr_short(XDR ****xdrs*, short ****sp***);**

A filter primitive that translates between C *short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

void xdrstdio_create(XDR ****xdrs*, FILE ****file***, enum xdr_op** *op***);**

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the *stdio* stream *file*. The argument *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

Warning: the destroy routine associated with such XDR streams calls *fflush*(3) on the *file* stream, but never *fclose*(3).

bool_t xdr_string(XDR ****xdrs*, char *****sp***, unsigned int** *maxsize***);**

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_u_char(XDR ****xdrs*, unsigned char ****ucp***);**

A filter primitive that translates between *unsigned* C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_u_int(XDR ****xdrs*, unsigned int ****up***);**

A filter primitive that translates between C *unsigned* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_u_long(XDR ****xdrs*, unsigned long ****ulp***);**

A filter primitive that translates between C *unsigned long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_u_short(XDR ****xdrs*, unsigned short ****usp***);**

A filter primitive that translates between C *unsigned short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_union(XDR ****xdrs*, enum_t ****dscmp***, char ****unp***,**
const struct xdr_discrim ****choices*,**
xdrproc_t *defaultarm***); /* may equal NULL */**

A filter primitive that translates between a discriminated C *union* and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an *enum_t*. Next the union located at *unp* is translated. The argument *choices* is a pointer to an array of **xdr_discrim**() structures. Each structure contains an ordered pair of [*value*,*proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the **xdr_discrim**() structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns one if it succeeds, zero otherwise.

bool_t xdr_vector(XDR ****xdrs*, char ****arrp***, unsigned int** *size***,**
unsigned int *elsize***, xdrproc_t** *elproc***);**

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The argument *arrp* is the address of the pointer to the array, while *size* is the element count of the array. The argument *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

bool_t xdr_void(void);

This routine always returns one. It may be passed to RPC routines that require a function argument, where nothing is to be done.

bool_t xdr_wrapstring(XDR ****xdrs*, char *****sp***);**

A primitive that calls **xdr_string(xdrs, sp,MAXUN.UNSIGNED);** where **MAXUN.UNSIGNED** is the maximum value of an unsigned integer. **xdr_wrapstring**() is handy because the RPC package passes a maximum of two XDR routines as arguments, and **xdr_string**(), one of the most frequently used primitives, requires three. Returns one if it succeeds, zero otherwise.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
xdr_array(), **xdr_bool**(), **xdr_bytes**(), **xdr_char**(), **xdr_destroy**(), **xdr_double**(), **xdr_enum**(), **xdr_float**(), **xdr_free**(), **xdr_getpos**(), **xdr_inline**(), **xdr_int**(), **xdr_long**(), **xdrmem_create**(), **xdr_opaque**(), **xdr_pointer**(), **xdrrec_create**(), **xdrrec_eof**(), **xdrrec_endofrecord**(), **xdrrec_skiprecord**(), **xdr_reference**(), **xdr_setpos**(), **xdr_short**(), **xdrstdio_create**(), **xdr_string**(), **xdr_u_char**(), **xdr_u_int**(), **xdr_u_long**(), **xdr_u_short**(), **xdr_union**(), **xdr_vector**(), **xdr_void**(), **xdr_wrapstring**()	Thread safety	MT-Safe

SEE ALSO

rpc(3)

The following manuals:

eXternal Data Representation Standard: Protocol Specification
eXternal Data Representation: Sun Technical Notes
XDR: External Data Representation Standard, RFC 1014, Sun Microsystems, Inc., USC-ISI.

NAME

y0, y0f, y0l, y1, y1f, y1l, yn, ynf, ynl − Bessel functions of the second kind

LIBRARY

Math library (*libm*, −*lm*)

SYNOPSIS

```
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);

float y0f(float x);
float y1f(float x);
float ynf(int n, float x);

long double y0l(long double x);
long double y1l(long double x);
long double ynl(int n, long double x);
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

y0(), **y1**(), **yn**():
_XOPEN_SOURCE
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

y0f(), **y0l**(), **y1f**(), **y1l**(), **ynf**(), **ynl**():
_XOPEN_SOURCE >= 600
|| (_ISOC99_SOURCE && _XOPEN_SOURCE)
|| /* Since glibc 2.19: */ _DEFAULT_SOURCE
|| /* glibc <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

DESCRIPTION

The **y0**() and **y1**() functions return Bessel functions of *x* of the second kind of orders 0 and 1, respectively. The **yn**() function returns the Bessel function of *x* of the second kind of order *n*.

The value of *x* must be positive.

The **y0f**(), **y1f**(), and **ynf**() functions are versions that take and return *float* values. The **y0l**(), **y1l**(), and **ynl**() functions are versions that take and return *long double* values.

RETURN VALUE

On success, these functions return the appropriate Bessel value of the second kind for *x*.

If *x* is a NaN, a NaN is returned.

If *x* is negative, a domain error occurs, and the functions return −**HUGE_VAL**, −**HUGE_VALF**, or −**HUGE_VALL**, respectively. (POSIX.1-2001 also allows a NaN return for this case.)

If *x* is 0.0, a pole error occurs, and the functions return −**HUGE_VAL**, −**HUGE_VALF**, or −**HUGE_VALL**, respectively.

If the result underflows, a range error occurs, and the functions return 0.0

If the result overflows, a range error occurs, and the functions return −**HUGE_VAL**, −**HUGE_VALF**, or −**HUGE_VALL**, respectively. (POSIX.1-2001 also allows a 0.0 return for this case.)

ERRORS

See *math_error*(7) for information on how to determine whether an error has occurred when calling these functions.

The following errors can occur:

Domain error: *x* is negative
: *errno* is set to **EDOM**. An invalid floating-point exception (**FE_INVALID**) is raised.

Pole error: *x* is 0.0
: *errno* is set to **ERANGE** and an **FE_DIVBYZERO** exception is raised (but see BUGS).

Range error: result underflow
: *errno* is set to **ERANGE**. No **FE_UNDERFLOW** exception is returned by *fetestexcept*(3) for this case.

Range error: result overflow
: *errno* is set to **ERANGE** (but see BUGS). An overflow floating-point exception (**FE_OVERFLOW**) is raised.

ATTRIBUTES

For an explanation of the terms used in this section, see *attributes*(7).

Interface	Attribute	Value
y0(), **y0f**(), **y0l**()	Thread safety	MT-Safe
y1(), **y1f**(), **y1l**()	Thread safety	MT-Safe
yn(), **ynf**(), **ynl**()	Thread safety	MT-Safe

STANDARDS

y0()
y1()
yn() POSIX.1-2008.

Others: BSD.

HISTORY

y0()
y1()
yn() SVr4, 4.3BSD, POSIX.1-2001.

Others: BSD.

BUGS

Before glibc 2.19, these functions misdiagnosed pole errors: *errno* was set to **EDOM**, instead of **ERANGE** and no **FE_DIVBYZERO** exception was raised.

Before glibc 2.17, did not set *errno* for "range error: result underflow".

In glibc 2.3.2 and earlier, these functions do not raise an invalid floating-point exception (**FE_INVALID**) when a domain error occurs.

SEE ALSO

j0(3)

NAME

EOF − end of file or error indicator

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdio.h>

#define EOF  /* ... */
```

DESCRIPTION

EOF represents the end of an input file, or an error indication. It is a negative value, of type *int*.

EOF is not a character (it can't be represented by *unsigned char*). It is instead a sentinel value outside of the valid range for valid characters.

CONFORMING TO

C99 and later; POSIX.1-2001 and later.

CAVEATS

Programs can't pass this value to an output function to "write" the end of a file. That would likely result in undefined behavior. Instead, closing the writing stream or file descriptor that refers to such file is the way to signal the end of that file.

SEE ALSO

feof(3), *fgetc*(3)

NAME

EXIT_SUCCESS, EXIT_FAILURE − termination status constants

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <stdlib.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE /* nonzero */

DESCRIPTION

EXIT_SUCCESS and **EXIT_FAILURE** represent a successful and unsuccessful exit status respectively, and can be used as arguments to the *exit*(3) function.

CONFORMING TO

C99 and later; POSIX.1-2001 and later.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    /* Other code omitted */

    fclose(fp);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

exit(3), *sysexits.h*(3head)

NAME

NULL − null pointer constant

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stddef.h>

#define NULL  ((void *) 0)
```

DESCRIPTION

NULL represents a null pointer constant, that is, a pointer that does not point to anything.

CONFORMING TO

C99 and later; POSIX.1-2001 and later.

NOTES

The following headers also provide **NULL**: <*locale.h*>, <*stdio.h*>, <*stdlib.h*>, <*string.h*>, <*time.h*>, <*unistd.h*>, and <*wchar.h*>.

CAVEATS

It is undefined behavior to dereference a null pointer, and that usually causes a segmentation fault in practice.

It is also undefined behavior to perform pointer arithmetic on it.

NULL − *NULL* is undefined behavior, according to ISO C, but is defined to be 0 in C++.

To avoid confusing human readers of the code, do not compare pointer variables to **0**, and do not assign **0** to them. Instead, always use **NULL**.

NULL shouldn't be confused with **NUL**, which is an *ascii*(7) character, represented in C as '**\0**'.

BUGS

When it is necessary to set a pointer variable to a null pointer, it is not enough to use *memset*(3) to zero the pointer (this is usually done when zeroing a struct that contains pointers), since ISO C and POSIX don't guarantee that a bit pattern of all 0s represent a null pointer. See the EXAMPLES section in *getaddrinfo*(3) for an example program that does this correctly.

SEE ALSO

void(3type)

NAME

printf.h, register_printf_specifier, register_printf_modifier, register_printf_type, printf_function, printf_arginfo_size_function, printf_va_arg_function, printf_info, PA_INT, PA_CHAR, PA_WCHAR, PA_STRING, PA_WSTRING, PA_POINTER, PA_FLOAT, PA_DOUBLE, PA_LAST, PA_FLAG_LONG_LONG, PA_FLAG_LONG_DOUBLE, PA_FLAG_LONG, PA_FLAG_SHORT, PA_FLAG_PTR − define custom behavior for printf-like functions

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <printf.h>

int register_printf_specifier(int spec, printf_function  func,
                              printf_arginfo_size_function arginfo);
int register_printf_modifier(const wchar_t *str);
int register_printf_type(printf_va_arg_function  fct);
```

Callbacks

```
typedef int printf_function(FILE *stream, const struct printf_info *info,
                            const void *const args[]);
typedef int printf_arginfo_size_function(const struct printf_info *info,
                            size_t n, int argtypes[n], int size[n]);
typedef void printf_va_arg_function(void *mem, va_list *ap);
```

Types

```
struct printf_info {
    int            prec;            // Precision
    int            width;           // Width
    wchar_t        spec;            // Format letter
    unsigned int   is_long_double:1;// L or ll flag
    unsigned int   is_short:1;      // h flag
    unsigned int   is_long:1;       // l flag
    unsigned int   alt:1;           // # flag
    unsigned int   space:1;         // Space flag
    unsigned int   left:1;          // - flag
    unsigned int   showsign:1;      // + flag
    unsigned int   group:1;         // ' flag
    unsigned int   extra:1;         // For special use
    unsigned int   is_char:1;       // hh flag
    unsigned int   wide:1;          // True for wide character streams
    unsigned int   i18n:1;          // I flag
    unsigned int   is_binary128:1;  /* Floating-point argument is
                                       ABI-compatible with
                                       IEC 60559 binary128 */
    unsigned short user;            // Bits for user-installed modifiers
    wchar_t        pad;             // Padding character
};
```

Constants

```
#define PA_FLAG_LONG_LONG    /* ... */
#define PA_FLAG_LONG_DOUBLE  /* ... */
#define PA_FLAG_LONG         /* ... */
#define PA_FLAG_SHORT        /* ... */
#define PA_FLAG_PTR          /* ... */
```

DESCRIPTION

These functions serve to extend and/or modify the behavior of the *printf*(3) family of functions.

register_printf_specifier()

This function registers a custom conversion specifier for the *printf*(3) family of functions.

spec The character which will be used as a conversion specifier in the format string.

func Callback function that will be executed by the *printf*(3) family of functions to format the input arguments into the output *stream*.

stream Output stream where the formatted output should be printed. This stream transparently represents the output, even in the case of functions that write to a string.

info Structure that holds context information, including the modifiers specified in the format string. This holds the same contents as in *arginfo*.

args Array of pointers to the arguments to the *printf*(3)-like function.

arginfo Callback function that will be executed by the *printf*(3) family of functions to know how many arguments should be parsed for the custom specifier and also their types.

info Structure that holds context information, including the modifiers specified in the format string. This holds the same contents as in *func*.

n Number of arguments remaining to be parsed.

argtypes
This array should be set to define the type of each of the arguments that will be parsed. Each element in the array represents one of the arguments to be parsed, in the same order that they are passed to the *printf*(3)-like function. Each element should be set to a base type (**PA_***) from the enum above, or a custom one, and optionally ORed with an appropriate length modifier (**PA_FLAG_***).

The type is determined by using one of the following constants:

PA_INT
int.

PA_CHAR
int, cast to *char*.

PA_WCHAR
wchar_t.

PA_STRING
const char *, a '\0'-terminated string.

PA_WSTRING
const wchar_t *, a wide character L'\0'-terminated string.

PA_POINTER
void *.

PA_FLOAT
float.

PA_DOUBLE
double.

PA_LAST
TODO.

size For user-defined types, the size of the type (in bytes) should also be specified through this array. Otherwise, leave it unused.

arginfo is called before *func*, and prepares some information needed to call *func*.

register_printf_modifier()
TODO

register_printf_type()
TODO

RETURN VALUE

register_printf_specifier(), **register_printf_modifier**(), and **register_printf_type**() return zero on success, or −1 on error.

Callbacks

The callback of type *printf_function* should return the number of characters written, or −1 on error.

The callback of type *printf_arginfo_size_function* should return the number of arguments to be parsed by this specifier. It also passes information about the type of those arguments to the caller through *argtypes*. On error, it should return −1.

ERRORS

EINVAL

The specifier was not a valid character.

STANDARDS

GNU.

HISTORY

register_printf_function(3) is an older function similar to **register_printf_specifier**(), and is now deprecated. That function can't handle user-defined types.

register_printf_specifier() supersedes **register_printf_function**(3).

EXAMPLES

The following example program registers the 'b' and 'B' specifiers to print integers in binary format, mirroring rules for other unsigned conversion specifiers like 'x' and 'u'. This can be used to print in binary prior to C23.

```
/* This code is in the public domain */

#include <err.h>
#include <limits.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/param.h>

#include <printf.h>

#define GROUP_SEP  '\''

struct Printf_Pad {
    char    ch;
    size_t  len;
};

static int b_printf(FILE *stream, const struct printf_info *info,
                    const void *const args[]);
static int b_arginf_sz(const struct printf_info *info,
                    size_t n, int argtypes[n], int size[n]);

static uintmax_t b_value(const struct printf_info *info,
                    const void *arg);
static size_t b_bin_repr(char bin[UINTMAX_WIDTH],
                    const struct printf_info *info, const void *arg);
static size_t b_bin_len(const struct printf_info *info,
                    ptrdiff_t min_len);
static size_t b_pad_len(const struct printf_info *info,
                    ptrdiff_t bin_len);
static ssize_t b_print_prefix(FILE *stream,
                    const struct printf_info *info);
static ssize_t b_pad_zeros(FILE *stream, const struct printf_info *info,
                    ptrdiff_t min_len);
static ssize_t b_print_number(FILE *stream,
```

```
                    const struct printf_info *info,
                    const char bin[UINTMAX_WIDTH],
                    size_t min_len, size_t bin_len);
static char pad_ch(const struct printf_info *info);
static ssize_t pad_spaces(FILE *stream, size_t pad_len);

int
main(void)
{
    if (register_printf_specifier('b', b_printf, b_arginf_sz) == -1)
        err(EXIT_FAILURE, "register_printf_specifier('b', ...)");
    if (register_printf_specifier('B', b_printf, b_arginf_sz) == -1)
        err(EXIT_FAILURE, "register_printf_specifier('B', ...)");

    printf("....----....----....----....----\n");
    printf("%llb;\n", 0x5Ellu);
    printf("%lB;\n", 0x5Elu);
    printf("%b;\n", 0x5Eu);
    printf("%hB;\n", 0x5Eu);
    printf("%hhb;\n", 0x5Eu);
    printf("%jb;\n", (uintmax_t)0x5E);
    printf("%zb;\n", (size_t)0x5E);
    printf("....----....----....----....----\n");
    printf("%#b;\n", 0x5Eu);
    printf("%#B;\n", 0x5Eu);
    printf("....----....----....----....----\n");
    printf("%10b;\n", 0x5Eu);
    printf("%010b;\n", 0x5Eu);
    printf("%.10b;\n", 0x5Eu);
    printf("....----....----....----....----\n");
    printf("%-10B;\n", 0x5Eu);
    printf("....----....----....----....----\n");
    printf("%'B;\n", 0x5Eu);
    printf("....----....----....----....----\n");
    printf("....----....----....----....----\n");
    printf("%#16.12b;\n", 0xAB);
    printf("%-#'20.12b;\n", 0xAB);
    printf("%#'020B;\n", 0xAB);
    printf("....----....----....----....----\n");
    printf("%#020B;\n", 0xAB);
    printf("%'020B;\n", 0xAB);
    printf("%020B;\n", 0xAB);
    printf("....----....----....----....----\n");
    printf("%#021B;\n", 0xAB);
    printf("%'021B;\n", 0xAB);
    printf("%021B;\n", 0xAB);
    printf("....----....----....----....----\n");
    printf("%#022B;\n", 0xAB);
    printf("%'022B;\n", 0xAB);
    printf("%022B;\n", 0xAB);
    printf("....----....----....----....----\n");
    printf("%#023B;\n", 0xAB);
    printf("%'023B;\n", 0xAB);
    printf("%023B;\n", 0xAB);
    printf("....----....----....----....----\n");
    printf("%-#'19.11b;\n", 0xAB);
    printf("%#'019B;\n", 0xAB);
    printf("%#019B;\n", 0xAB);
    printf("....----....----....----....----\n");
```

```
    printf("%'019B;\n", 0xAB);
    printf("%019B;\n", 0xAB);
    printf("%#016b;\n", 0xAB);
    printf("....----....----....----....----\n");

    return 0;
}

static int
b_printf(FILE *stream, const struct printf_info *info,
         const void *const args[])
{
    char               bin[UINTMAX_WIDTH];
    size_t             min_len, bin_len;
    ssize_t            len, tmp;
    struct Printf_Pad  pad = {0};

    len = 0;

    min_len = b_bin_repr(bin, info, args[0]);
    bin_len = b_bin_len(info, min_len);

    pad.ch = pad_ch(info);
    if (pad.ch == ' ')
        pad.len = b_pad_len(info, bin_len);

    /* Padding with ' ' (right aligned) */
    if ((pad.ch == ' ') && !info->left) {
        tmp = pad_spaces(stream, pad.len);
        if (tmp == EOF)
            return EOF;
        len += tmp;
    }

    /* "0b"/"0B" prefix */
    if (info->alt) {
        tmp = b_print_prefix(stream, info);
        if (tmp == EOF)
            return EOF;
        len += tmp;
    }

    /* Padding with '0' */
    if (pad.ch == '0') {
        tmp = b_pad_zeros(stream, info, min_len);
        if (tmp == EOF)
            return EOF;
        len += tmp;
    }

    /* Print number (including leading 0s to fill precision) */
    tmp = b_print_number(stream, info, bin, min_len, bin_len);
    if (tmp == EOF)
        return EOF;
    len += tmp;

    /* Padding with ' ' (left aligned) */
    if (info->left) {
        tmp = pad_spaces(stream, pad.len);
```

```
        if (tmp == EOF)
            return EOF;
        len += tmp;
    }

    return len;
}

static int
b_arginf_sz(const struct printf_info *info, size_t n, int argtypes[n],
            [[maybe_unused]] int size[n])
{
    if (n < 1)
        return -1;

    if (info->is_long_double)
        argtypes[0] = PA_INT | PA_FLAG_LONG_LONG;
    else if (info->is_long)
        argtypes[0] = PA_INT | PA_FLAG_LONG;
    else
        argtypes[0] = PA_INT;

    return 1;
}

static uintmax_t
b_value(const struct printf_info *info, const void *arg)
{
    if (info->is_long_double)
        return *(const unsigned long long *)arg;
    if (info->is_long)
        return *(const unsigned long *)arg;

    /* short and char are both promoted to int */
    return *(const unsigned int *)arg;
}

static size_t
b_bin_repr(char bin[UINTMAX_WIDTH],
           const struct printf_info *info, const void *arg)
{
    size_t     min_len;
    uintmax_t  val;

    val = b_value(info, arg);

    bin[0] = '0';
    for (min_len = 0; val; min_len++) {
        bin[min_len] = '0' + (val % 2);
        val >>= 1;
    }

    return MAX(min_len, 1);
}

static size_t
b_bin_len(const struct printf_info *info, ptrdiff_t min_len)
{
    return MAX(info->prec, min_len);
```

```
}

static size_t
b_pad_len(const struct printf_info *info, ptrdiff_t bin_len)
{
    ptrdiff_t  pad_len;

    pad_len = info->width - bin_len;
    if (info->alt)
        pad_len -= 2;
    if (info->group)
        pad_len -= (bin_len - 1) / 4;

    return MAX(pad_len, 0);
}

static ssize_t
b_print_prefix(FILE *stream, const struct printf_info *info)
{
    ssize_t len;

    len = 0;
    if (fputc('0', stream) == EOF)
        return EOF;
    len++;
    if (fputc(info->spec, stream) == EOF)
        return EOF;
    len++;

    return len;
}

static ssize_t
b_pad_zeros(FILE *stream, const struct printf_info *info,
            ptrdiff_t min_len)
{
    ssize_t    len;
    ptrdiff_t  tmp;

    len = 0;
    tmp = info->width - (info->alt * 2);
    if (info->group)
        tmp -= tmp / 5 - !(tmp % 5);
    for (ptrdiff_t i = tmp - 1; i > min_len - 1; i--) {
        if (fputc('0', stream) == EOF)
            return EOF;
        len++;

        if (!info->group || (i % 4))
            continue;
        if (fputc(GROUP_SEP, stream) == EOF)
            return EOF;
        len++;
    }

    return len;
}

static ssize_t
```

```
b_print_number(FILE *stream, const struct printf_info *info,
               const char bin[UINTMAX_WIDTH],
               size_t min_len, size_t bin_len)
{
    ssize_t  len;

    len = 0;

    /* Print leading zeros to fill precision */
    for (size_t i = bin_len - 1; i > min_len - 1; i--) {
        if (fputc('0', stream) == EOF)
            return EOF;
        len++;

        if (!info->group || (i % 4))
            continue;
        if (fputc(GROUP_SEP, stream) == EOF)
            return EOF;
        len++;
    }

    /* Print number */
    for (size_t i = min_len - 1; i < min_len; i--) {
        if (fputc(bin[i], stream) == EOF)
            return EOF;
        len++;

        if (!info->group || (i % 4) || !i)
            continue;
        if (fputc(GROUP_SEP, stream) == EOF)
            return EOF;
        len++;
    }

    return len;
}

static char
pad_ch(const struct printf_info *info)
{
    if ((info->prec != -1) || (info->pad == ' ') || info->left)
        return ' ';
    return '0';
}

static ssize_t
pad_spaces(FILE *stream, size_t pad_len)
{
    ssize_t  len;

    len = 0;
    for (size_t i = pad_len - 1; i < pad_len; i--) {
        if (fputc(' ', stream) == EOF)
            return EOF;
        len++;
    }

    return len;
}
```

SEE ALSO

asprintf(3), *printf*(3), *wprintf*(3)

NAME

sysexits.h − exit codes for programs

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sysexits.h>
#define EX_OK          0   /* successful termination */

    #define EX__BASE      64   /* base value for error messages */

    #define EX_USAGE      64   /* command line usage error */
#define EX_DATAERR     65  /* data format error */
#define EX_NOINPUT     66  /* cannot open input */
#define EX_NOUSER      67  /* addressee unknown */
#define EX_NOHOST      68  /* host name unknown */
#define EX_UNAVAILABLE 69  /* service unavailable */
#define EX_SOFTWARE    70  /* internal software error */
#define EX_OSERR       71  /* system error (e.g., can't fork) */
#define EX_OSFILE      72  /* critical OS file missing */
#define EX_CANTCREAT   73  /* can't create (user) output file */
#define EX_IOERR       74  /* input/output error */
#define EX_TEMPFAIL    75  /* temp failure; user is invited to retry */
#define EX_PROTOCOL    76  /* remote error in protocol */
#define EX_NOPERM      77  /* permission denied */
#define EX_CONFIG      78  /* configuration error */

    #define EX__MAX       ...   /* maximum listed value */
```

DESCRIPTION

A few programs exit with the following error codes.

The successful exit is always indicated by a status of **0**, or **EX_OK** (equivalent to **EXIT_SUCCESS** from <*stdlib.h*>). Error numbers begin at **EX__BASE** to reduce the possibility of clashing with other exit statuses that random programs may already return. The meaning of the code is approximately as follows:

EX_USAGE
: The command was used incorrectly, e.g., with the wrong number of arguments, a bad flag, bad syntax in a parameter, or whatever.

EX_DATAERR
: The input data was incorrect in some way. This should only be used for user's data and not system files.

EX_NOINPUT
: An input file (not a system file) did not exist or was not readable. This could also include errors like "No message" to a mailer (if it cared to catch it).

EX_NOUSER
: The user specified did not exist. This might be used for mail addresses or remote logins.

EX_NOHOST
: The host specified did not exist. This is used in mail addresses or network requests.

EX_UNAVAILABLE
: A service is unavailable. This can occur if a support program or file does not exist. This can also be used as a catch-all message when something you wanted to do doesn't work, but you don't know why.

EX_SOFTWARE
: An internal software error has been detected. This should be limited to non-operating system related errors if possible.

EX_OSERR
: An operating system error has been detected. This is intended to be used for such things as "cannot fork", "cannot create pipe", or the like. It includes things like *getuid*(2) returning a

user that does not exist in the *passwd*(5) file.

EX_OSFILE
Some system file (e.g., */etc/passwd*, */etc/utmp*, etc.) does not exist, cannot be opened, or has some sort of error (e.g., syntax error).

EX_CANTCREAT
A (user specified) output file cannot be created.

EX_IOERR
An error occurred while doing I/O on some file.

EX_TEMPFAIL
Temporary failure, indicating something that is not really an error. For example that a mailer could not create a connection, and the request should be reattempted later.

EX_PROTOCOL
The remote system returned something that was "not possible" during a protocol exchange.

EX_OSFILE
You did not have sufficient permission to perform the operation. This is not intended for file system problems, which should use **EX_NOINPUT** or **EX_CANTCREAT**, but rather for higher level permissions.

EX_CONFIG
Something was found in an unconfigured or misconfigured state.

The numerical values corresponding to the symbolical ones are given in parenthesis for easy reference.

STANDARDS
BSD.

HISTORY
The <*sysexits.h*> file appeared in 4.0BSD for use by the deliverymail utility, later renamed to *sendmail*(8)

CAVEATS
The choice of an appropriate exit value is often ambiguous.

SEE ALSO
err(3), *error*(3), *exit*(3)

NAME

aiocb − asynchronous I/O control block

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <aio.h>

struct aiocb {
    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority offset */
    struct sigevent aio_sigevent;   /* Signal number and value */
    int             aio_lio_opcode; /* Operation to be performed */
};
```

DESCRIPTION

For further information about this structure, see *aio*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

aio_cancel(3), *aio_error*(3), *aio_fsync*(3), *aio_read*(3), *aio_return*(3), *aio_suspend*(3), *aio_write*(3), *lio_listio*(3)

NAME

blkcnt_t − file block counts

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ blkcnt_t;
```

DESCRIPTION

Used for file block counts. It is a signed integer type.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides this type: <*sys/stat.h*>.

SEE ALSO

stat(3type)

NAME

blksize_t − file block sizes

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <sys/types.h>

typedef /* ... */ **blksize_t;**

DESCRIPTION

Used for file block sizes. It is a signed integer type.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides this type: <*sys/stat.h*>.

SEE ALSO

stat(3type)

NAME

cc_t, speed_t, tcflag_t − terminal special characters, baud rates, modes

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <termios.h>

typedef /* ... */ cc_t;
typedef /* ... */ speed_t;
typedef /* ... */ tcflag_t;
```

DESCRIPTION

cc_t is used for terminal special characters, *speed_t* for baud rates, and *tcflag_t* for modes.

All are unsigned integer types.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

termios(3)

NAME

clock_t − system time

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <time.h>

typedef /* ... */ **clock_t;**

DESCRIPTION

Used for system time in clock ticks or **CLOCKS_PER_SEC** (defined in <*time.h*>). According to POSIX, it is an integer type or a real-floating type.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

NOTES

The following headers also provide this type: <*sys/types.h*> and <*sys/times.h*>.

SEE ALSO

times(2), *clock*(3)

NAME

clockid_t − clock ID for the clock and timer functions

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <sys/types.h>

typedef /* ... */ **clockid_t;**

DESCRIPTION

Used for clock ID type in the clock and timer functions. It is defined as an arithmetic type.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides this type: <*time.h*>.

SEE ALSO

clock_adjtime(2), *clock_getres*(2), *clock_nanosleep*(2), *timer_create*(2), *clock_getcpuclockid*(3)

NAME

dev_t − device ID

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ dev_t;
```

DESCRIPTION

Used for device IDs. It is an integer type. For further details of this type, see *makedev*(3).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides this type: <*sys/stat.h*>.

SEE ALSO

mknod(2), *stat*(3type)

NAME

div_t, ldiv_t, lldiv_t, imaxdiv_t − quotient and remainder of an integer division

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdlib.h>

typedef struct {
    int quot; /* Quotient */
    int rem;  /* Remainder */
} div_t;

typedef struct {
    long quot; /* Quotient */
    long rem;  /* Remainder */
} ldiv_t;

typedef struct {
    long long quot; /* Quotient */
    long long rem;  /* Remainder */
} lldiv_t;

#include <inttypes.h>

typedef struct {
    intmax_t quot; /* Quotient */
    intmax_t rem;  /* Remainder */
} imaxdiv_t;
```

DESCRIPTION

[[*l*]*l*]*div_t* is the type of the value returned by the **[[l]l]***div*(3) function.

imaxdiv_t is the type of the value returned by the *imaxdiv*(3) function.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

SEE ALSO

div(3), *imaxdiv*(3), *ldiv*(3), *lldiv*(3)

NAME

float_t, double_t − most efficient floating types

LIBRARY

Math library (*libm*)

SYNOPSIS

#include <math.h>

typedef /* ... */ **float_t;**
typedef /* ... */ **double_t;**

DESCRIPTION

The implementation's most efficient floating types at least as wide as *float* and *double* respectively. Their type depends on the value of the macro **FLT_EVAL_METHOD** (defined in <*float.h*>):

FLT_EVAL_METHOD	*float_t*	*double_t*
0	*float*	*double*
1	*double*	*double*
2	*long double*	*long double*

For other values of **FLT_EVAL_METHOD**, the types of *float_t* and *double_t* are implementation-defined.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

SEE ALSO

float.h(0p), *math.h*(0p)

NAME

epoll_event, epoll_data, epoll_data_t − epoll event

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/epoll.h>

struct epoll_event {
    uint32_t      events;  /* Epoll events */
    epoll_data_t  data;    /* User data variable */
};

union epoll_data {
    void     *ptr;
    int       fd;
    uint32_t  u32;
    uint64_t  u64;
};

typedef union epoll_data  epoll_data_t;
```

DESCRIPTION

The *epoll_event* structure specifies data that the kernel should save and return when the corresponding file descriptor becomes ready.

VERSIONS

C library/kernel differences

The Linux kernel headers also provide this type, with a slightly different definition:

```
#include <linux/eventpoll.h>

struct epoll_event {
    __poll_t  events;
    __u64     data;
};
```

STANDARDS

Linux.

SEE ALSO

epoll_wait(2), *epoll_ctl*(2)

NAME

fenv_t, fexcept_t − floating-point environment

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <fenv.h>

typedef /* ... */ fenv_t;
typedef /* ... */ fexcept_t;
```

DESCRIPTION

fenv_t represents the entire floating-point environment, including control modes and status flags.

fexcept_t represents the floating-point status flags collectively.

For further details see *fenv*(3).

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

SEE ALSO

fenv(3)

NAME

FILE − input/output stream

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdio.h>

typedef /* ... */ FILE;
```

DESCRIPTION

An object type used for streams.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

NOTES

The following header also provides this type: <*wchar.h*>.

SEE ALSO

fclose(3), *flockfile*(3), *fopen*(3), *fprintf*(3), *fread*(3), *fscanf*(3), *stdin*(3), *stdio*(3)

NAME

pid_t, uid_t, gid_t, id_t − process/user/group identifier

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ pid_t;
typedef /* ... */ uid_t;
typedef /* ... */ gid_t;
typedef /* ... */ id_t;
```

DESCRIPTION

pid_t is a type used for storing process IDs, process group IDs, and session IDs. It is a signed integer type.

uid_t is a type used to hold user IDs. It is an integer type.

gid_t is a type used to hold group IDs. It is an integer type.

id_t is a type used to hold a general identifier. It is an integer type that can be used to contain a *pid_t*, *uid_t*, or *gid_t*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following headers also provide *pid_t*: <*fcntl.h*>, <*sched.h*>, <*signal.h*>, <*spawn.h*>, <*sys/msg.h*>, <*sys/sem.h*>, <*sys/shm.h*>, <*sys/wait.h*>, <*termios.h*>, <*time.h*>, <*unistd.h*>, and <*utmpx.h*>.

The following headers also provide *uid_t*: <*pwd.h*>, <*signal.h*>, <*stropts.h*>, <*sys/ipc.h*>, <*sys/stat.h*>, and <*unistd.h*>.

The following headers also provide *gid_t*: <*grp.h*>, <*pwd.h*>, <*signal.h*>, <*stropts.h*>, <*sys/ipc.h*>, <*sys/stat.h*>, and <*unistd.h*>.

The following header also provides *id_t*: <*sys/resource.h*>.

SEE ALSO

chown(2), *fork*(2), *getegid*(2), *geteuid*(2), *getgid*(2), *getgroups*(2), *getpgid*(2), *getpid*(2), *getppid*(2), *getpriority*(2), *getpwnam*(3), *getresgid*(2), *getresuid*(2), *getsid*(2), *gettid*(2), *getuid*(2), *kill*(2), *pidfd_open*(2), *sched_setscheduler*(2), *waitid*(2), *getgrnam*(3), *sigqueue*(3), *credentials*(7)

NAME

intmax_t, uintmax_t − greatest-width basic integer types

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdint.h>

typedef /* ... */ intmax_t;
typedef /* ... */ uintmax_t;

#define INTMAX_WIDTH  /* ... */
#define UINTMAX_WIDTH INTMAX_WIDTH

#define INTMAX_MAX    /*  2**(INTMAX_WIDTH - 1) - 1  */
#define INTMAX_MIN    /*  - 2**(INTMAX_WIDTH - 1)    */
#define UINTMAX_MAX   /*  2**UINTMAX_WIDTH - 1       */

#define INTMAX_C(c)   c ## /* ... */
#define UINTMAX_C(c)  c ## /* ... */
```

DESCRIPTION

intmax_t is a signed integer type capable of representing any value of any basic signed integer type supported by the implementation. It is capable of storing values in the range [**INTMAX_MIN**, **INTMAX_MAX**].

uintmax_t is an unsigned integer type capable of representing any value of any basic unsigned integer type supported by the implementation. It is capable of storing values in the range [**0**, **UINTMAX_MAX**].

The macros [**U**]**INTMAX_WIDTH** expand to the width in bits of these types.

The macros [**U**]**INTMAX_MAX** expand to the maximum value that these types can hold.

The macro **INTMAX_MIN** expands to the minimum value that *intmax_t* can hold.

The macros [**U**]**INTMAX_C**() expand their argument to an integer constant of type [*u*]*intmax_t*.

The length modifier for [*u*]*intmax_t* for the *printf*(3) and the *scanf*(3) families of functions is **j**; resulting commonly in **%jd**, **%ji**, **%ju**, or **%jx** for printing [*u*]*intmax_t* values.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The following header also provides these types: <*inttypes.h*>.

BUGS

These types may not be as large as extended integer types, such as *__int128*

SEE ALSO

int64_t(3type), *intptr_t*(3type), *printf*(3), *strtoimax*(3)

NAME

intN_t, int8_t, int16_t, int32_t, int64_t, uintN_t, uint8_t, uint16_t, uint32_t, uint64_t − fixed-width basic integer types

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdint.h>

typedef /* ... */ int8_t;
typedef /* ... */ int16_t;
typedef /* ... */ int32_t;
typedef /* ... */ int64_t;

typedef /* ... */ uint8_t;
typedef /* ... */ uint16_t;
typedef /* ... */ uint32_t;
typedef /* ... */ uint64_t;

#define INT8_WIDTH   8
#define INT16_WIDTH  16
#define INT32_WIDTH  32
#define INT64_WIDTH  64

#define UINT8_WIDTH  8
#define UINT16_WIDTH 16
#define UINT32_WIDTH 32
#define UINT64_WIDTH 64

#define INT8_MAX     /*  2**(INT8_WIDTH - 1) - 1   */
#define INT16_MAX    /*  2**(INT16_WIDTH - 1) - 1  */
#define INT32_MAX    /*  2**(INT32_WIDTH - 1) - 1  */
#define INT64_MAX    /*  2**(INT64_WIDTH - 1) - 1  */

#define INT8_MIN     /*  - 2**(INT8_WIDTH - 1)     */
#define INT16_MIN    /*  - 2**(INT16_WIDTH - 1)    */
#define INT32_MIN    /*  - 2**(INT32_WIDTH - 1)    */
#define INT64_MIN    /*  - 2**(INT64_WIDTH - 1)    */

#define UINT8_MAX    /*  2**INT8_WIDTH - 1         */
#define UINT16_MAX   /*  2**INT16_WIDTH - 1        */
#define UINT32_MAX   /*  2**INT32_WIDTH - 1        */
#define UINT64_MAX   /*  2**INT64_WIDTH - 1        */

#define INT8_C(c)    c ## /* ... */
#define INT16_C(c)   c ## /* ... */
#define INT32_C(c)   c ## /* ... */
#define INT64_C(c)   c ## /* ... */

#define UINT8_C(c)   c ## /* ... */
#define UINT16_C(c)  c ## /* ... */
#define UINT32_C(c)  c ## /* ... */
#define UINT64_C(c)  c ## /* ... */
```

DESCRIPTION

*int*N*_t* are signed integer types of a fixed width of exactly N bits, *N* being the value specified in its type name. They are be capable of storing values in the range [**INT***N***_MIN**, **INT***N***_MAX**], substituting *N* by the appropriate number.

*uint*N*_t* are unsigned integer types of a fixed width of exactly N bits, N being the value specified in its type name. They are capable of storing values in the range [**0**, **UINT***N***_MAX**], substituting *N* by the appropriate number.

According to POSIX, [*u*]*int8_t*, [*u*]*int16_t*, and [*u*]*int32_t* are required; [*u*]*int64_t* are only required in implementations that provide integer types with width 64; and all other types of this form are optional.

The macros [**U**]**INT***N***_WIDTH** expand to the width in bits of these types (*N*).

The macros [**U**]**INT***N***_MAX** expand to the maximum value that these types can hold.

The macros **INT***N***_MIN** expand to the minimum value that these types can hold.

The macros [**U**]**INT***N***_C**() expand their argument to an integer constant of type [*u*]*int*N*_t*.

The length modifiers for the [*u*]*int*N*_t* types for the *printf*(3) family of functions are expanded by macros of the forms **PRId***N*, **PRIi***N*, **PRIu***N*, and **PRIx***N* (defined in <*inttypes.h*>); resulting for example in **%"PRId64"** or **%"PRIi64"** for printing *int64_t* values. The length modifiers for the [*u*]*int*N*_t* types for the *scanf*(3) family of functions are expanded by macros of the forms **SCNd***N*, **SCNi***N*, **SCNu***N*, and **SCNx***N*, (defined in <*inttypes.h*>); resulting for example in **%"SCNu8"** or **%"SCNx8"** for scanning *uint8_t* values.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

The [**U**]**INT***N***_WIDTH** macros were added in C23.

NOTES

The following header also provides these types: <*inttypes.h*>. <*arpa/inet.h*> also provides *uint16_t* and *uint32_t*.

SEE ALSO

intmax_t(3type), *intptr_t*(3type), *printf*(3)

NAME

intptr_t, uintptr_t − integer types wide enough to hold pointers

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stdint.h>

typedef /* ... */ intptr_t;
typedef /* ... */ uintptr_t;

#define INTPTR_WIDTH  /* ... */
#define UINTPTR_WIDTH INTPTR_WIDTH

#define INTPTR_MAX    /*  2**(INTPTR_WIDTH - 1) - 1  */
#define INTPTR_MIN    /*  - 2**(INTPTR_WIDTH - 1)    */
#define UINTPTR_MAX   /*  2**UINTPTR_WIDTH - 1       */
```

DESCRIPTION

intptr_t is a signed integer type such that any valid (*void **) value can be converted to this type and then converted back. It is capable of storing values in the range [**INTPTR_MIN**, **INTPTR_MAX**].

uintptr_t is an unsigned integer type such that any valid (*void **) value can be converted to this type and then converted back. It is capable of storing values in the range [**0**, **INTPTR_MAX**].

The macros [**U**]**INTPTR_WIDTH** expand to the width in bits of these types.

The macros [**U**]**INTPTR_MAX** expand to the maximum value that these types can hold.

The macro **INTPTR_MIN** expands to the minimum value that *intptr_t* can hold.

The length modifiers for the [*u*]*intptr_t* types for the *printf*(3) family of functions are expanded by the macros **PRIdPTR**, **PRIiPTR**, and **PRIuPTR** (defined in <*inttypes.h*>); resulting commonly in **%"PRIdPTR"** or **%"PRIiPTR"** for printing *intptr_t* values. The length modifiers for the *[*u*]*intptr_t types for the *scanf*(3) family of functions are expanded by the macros **SCNdPTR**, **SCNiPTR**, and **SCNuPTR** (defined in <*inttypes.h*>); resulting commonly in **%"SCNuPTR"** for scanning *uintptr_t* values.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C99, POSIX.1-2001.

NOTES

The following header also provides these types: <*inttypes.h*>.

SEE ALSO

intmax_t(3type), *void*(3)

NAME

iovec − Vector I/O data structure

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/uio.h>

struct iovec {
    void   *iov_base;  /* Starting address */
    size_t  iov_len;   /* Size of the memory pointed to by iov_base. */
};
```

DESCRIPTION

Describes a region of memory, beginning at *iov_base* address and with the size of *iov_len* bytes. System calls use arrays of this structure, where each element of the array represents a memory region, and the whole array represents a vector of memory regions. The maximum number of *iovec* structures in that array is limited by **IOV_MAX** (defined in <*limits.h*>, or accessible via the call *sysconf(_SC_IOV_MAX)*).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides this type: <*sys/socket.h*>.

SEE ALSO

process_madvise(2), *readv*(2)

NAME

itimerspec − interval for a timer with nanosecond precision

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <time.h>

struct itimerspec {
    struct timespec  it_interval;  /* Interval for periodic timer */
    struct timespec  it_value;     /* Initial expiration */
};
```

DESCRIPTION

Describes the initial expiration of a timer, and its interval, in seconds and nanoseconds.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

timerfd_create(2), *timer_settime*(2), *timespec*(3type)

NAME

lconv − numeric formatting information

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <locale.h>

struct lconv {                  /* Values in the "C" locale: */
    char *decimal_point;        /* "." */
    char *thousands_sep;        /* "" */
    char *grouping;             /* "" */
    char *mon_decimal_point;    /* "" */
    char *mon_thousands_sep;    /* "" */
    char *mon_grouping;         /* "" */
    char *positive_sign;        /* "" */
    char *negative_sign;        /* "" */
    char *currency_symbol;      /* "" */
    char  frac_digits;          /* CHAR_MAX */
    char  p_cs_precedes;        /* CHAR_MAX */
    char  n_cs_precedes;        /* CHAR_MAX */
    char  p_sep_by_space;       /* CHAR_MAX */
    char  n_sep_by_space;       /* CHAR_MAX */
    char  p_sign_posn;          /* CHAR_MAX */
    char  n_sign_posn;          /* CHAR_MAX */
    char *int_curr_symbol;      /* "" */
    char  int_frac_digits;      /* CHAR_MAX */
    char  int_p_cs_precedes;    /* CHAR_MAX */
    char  int_n_cs_precedes;    /* CHAR_MAX */
    char  int_p_sep_by_space;   /* CHAR_MAX */
    char  int_n_sep_by_space;   /* CHAR_MAX */
    char  int_p_sign_posn;      /* CHAR_MAX */
    char  int_n_sign_posn;      /* CHAR_MAX */
};
```

DESCRIPTION

Contains members related to the formatting of numeric values. In the "C" locale, its members have the values shown in the comments above.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

setlocale(3), *localeconv*(3), *charsets*(7), *locale*(7)

NAME

mode_t − file attributes

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ mode_t;
```

DESCRIPTION

Used for some file attributes (e.g., file mode). It is an integer type.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following headers also provide this type: <*fcntl.h*>, <*ndbm.h*>, <*spawn.h*>, <*sys/ipc.h*>, <*sys/mman.h*>, and <*sys/stat.h*>.

SEE ALSO

chmod(2), *mkdir*(2), *open*(2), *umask*(2), *stat*(3type)

NAME

off_t, off64_t, loff_t − file sizes

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ off_t;

#define _LARGEFILE64_SOURCE
#include <sys/types.h>

typedef /* ... */ off64_t;

#define _GNU_SOURCE
#include <sys/types.h>

typedef /* ... */ loff_t;
```

DESCRIPTION

off_t is used for describing file sizes. It is a signed integer type.

off64_t is a 64-bit version of the type, used in glibc.

loff_t is a 64-bit version of the type, introduced by the Linux kernel.

STANDARDS

off_t POSIX.1-2008.

off64_t GNU and some BSDs.

loff_t Linux.

VERSIONS

off_t POSIX.1-2001.

<*aio.h*> and <*stdio.h*> define *off_t* since POSIX.1-2008.

NOTES

On some architectures, the width of *off_t* can be controlled with the feature test macro **_FILE_OFFSET_BITS**.

The following headers also provide *off_t*: <*aio.h*>, <*fcntl.h*>, <*stdio.h*>, <*sys/mman.h*>, <*sys/stat.h*>, and <*unistd.h*>.

SEE ALSO

copy_file_range(2), *llseek*(2), *lseek*(2), *mmap*(2), *posix_fadvise*(2), *pread*(2), *readahead*(2), *sync_file_range*(2), *truncate*(2), *fseeko*(3), *lockf*(3), *lseek64*(3), *posix_fallocate*(3), *feature_test_macros*(7)

NAME

ptrdiff_t − count of elements or array index

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <stddef.h>

typedef /* ... */ **ptrdiff_t;**

DESCRIPTION

Used for a count of elements, or an array index. It is the result of subtracting two pointers. It is a signed integer type capable of storing values in the range [**PTRDIFF_MIN**, **PTRDIFF_MAX**].

The length modifier for *ptrdiff_t* for the *printf*(3) and the *scanf*(3) families of functions is **t**, resulting commonly in **%td** or **%ti** for printing *ptrdiff_t* values.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

SEE ALSO

size_t(3type)

NAME

sigevent, sigval − structure for notification from asynchronous routines

SYNOPSIS

```
#include <signal.h>

struct sigevent {
    int             sigev_notify;  /* Notification type */
    int             sigev_signo;   /* Signal number */
    union sigval    sigev_value;   /* Data passed with notification */

    void          (*sigev_notify_function)(union sigval);
                                   /* Notification function
                                      (SIGEV_THREAD) */
    pthread_attr_t *sigev_notify_attributes;
                                   /* Notification attributes */

    /* Linux only: */
    pid_t           sigev_notify_thread_id;
                                   /* ID of thread to signal
                                      (SIGEV_THREAD_ID) */
};

union sigval {                     /* Data passed with notification */
    int             sival_int;     /* Integer value */
    void           *sival_ptr;     /* Pointer value */
};
```

DESCRIPTION

sigevent

The *sigevent* structure is used by various APIs to describe the way a process is to be notified about an event (e.g., completion of an asynchronous request, expiration of a timer, or the arrival of a message).

The definition shown in the SYNOPSIS is approximate: some of the fields in the *sigevent* structure may be defined as part of a union. Programs should employ only those fields relevant to the value specified in *sigev_notify*.

The *sigev_notify* field specifies how notification is to be performed. This field can have one of the following values:

SIGEV_NONE
A "null" notification: don't do anything when the event occurs.

SIGEV_SIGNAL
Notify the process by sending the signal specified in *sigev_signo*.

If the signal is caught with a signal handler that was registered using the *sigaction*(2) **SA_SIGINFO** flag, then the following fields are set in the *siginfo_t* structure that is passed as the second argument of the handler:

si_code This field is set to a value that depends on the API delivering the notification.

si_signo This field is set to the signal number (i.e., the same value as in *sigev_signo*).

si_value This field is set to the value specified in *sigev_value*.

Depending on the API, other fields may also be set in the *siginfo_t* structure.

The same information is also available if the signal is accepted using *sigwaitinfo*(2).

SIGEV_THREAD
Notify the process by invoking *sigev_notify_function* "as if" it were the start function of a new thread. (Among the implementation possibilities here are that each timer notification could result in the creation of a new thread, or that a single thread is created to receive all notifications.) The function is invoked with *sigev_value* as its sole argument. If *sigev_notify_attributes* is not NULL, it should point to a *pthread_attr_t* structure that defines attributes for the

new thread (see *pthread_attr_init*(3)).

SIGEV_THREAD_ID (Linux-specific)
Currently used only by POSIX timers; see *timer_create*(2).

sigval
Data passed with a signal.

STANDARDS
POSIX.1-2008.

HISTORY
POSIX.1-2001.

<*aio.h*> and <*time.h*> define *sigevent* since POSIX.1-2008.

NOTES
The following headers also provide *sigevent*: <*aio.h*>, <*mqueue.h*>, and <*time.h*>.

SEE ALSO
timer_create(2), *getaddrinfo_a*(3), *lio_listio*(3), *mq_notify*(3), *pthread_sigqueue*(3), *sigqueue*(3), *aiocb*(3type), *siginfo_t*(3type)

NAME

size_t, ssize_t − count of bytes

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <stddef.h>

typedef /* ... */ size_t;

#include <sys/types.h>

typedef /* ... */ ssize_t;
```

DESCRIPTION

size_t Used for a count of bytes. It is the result of the *sizeof*() operator. It is an unsigned integer type capable of storing values in the range [**0**, **SIZE_MAX**].

ssize_t Used for a count of bytes or an error indication. It is a signed integer type capable of storing values at least in the range [**−1**, **SSIZE_MAX**].

Use with printf(3) and scanf(3)

size_t The length modifier for *size_t* for the *printf*(3) and the *scanf*(3) families of functions is **z**, resulting commonly in **%zu** or **%zx** for printing *size_t* values.

ssize_t glibc and most other implementations provide a length modifier for *ssize_t* for the *printf*(3) and the *scanf*(3) families of functions, which is **z**; resulting commonly in **%zd** or **%zi** for printing *ssize_t* values. Although **z** works for *ssize_t* on most implementations, portable POSIX programs should avoid using it—for example, by converting the value to *intmax_t* and using its length modifier (**j**).

STANDARDS

size_t C11, POSIX.1-2008.

ssize_t POSIX.1-2008.

HISTORY

size_t C89, POSIX.1-2001.

ssize_t POSIX.1-2001.

<*aio.h*>, <*glob.h*>, <*grp.h*>, <*iconv.h*>, <*mqueue.h*>, <*pwd.h*>, <*signal.h*>, and <*sys/socket.h*> define *size_t* since POSIX.1-2008.

<*aio.h*>, <*mqueue.h*>, and <*sys/socket.h*> define *ssize_t* since POSIX.1-2008.

NOTES

size_t The following headers also provide *size_t*: <*aio.h*>, <*glob.h*>, <*grp.h*>, <*iconv.h*>, <*monetary.h*>, <*mqueue.h*>, <*ndbm.h*>, <*pwd.h*>, <*regex.h*>, <*search.h*>, <*signal.h*>, <*stdio.h*>, <*stdlib.h*>, <*string.h*>, <*strings.h*>, <*sys/mman.h*>, <*sys/msg.h*>, <*sys/sem.h*>, <*sys/shm.h*>, <*sys/socket.h*>, <*sys/types.h*>, <*sys/uio.h*>, <*time.h*>, <*unistd.h*>, <*wchar.h*>, and <*wordexp.h*>.

ssize_t The following headers also provide *ssize_t*: <*aio.h*>, <*monetary.h*>, <*mqueue.h*>, <*stdio.h*>, <*sys/msg.h*>, <*sys/socket.h*>, <*sys/uio.h*>, and <*unistd.h*>.

SEE ALSO

read(2), *readlink*(2), *readv*(2), *recv*(2), *send*(2), *write*(2), *fread*(3), *fwrite*(3), *memcmp*(3), *memcpy*(3), *memset*(3), *offsetof*(3), *ptrdiff_t*(3type)

NAME

sockaddr, sockaddr_storage, sockaddr_in, sockaddr_in6, sockaddr_un, socklen_t, in_addr, in6_addr, in_addr_t, in_port_t, − socket address

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/socket.h>

struct sockaddr {
    sa_family_t     sa_family;      /* Address family */
    char            sa_data[];      /* Socket address */
};

struct sockaddr_storage {
    sa_family_t     ss_family;      /* Address family */
};

typedef /* ... */ socklen_t;
typedef /* ... */ sa_family_t;
```

Internet domain sockets

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t     sin_family;     /* AF_INET */
    in_port_t       sin_port;       /* Port number */
    struct in_addr  sin_addr;       /* IPv4 address */
};

struct sockaddr_in6 {
    sa_family_t     sin6_family;    /* AF_INET6 */
    in_port_t       sin6_port;      /* Port number */
    uint32_t        sin6_flowinfo;  /* IPv6 flow info */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t        sin6_scope_id;  /* Set of interfaces for a scope */
};

struct in_addr {
    in_addr_t s_addr;
};

struct in6_addr {
    uint8_t   s6_addr[16];
};

typedef uint32_t in_addr_t;
typedef uint16_t in_port_t;
```

UNIX domain sockets

```
#include <sys/un.h>

struct sockaddr_un {
    sa_family_t     sun_family;     /* Address family */
    char            sun_path[];     /* Socket pathname */
};
```

DESCRIPTION

sockaddr
: Describes a socket address.

sockaddr_storage
: A structure at least as large as any other *sockaddr_** address structures. It's aligned so that a pointer to it can be cast as a pointer to other *sockaddr_** structures and used to access its fields.

socklen_t
: Describes the length of a socket address. This is an integer type of at least 32 bits.

sa_family_t
: Describes a socket's protocol family. This is an unsigned integer type.

Internet domain sockets

sockaddr_in
: Describes an IPv4 Internet domain socket address. The *sin_port* and *sin_addr* members are stored in network byte order.

sockaddr_in6
: Describes an IPv6 Internet domain socket address. The *sin6_addr.s6_addr* array is used to contain a 128-bit IPv6 address, stored in network byte order.

UNIX domain sockets

sockaddr_un
: Describes a UNIX domain socket address.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

socklen_t was invented by POSIX. See also *accept*(2).

These structures were invented before modern ISO C strict-aliasing rules. If aliasing rules are applied strictly, these structures would be extremely difficult to use without invoking Undefined Behavior. POSIX Issue 8 will fix this by requiring that implementations make sure that these structures can be safely used as they were designed.

NOTES

socklen_t is also defined in <*netdb.h*>.

sa_family_t is also defined in <*netinet/in.h*> and <*sys/un.h*>.

SEE ALSO

accept(2), *bind*(2), *connect*(2), *getpeername*(2), *getsockname*(2), *getsockopt*(2), *sendto*(2), *setsockopt*(2), *socket*(2), *socketpair*(2), *getaddrinfo*(3), *gethostbyaddr*(3), *getnameinfo*(3), *htonl*(3), *ipv6*(7), *socket*(7)

NAME

stat − file status

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/stat.h>

struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* Inode number */
    mode_t     st_mode;     /* File type and mode */
    nlink_t    st_nlink;    /* Number of hard links */
    uid_t      st_uid;      /* User ID of owner */
    gid_t      st_gid;      /* Group ID of owner */
    dev_t      st_rdev;     /* Device ID (if special file) */
    off_t      st_size;     /* Total size, in bytes */
    blksize_t  st_blksize;  /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;   /* Number of 512 B blocks allocated */

    /* Since POSIX.1-2008, this structure supports nanosecond
       precision for the following timestamp fields.
       For the details before POSIX.1-2008, see VERSIONS. */

    struct timespec  st_atim;  /* Time of last access */
    struct timespec  st_mtim;  /* Time of last modification */
    struct timespec  st_ctim;  /* Time of last status change */

#define st_atime  st_atim.tv_sec  /* Backward compatibility */
#define st_mtime  st_mtim.tv_sec
#define st_ctime  st_ctim.tv_sec
};
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

st_atim, *st_mtim*, *st_ctim*:
Since glibc 2.12:
_POSIX_C_SOURCE >= 200809L || _XOPEN_SOURCE >= 700
glibc 2.19 and earlier:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

Describes information about a file.

The fields are as follows:

st_dev This field describes the device on which this file resides. (The *major*(3) and *minor*(3) macros may be useful to decompose the device ID in this field.)

st_ino This field contains the file's inode number.

st_mode
This field contains the file type and mode. See *inode*(7) for further information.

st_nlink
This field contains the number of hard links to the file.

st_uid This field contains the user ID of the owner of the file.

st_gid This field contains the ID of the group owner of the file.

st_rdev This field describes the device that this file (inode) represents.

st_size This field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

st_blksize
: This field gives the "preferred" block size for efficient filesystem I/O.

st_blocks
: This field indicates the number of blocks allocated to the file, in 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

st_atime
: This is the time of the last access of file data.

st_mtime
: This is the time of last modification of file data.

st_ctime
: This is the file's last status change timestamp (time of last change to the inode).

For further information on the above fields, see *inode*(7).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

Old kernels and old standards did not support nanosecond timestamp fields. Instead, there were three timestamp fields—*st_atime*, *st_mtime*, and *st_ctime*—typed as *time_t* that recorded timestamps with one-second precision.

Since Linux 2.5.48, the *stat* structure supports nanosecond resolution for the three file timestamp fields. The nanosecond components of each timestamp are available via names of the form *st_atim.tv_nsec*, if suitable test macros are defined. Nanosecond timestamps were standardized in POSIX.1-2008, and, starting with glibc 2.12, glibc exposes the nanosecond component names if **_POSIX_C_SOURCE** is defined with the value 200809L or greater, or **_XOPEN_SOURCE** is defined with the value 700 or greater. Up to and including glibc 2.19, the definitions of the nanoseconds components are also defined if **_BSD_SOURCE** or **_SVID_SOURCE** is defined. If none of the aforementioned macros are defined, then the nanosecond values are exposed with names of the form *st_atimensec*.

NOTES

The following header also provides this type: <*ftw.h*>.

SEE ALSO

stat(2), *inode*(7)

NAME

time_t, suseconds_t, useconds_t − integer time

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <time.h>

typedef /* ... */  time_t;

#include <sys/types.h>

typedef /* ... */  suseconds_t;
typedef /* ... */  useconds_t;
```

DESCRIPTION

time_t Used for time in seconds. According to POSIX, it is an integer type.

suseconds_t
: Used for time in microseconds. It is a signed integer type capable of storing values at least in the range [**−1**, **1000000**].

useconds_t
: Used for time in microseconds. It is an unsigned integer type capable of storing values at least in the range [**0**, **1000000**].

STANDARDS

time_t C11, POSIX.1-2008.

suseconds_t
useconds_t
: POSIX.1-2008.

HISTORY

time_t C89, POSIX.1-2001.

suseconds_t
useconds_t
: POSIX.1-2001.

<*sched.h*> defines *time_t* since POSIX.1-2008.

POSIX.1-2001 defined *useconds_t* in <*unistd.h*> too.

NOTES

On some architectures, the width of *time_t* can be controlled with the feature test macro **_TIME_BITS**. See *feature_test_macros*(7).

The following headers also provide *time_t*: <*sched.h*>, <*sys/msg.h*>, <*sys/select.h*>, <*sys/sem.h*>, <*sys/shm.h*>, <*sys/stat.h*>, <*sys/time.h*>, <*sys/types.h*>, and <*utime.h*>.

The following headers also provide *suseconds_t*: <*sys/select.h*> and <*sys/time.h*>.

SEE ALSO

stime(2), *time*(2), *ctime*(3), *difftime*(3), *usleep*(3), *timeval*(3type)

NAME

timer_t − timer ID

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ timer_t;
```

DESCRIPTION

Used for timer ID returned by *timer_create*(2).

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following header also provides *timer_t*: <*time.h*>.

SEE ALSO

timer_create(2), *timer_delete*(2), *timer_getoverrun*(2), *timer_settime*(2)

NAME

timespec − time in seconds and nanoseconds

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <time.h>

struct timespec {
    time_t     tv_sec;   /* Seconds */
    /* ... */  tv_nsec;  /* Nanoseconds [0, 999'999'999] */
};
```

DESCRIPTION

Describes times in seconds and nanoseconds.

tv_nsec is of an implementation-defined signed type capable of holding the specified range. Under glibc, this is usually *long*, and *long long* on X32. It can be safely down-cast to any concrete 32-bit integer type for processing.

VERSIONS

Prior to C23, *tv_nsec* was *long*.

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following headers also provide this type: <*aio.h*>, <*mqueue.h*>, <*sched.h*>, <*signal.h*>, <*sys/select.h*>, and <*sys/stat.h*>.

SEE ALSO

clock_gettime(2), *clock_nanosleep*(2), *nanosleep*(2), *timerfd_gettime*(2), *timer_gettime*(2), *time_t*(3type), *timeval*(3type)

NAME

timeval − time in seconds and microseconds

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <sys/time.h>

struct timeval {
    time_t       tv_sec;   /* Seconds */
    suseconds_t  tv_usec;  /* Microseconds */
};
```

DESCRIPTION

Describes times in seconds and microseconds.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

NOTES

The following headers also provide this type: <*sys/resource.h*>, <*sys/select.h*>, and <*utmpx.h*>.

SEE ALSO

gettimeofday(2), *select*(2), *utimes*(2), *adjtime*(3), *futimes*(3), *timeradd*(3), *suseconds_t*(3type), *time_t*(3type), *timespec*(3type)

NAME

tm − broken-down time

LIBRARY

Standard C library (*libc*)

SYNOPSIS

```
#include <time.h>

struct tm {
    int         tm_sec;    /* Seconds          [0, 60] */
    int         tm_min;    /* Minutes          [0, 59] */
    int         tm_hour;   /* Hour             [0, 23] */
    int         tm_mday;   /* Day of the month [1, 31] */
    int         tm_mon;    /* Month            [0, 11]  (January = 0) */
    int         tm_year;   /* Year minus 1900 */
    int         tm_wday;   /* Day of the week  [0, 6]   (Sunday = 0) */
    int         tm_yday;   /* Day of the year  [0, 365] (Jan/01 = 0) */
    int         tm_isdst;  /* Daylight savings flag */

    long        tm_gmtoff; /* Seconds East of UTC */
    const char *tm_zone;   /* Timezone abbreviation */
};
```

Feature Test Macro Requirements for glibc (see *feature_test_macros*(7)):

tm_gmtoff , *tm_zone*:
Since glibc 2.20:
_DEFAULT_SOURCE
glibc 2.20 and earlier:
_BSD_SOURCE

DESCRIPTION

Describes time, broken down into distinct components.

tm_isdst describes whether daylight saving time is in effect at the time described. The value is positive if daylight saving time is in effect, zero if it is not, and negative if the information is not available.

tm_gmtoff is the difference, in seconds, of the timezone represented by this broken-down time and UTC (this is the additive inverse of *timezone*(3)).

tm_zone is the equivalent of *tzname*(3) for the timezone represented by this broken-down time.

VERSIONS

In C90, *tm_sec* could represent values in the range [**0**, **61**], which could represent a double leap second. UTC doesn't permit double leap seconds, so it was limited to **60** in C99.

timezone(3), as a variable, is an XSI extension: some systems provide the V7-compatible *timezone*(3) function. The *tm_gmtoff* field provides an alternative (with the opposite sign) for those systems.

tm_zone points to static storage and may be overridden on subsequent calls to *localtime*(3) and similar functions (however, this never happens under glibc).

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

tm_gmtoff and *tm_zone* originate from 4.3BSD-Tahoe (where *tm_zone* is a *char* *).

NOTES

tm_sec can represent a leap second with the value **60**.

SEE ALSO

ctime(3), *strftime*(3), *strptime*(3), *time*(7)

NAME

va_list − variable argument list

LIBRARY

Standard C library (*libc*)

SYNOPSIS

#include <stdarg.h>

typedef /* ... */ **va_list;**

DESCRIPTION

Used by functions with a varying number of arguments of varying types. The function must declare an object of type *va_list* which is used by the macros *va_start*(3), *va_arg*(3), *va_copy*(3), and *va_end*(3) to traverse the list of arguments.

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

NOTES

The following headers also provide *va_list*: <*stdio.h*> and <*wchar.h*>.

SEE ALSO

va_start(3), *va_arg*(3), *va_copy*(3), *va_end*(3)

NAME

void − abstract type

SYNOPSIS

void *

DESCRIPTION

A pointer to any object type may be converted to a pointer to *void* and back. POSIX further requires that any pointer, including pointers to functions, may be converted to a pointer to *void* and back.

Conversions from and to any other pointer type are done implicitly, not requiring casts at all. Note that this feature prevents any kind of type checking: the programmer should be careful not to convert a *void* * value to a type incompatible to that of the underlying data, because that would result in undefined behavior.

This type is useful in function parameters and return value to allow passing values of any type. The function will typically use some mechanism to know the real type of the data being passed via a pointer to *void*.

A value of this type can't be dereferenced, as it would give a value of type *void*, which is not possible. Likewise, pointer arithmetic is not possible with this type. However, in GNU C, pointer arithmetic is allowed as an extension to the standard; this is done by treating the size of a *void* or of a function as 1. A consequence of this is that *sizeof* is also allowed on *void* and on function types, and returns 1.

Use with printf(3) and scanf(3)

The conversion specifier for *void* * for the *printf*(3) and the *scanf*(3) families of functions is **p**.

VERSIONS

The POSIX requirement about compatibility between *void* * and function pointers was added in POSIX.1-2008 Technical Corrigendum 1 (2013).

STANDARDS

C11, POSIX.1-2008.

HISTORY

C89, POSIX.1-2001.

SEE ALSO

malloc(3), *memcmp*(3), *memcpy*(3), *memset*(3), *intptr_t*(3type)

NAME

intro − introduction to special files

DESCRIPTION

Section 4 of the manual describes special files (devices).

FILES

/dev/* — device files

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

SEE ALSO

mknod(1), *mknod*(2), *standards*(7)

NAME

cciss − HP Smart Array block driver

SYNOPSIS

modprobe cciss [cciss_allow_hpsa=1]

DESCRIPTION

Note: This obsolete driver was removed in Linux 4.14, as it is superseded by the *hpsa*(4) driver in newer kernels.

cciss is a block driver for older HP Smart Array RAID controllers.

Options

cciss_allow_hpsa=1: This option prevents the **cciss** driver from attempting to drive any controllers that the *hpsa*(4) driver is capable of controlling, which is to say, the **cciss** driver is restricted by this option to the following controllers:

Smart Array 5300
Smart Array 5i
Smart Array 532
Smart Array 5312
Smart Array 641
Smart Array 642
Smart Array 6400
Smart Array 6400 EM
Smart Array 6i
Smart Array P600
Smart Array P400i
Smart Array E200i
Smart Array E200
Smart Array E200i
Smart Array E200i
Smart Array E200i
Smart Array E500

Supported hardware

The **cciss** driver supports the following Smart Array boards:

Smart Array 5300
Smart Array 5i
Smart Array 532
Smart Array 5312
Smart Array 641
Smart Array 642
Smart Array 6400
Smart Array 6400 U320 Expansion Module
Smart Array 6i
Smart Array P600
Smart Array P800
Smart Array E400
Smart Array P400i
Smart Array E200
Smart Array E200i
Smart Array E500
Smart Array P700m
Smart Array P212
Smart Array P410
Smart Array P410i
Smart Array P411
Smart Array P812
Smart Array P712m
Smart Array P711m

Configuration details

To configure HP Smart Array controllers, use the HP Array Configuration Utility (either *hpacuxe*(8) or *hpacucli*(8)) or the Offline ROM-based Configuration Utility (ORCA) run from the Smart Array's option ROM at boot time.

FILES

Device nodes

The device naming scheme is as follows:

Major numbers:

104 cciss0
105 cciss1
106 cciss2
105 cciss3
108 cciss4
109 cciss5
110 cciss6
111 cciss7

Minor numbers:

```
b7 b6 b5 b4 b3 b2 b1 b0
|----+----| |----+----|
     |           |
     |           +-------- Partition ID (0=wholedev, 1-15 partition)
     |
     +-------------------- Logical Volume number
```

The device naming scheme is:

/dev/cciss/c0d0 Controller 0, disk 0, whole device
/dev/cciss/c0d0p1 Controller 0, disk 0, partition 1
/dev/cciss/c0d0p2 Controller 0, disk 0, partition 2
/dev/cciss/c0d0p3 Controller 0, disk 0, partition 3

/dev/cciss/c1d1 Controller 1, disk 1, whole device
/dev/cciss/c1d1p1 Controller 1, disk 1, partition 1
/dev/cciss/c1d1p2 Controller 1, disk 1, partition 2
/dev/cciss/c1d1p3 Controller 1, disk 1, partition 3

Files in /proc

The files */proc/driver/cciss/cciss[0-9]+* contain information about the configuration of each controller. For example:

```
$ cd /proc/driver/cciss
$ ls -l
total 0
-rw-r--r-- 1 root root 0 2010-09-10 10:38 cciss0
-rw-r--r-- 1 root root 0 2010-09-10 10:38 cciss1
-rw-r--r-- 1 root root 0 2010-09-10 10:38 cciss2
$ cat cciss2
cciss2: HP Smart Array P800 Controller
Board ID: 0x3223103c
Firmware Version: 7.14
IRQ: 16
Logical drives: 1
Current Q depth: 0
Current # commands on controller: 0
Max Q depth since init: 1
Max # commands on controller since init: 2
Max SG entries since init: 32
Sequential access devices: 0
```

```
cciss/c2d0:   36.38GB       RAID 0
```

Files in /sys

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/model*
Displays the SCSI INQUIRY page 0 model for logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/rev*
Displays the SCSI INQUIRY page 0 revision for logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/unique_id*
Displays the SCSI INQUIRY page 83 serial number for logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/vendor*
Displays the SCSI INQUIRY page 0 vendor for logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/block:cciss!cXdY*
A symbolic link to */sys/block/cciss!cXdY*.

*/sys/bus/pci/devices/*dev*/ccissX/rescan*
When this file is written to, the driver rescans the controller to discover any new, removed, or modified logical drives.

*/sys/bus/pci/devices/*dev*/ccissX/resettable*
A value of 1 displayed in this file indicates that the "reset_devices=1" kernel parameter (used by **kdump**) is honored by this controller. A value of 0 indicates that the "reset_devices=1" kernel parameter will not be honored. Some models of Smart Array are not able to honor this parameter.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/lunid*
Displays the 8-byte LUN ID used to address logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/raid_level*
Displays the RAID level of logical drive *Y* of controller *X*.

*/sys/bus/pci/devices/*dev*/ccissX/cXdY/usage_count*
Displays the usage count (number of opens) of logical drive *Y* of controller *X*.

SCSI tape drive and medium changer support

SCSI sequential access devices and medium changer devices are supported and appropriate device nodes are automatically created (e.g., */dev/st0*, */dev/st1*, etc.; see *st*(4) for more details.) You must enable "SCSI tape drive support for Smart Array 5xxx" and "SCSI support" in your kernel configuration to be able to use SCSI tape drives with your Smart Array 5xxx controller.

Additionally, note that the driver will not engage the SCSI core at init time. The driver must be directed to dynamically engage the SCSI core via the */proc* filesystem entry, which the "block" side of the driver creates as */proc/driver/cciss/cciss** at run time. This is because at driver init time, the SCSI core may not yet be initialized (because the driver is a block driver) and attempting to register it with the SCSI core in such a case would cause a hang. This is best done via an initialization script (typically in */etc/init.d*, but could vary depending on distribution). For example:

```
for x in /proc/driver/cciss/cciss[0-9]*
do
    echo "engage scsi" > $x
done
```

Once the SCSI core is engaged by the driver, it cannot be disengaged (except by unloading the driver, if it happens to be linked as a module.)

Note also that if no sequential access devices or medium changers are detected, the SCSI core will not be engaged by the action of the above script.

Hot plug support for SCSI tape drives

Hot plugging of SCSI tape drives is supported, with some caveats. The **cciss** driver must be informed that changes to the SCSI bus have been made. This may be done via the */proc* filesystem. For example:

```
echo "rescan" > /proc/scsi/cciss0/1
```

This causes the driver to:

(1) query the adapter about changes to the physical SCSI buses and/or fiber channel arbitrated loop, and

(2) make note of any new or removed sequential access devices or medium changers.

The driver will output messages indicating which devices have been added or removed and the controller, bus, target, and lun used to address each device. The driver then notifies the SCSI midlayer of these changes.

Note that the naming convention of the */proc* filesystem entries contains a number in addition to the driver name (e.g., "cciss0" instead of just "cciss", which you might expect).

Note: *Only* sequential access devices and medium changers are presented as SCSI devices to the SCSI midlayer by the **cciss** driver. Specifically, physical SCSI disk drives are *not* presented to the SCSI midlayer. The only disk devices that are presented to the kernel are logical drives that the array controller constructs from regions on the physical drives. The logical drives are presented to the block layer (not to the SCSI midlayer). It is important for the driver to prevent the kernel from accessing the physical drives directly, since these drives are used by the array controller to construct the logical drives.

SCSI error handling for tape drives and medium changers

The Linux SCSI midlayer provides an error-handling protocol that is initiated whenever a SCSI command fails to complete within a certain amount of time (which can vary depending on the command). The **cciss** driver participates in this protocol to some extent. The normal protocol is a four-step process:

(1) First, the device is told to abort the command.

(2) If that doesn't work, the device is reset.

(3) If that doesn't work, the SCSI bus is reset.

(4) If that doesn't work, the host bus adapter is reset.

The **cciss** driver is a block driver as well as a SCSI driver and only the tape drives and medium changers are presented to the SCSI midlayer. Furthermore, unlike more straightforward SCSI drivers, disk I/O continues through the block side during the SCSI error-recovery process. Therefore, the **cciss** driver implements only the first two of these actions, aborting the command, and resetting the device. Note also that most tape drives will not oblige in aborting commands, and sometimes it appears they will not even obey a reset command, though in most circumstances they will. If the command cannot be aborted and the device cannot be reset, the device will be set offline.

In the event that the error-handling code is triggered and a tape drive is successfully reset or the tardy command is successfully aborted, the tape drive may still not allow I/O to continue until some command is issued that positions the tape to a known position. Typically you must rewind the tape (by issuing *mt −f /dev/st0 rewind* for example) before I/O can proceed again to a tape drive that was reset.

SEE ALSO

hpsa(4), *cciss_vol_status*(8), *hpacucli*(8), *hpacuxe*(8)

, and *Documentation/blockdev/cciss.txt* and *Documentation/ABI/testing/sysfs−bus−pci−devices−cciss* in the Linux kernel source tree

NAME

console_codes − Linux console escape and control sequences

DESCRIPTION

The Linux console implements a large subset of the VT102 and ECMA-48 / ISO/IEC 6429 / ANSI X3.64 terminal controls, plus certain private-mode sequences for changing the color palette, character-set mapping, and so on. In the tabular descriptions below, the second column gives ECMA-48 or DEC mnemonics (the latter if prefixed with DEC) for the given function. Sequences without a mnemonic are neither ECMA-48 nor VT102.

After all the normal output processing has been done, and a stream of characters arrives at the console driver for actual printing, the first thing that happens is a translation from the code used for processing to the code used for printing.

If the console is in UTF-8 mode, then the incoming bytes are first assembled into 16-bit Unicode codes. Otherwise, each byte is transformed according to the current mapping table (which translates it to a Unicode value). See the **Character Sets** section below for discussion.

In the normal case, the Unicode value is converted to a font index, and this is stored in video memory, so that the corresponding glyph (as found in video ROM) appears on the screen. Note that the use of Unicode (and the design of the PC hardware) allows us to use 512 different glyphs simultaneously.

If the current Unicode value is a control character, or we are currently processing an escape sequence, the value will treated specially. Instead of being turned into a font index and rendered as a glyph, it may trigger cursor movement or other control functions. See the **Linux Console Controls** section below for discussion.

It is generally not good practice to hard-wire terminal controls into programs. Linux supports a *terminfo*(5) database of terminal capabilities. Rather than emitting console escape sequences by hand, you will almost always want to use a terminfo-aware screen library or utility such as *ncurses*(3), *tput*(1), or *reset*(1)

Linux console controls

This section describes all the control characters and escape sequences that invoke special functions (i.e., anything other than writing a glyph at the current cursor location) on the Linux console.

Control characters

A character is a control character if (before transformation according to the mapping table) it has one of the 14 codes 00 (NUL), 07 (BEL), 08 (BS), 09 (HT), 0a (LF), 0b (VT), 0c (FF), 0d (CR), 0e (SO), 0f (SI), 18 (CAN), 1a (SUB), 1b (ESC), 7f (DEL). One can set a "display control characters" mode (see below), and allow 07, 09, 0b, 18, 1a, 7f to be displayed as glyphs. On the other hand, in UTF-8 mode all codes 00–1f are regarded as control characters, regardless of any "display control characters" mode.

If we have a control character, it is acted upon immediately and then discarded (even in the middle of an escape sequence) and the escape sequence continues with the next character. (However, ESC starts a new escape sequence, possibly aborting a previous unfinished one, and CAN and SUB abort any escape sequence.) The recognized control characters are BEL, BS, HT, LF, VT, FF, CR, SO, SI, CAN, SUB, ESC, DEL, CSI. They do what one would expect:

BEL (0x07, ^**G**)
: beeps;

BS (0x08, ^**H**)
: backspaces one column (but not past the beginning of the line);

HT (0x09, ^**I**)
: goes to the next tab stop or to the end of the line if there is no earlier tab stop;

LF (0x0A, ^**J**)
VT (0x0B, ^**K**)
FF (0x0C, ^**L**)
: all give a linefeed, and if LF/NL (new-line mode) is set also a carriage return;

CR (0x0D, ^**M**)
: gives a carriage return;

SO (0x0E, ^**N**)
: activates the G1 character set;

SI (0x0F, ^**O**)
: activates the G0 character set;

CAN (0x18, ^**X**)
SUB (0x1A, ^**Z**)
: abort escape sequences;

ESC (0x1B, ^**[**)
: starts an escape sequence;

DEL (0x7F)
: is ignored;

CSI (0x9B)
: is equivalent to ESC [.

ESC- but not CSI-sequences

ESC c	RIS	Reset.
ESC D	IND	Linefeed.
ESC E	NEL	Newline.
ESC H	HTS	Set tab stop at current column.
ESC M	RI	Reverse linefeed.
ESC Z	DECID	DEC private identification. The kernel returns the string ESC [? 6 c, claiming that it is a VT102.
ESC 7	DECSC	Save current state (cursor coordinates, attributes, character sets pointed at by G0, G1).
ESC 8	DECRC	Restore state most recently saved by ESC 7.
ESC %		Start sequence selecting character set
ESC % @		Select default (ISO/IEC 646 / ISO/IEC 8859-1)
ESC % G		Select UTF-8
ESC % 8		Select UTF-8 (obsolete)
ESC # 8	DECALN	DEC screen alignment test − fill screen with E's.
ESC (		Start sequence defining G0 character set (followed by one of B, 0, U, K, as below)
ESC (B		Select default (ISO/IEC 8859-1 mapping).
ESC (0		Select VT100 graphics mapping.
ESC (U		Select null mapping − straight to character ROM.
ESC (K		Select user mapping − the map that is loaded by the utility*mapscrn*(8).
ESC)		Start sequence defining G1 (followed by one of B, 0, U, K, as above).
ESC >	DECPNM	Set numeric keypad mode
ESC =	DECPAM	Set application keypad mode
ESC]	OSC	Operating System Command prefix.
ESC] R		Reset palette.
ESC] P		Set palette, with parameter given in 7 hexadecimal digits *nrrggbb* after the final P. Here *n* is the color (0–15), and *rrggbb* indicates the red/green/blue values (0–255).

ECMA-48 CSI sequences

CSI (or ESC [) is followed by a sequence of parameters, at most NPAR (16), that are decimal numbers separated by semicolons. An empty or absent parameter is taken to be 0. The sequence of parameters may be preceded by a single question mark.

However, after CSI [(or ESC [[) a single character is read and this entire sequence is ignored. (The idea is to ignore an echoed function key.)

The action of a CSI sequence is determined by its final character.

@	ICH	Insert the indicated # of blank characters.
A	CUU	Move cursor up the indicated # of rows.
B	CUD	Move cursor down the indicated # of rows.
C	CUF	Move cursor right the indicated # of columns.

D	CUB	Move cursor left the indicated # of columns.
E	CNL	Move cursor down the indicated # of rows, to column 1.
F	CPL	Move cursor up the indicated # of rows, to column 1.
G	CHA	Move cursor to indicated column in current row.
H	CUP	Move cursor to the indicated row, column (origin at 1,1).
J	ED	Erase display (default: from cursor to end of display).
		ESC [1 J: erase from start to cursor.
		ESC [2 J: erase whole display.
		ESC [3 J: erase whole display including scroll-back buffer (since Linux 3.0).
K	EL	Erase line (default: from cursor to end of line).
		ESC [1 K: erase from start of line to cursor.
		ESC [2 K: erase whole line.
L	IL	Insert the indicated # of blank lines.
M	DL	Delete the indicated # of lines.
P	DCH	Delete the indicated # of characters on current line.
X	ECH	Erase the indicated # of characters on current line.
a	HPR	Move cursor right the indicated # of columns.
c	DA	Answer ESC [? 6 c: "I am a VT102".
d	VPA	Move cursor to the indicated row, current column.
e	VPR	Move cursor down the indicated # of rows.
f	HVP	Move cursor to the indicated row, column.
g	TBC	Without parameter: clear tab stop at current position.
		ESC [3 g: delete all tab stops.
h	SM	Set Mode (see below).
l	RM	Reset Mode (see below).
m	SGR	Set attributes (see below).
n	DSR	Status report (see below).
q	DECLL	Set keyboard LEDs.
		ESC [0 q: clear all LEDs
		ESC [1 q: set Scroll Lock LED
		ESC [2 q: set Num Lock LED
		ESC [3 q: set Caps Lock LED
r	DECSTBM	Set scrolling region; parameters are top and bottom row.
s	?	Save cursor location.
u	?	Restore cursor location.
`	HPA	Move cursor to indicated column in current row.

ECMA-48 Select Graphic Rendition

The ECMA-48 SGR sequence ESC [*parameters* m sets display attributes. Several attributes can be set in the same sequence, separated by semicolons. An empty parameter (between semicolons or string initiator or terminator) is interpreted as a zero.

param	result
0	reset all attributes to their defaults
1	set bold
2	set half-bright (simulated with color on a color display)
3	set italic (since Linux 2.6.22; simulated with color on a color display)
4	set underscore (simulated with color on a color display) (the colors used to simulate dim or underline are set using ESC] ...)
5	set blink
7	set reverse video
10	reset selected mapping, display control flag, and toggle meta flag (ECMA-48 says "primary font").
11	select null mapping, set display control flag, reset toggle meta flag (ECMA-48 says "first alternate font").
12	select null mapping, set display control flag, set toggle meta flag (ECMA-48 says "second alternate font"). The toggle meta flag causes the high bit of a byte to be toggled before the mapping table translation is done.

21 set underline; before Linux 4.17, this value set normal intensity (as is done in many other terminals)
22 set normal intensity
23 italic off (since Linux 2.6.22)
24 underline off
25 blink off
27 reverse video off
30 set black foreground
31 set red foreground
32 set green foreground
33 set brown foreground
34 set blue foreground
35 set magenta foreground
36 set cyan foreground
37 set white foreground
38 256/24-bit foreground color follows, shoehorned into 16 basic colors (before Linux 3.16: set underscore on, set default foreground color)
39 set default foreground color (before Linux 3.16: set underscore off, set default foreground color)
40 set black background
41 set red background
42 set green background
43 set brown background
44 set blue background
45 set magenta background
46 set cyan background
47 set white background
48 256/24-bit background color follows, shoehorned into 8 basic colors
49 set default background color
90..97 set foreground to bright versions of 30..37
100..107 set background, same as 40..47 (bright not supported)

Commands 38 and 48 require further arguments:
;5;x 256 color: values 0..15 are IBGR (black, red, green, ... white), 16..231 a 6x6x6 color cube, 232..255 a grayscale ramp
;2;r;g;b 24-bit color, r/g/b components are in the range 0..255

ECMA-48 Mode Switches

ESC [3 h
DECCRM (default off): Display control chars.

ESC [4 h
DECIM (default off): Set insert mode.

ESC [20 h
LF/NL (default off): Automatically follow echo of LF, VT, or FF with CR.

ECMA-48 Status Report Commands

ESC [5 n
Device status report (DSR): Answer is ESC [0 n (Terminal OK).

ESC [6 n
Cursor position report (CPR): Answer is ESC [*y* ; *x* R, where *x,y* is the cursor location.

DEC Private Mode (DECSET/DECRST) sequences

These are not described in ECMA-48. We list the Set Mode sequences; the Reset Mode sequences are obtained by replacing the final 'h' by 'l'.

ESC [? 1 h
DECCKM (default off): When set, the cursor keys send an ESC O prefix, rather than ESC [.

ESC [? 3 h
: DECCOLM (default off = 80 columns): 80/132 col mode switch. The driver sources note that this alone does not suffice; some user-mode utility such as *resizecons*(8) has to change the hardware registers on the console video card.

ESC [? 5 h
: DECSCNM (default off): Set reverse-video mode.

ESC [? 6 h
: DECOM (default off): When set, cursor addressing is relative to the upper left corner of the scrolling region.

ESC [? 7 h
: DECAWM (default on): Set autowrap on. In this mode, a graphic character emitted after column 80 (or column 132 of DECCOLM is on) forces a wrap to the beginning of the following line first.

ESC [? 8 h
: DECARM (default on): Set keyboard autorepeat on.

ESC [? 9 h
: X10 Mouse Reporting (default off): Set reporting mode to 1 (or reset to 0)—see below.

ESC [? 25 h
: DECTECM (default on): Make cursor visible.

ESC [? 1000 h
: X11 Mouse Reporting (default off): Set reporting mode to 2 (or reset to 0)—see below.

Linux Console Private CSI Sequences

The following sequences are neither ECMA-48 nor native VT102. They are native to the Linux console driver. Colors are in SGR parameters: 0 = black, 1 = red, 2 = green, 3 = brown, 4 = blue, 5 = magenta, 6 = cyan, 7 = white; 8–15 = bright versions of 0–7.

ESC [1 ; *n*] Set color *n* as the underline color.
ESC [2 ; *n*] Set color *n* as the dim color.
ESC [8] Make the current color pair the default attributes.
ESC [9 ; *n*] Set screen blank timeout to *n* minutes.
ESC [10 ; *n*] Set bell frequency in Hz.
ESC [11 ; *n*] Set bell duration in msec.
ESC [12 ; *n*] Bring specified console to the front.
ESC [13] Unblank the screen.
ESC [14 ; *n*] Set the VESA powerdown interval in minutes.
ESC [15] Bring the previous console to the front (since Linux 2.6.0).
ESC [16 ; *n*] Set the cursor blink interval in milliseconds (since Linux 4.2).

Character sets

The kernel knows about 4 translations of bytes into console-screen symbols. The four tables are: a) Latin1 −> PC, b) VT100 graphics −> PC, c) PC −> PC, d) user-defined.

There are two character sets, called G0 and G1, and one of them is the current character set. (Initially G0.) Typing **^N** causes G1 to become current, **^O** causes G0 to become current.

These variables G0 and G1 point at a translation table, and can be changed by the user. Initially they point at tables a) and b), respectively. The sequences ESC (B and ESC (0 and ESC (U and ESC (K cause G0 to point at translation table a), b), c), and d), respectively. The sequences ESC) B and ESC) 0 and ESC) U and ESC) K cause G1 to point at translation table a), b), c), and d), respectively.

The sequence ESC c causes a terminal reset, which is what you want if the screen is all garbled. The oft-advised "echo ^V^O" will make only G0 current, but there is no guarantee that G0 points at table a). In some distributions there is a program *reset*(1) that just does "echo ^[c". If your terminfo entry for the console is correct (and has an entry rs1=\Ec), then "tput reset" will also work.

The user-defined mapping table can be set using *mapscrn*(8)The result of the mapping is that if a symbol c is printed, the symbol s = map[c] is sent to the video memory. The bitmap that corresponds to s is found in the character ROM, and can be changed using *setfont*(8)

Mouse tracking

The mouse tracking facility is intended to return *xterm*(1)-compatible mouse status reports. Because the console driver has no way to know the device or type of the mouse, these reports are returned in the console input stream only when the virtual terminal driver receives a mouse update ioctl. These ioctls must be generated by a mouse-aware user-mode application such as the *gpm*(8) daemon.

The mouse tracking escape sequences generated by **xterm**(1) encode numeric parameters in a single character as *value*+040. For example, '!' is 1. The screen coordinate system is 1-based.

The X10 compatibility mode sends an escape sequence on button press encoding the location and the mouse button pressed. It is enabled by sending ESC [? 9 h and disabled with ESC [? 9 l. On button press, **xterm**(1) sends ESC [M *bxy* (6 characters). Here *b* is button−1, and *x* and *y* are the x and y coordinates of the mouse when the button was pressed. This is the same code the kernel also produces.

Normal tracking mode (not implemented in Linux 2.0.24) sends an escape sequence on both button press and release. Modifier information is also sent. It is enabled by sending ESC [? 1000 h and disabled with ESC [? 1000 l. On button press or release, **xterm**(1) sends ESC [M *bxy*. The low two bits of *b* encode button information: 0=MB1 pressed, 1=MB2 pressed, 2=MB3 pressed, 3=release. The upper bits encode what modifiers were down when the button was pressed and are added together: 4=Shift, 8=Meta, 16=Control. Again *x* and *y* are the x and y coordinates of the mouse event. The upper left corner is (1,1).

Comparisons with other terminals

Many different terminal types are described, like the Linux console, as being "VT100-compatible". Here we discuss differences between the Linux console and the two most important others, the DEC VT102 and *xterm*(1)

Control-character handling

The VT102 also recognized the following control characters:

NUL (0x00)
: was ignored;

ENQ (0x05)
: triggered an answerback message;

DC1 (0x11, **^Q**, XON)
: resumed transmission;

DC3 (0x13, **^S**, XOFF)
: caused VT100 to ignore (and stop transmitting) all codes except XOFF and XON.

VT100-like DC1/DC3 processing may be enabled by the terminal driver.

The *xterm*(1) program (in VT100 mode) recognizes the control characters BEL, BS, HT, LF, VT, FF, CR, SO, SI, ESC.

Escape sequences

VT100 console sequences not implemented on the Linux console:

ESC N	SS2	Single shift 2. (Select G2 character set for the next character only.)
ESC O	SS3	Single shift 3. (Select G3 character set for the next character only.)
ESC P	DCS	Device control string (ended by ESC \)
ESC X	SOS	Start of string.
ESC ^	PM	Privacy message (ended by ESC \)
ESC \	ST	String terminator
ESC * ...		Designate G2 character set
ESC + ...		Designate G3 character set

The program *xterm*(1) (in VT100 mode) recognizes ESC c, ESC # 8, ESC >, ESC =, ESC D, ESC E, ESC H, ESC M, ESC N, ESC O, ESC P ... ESC \, ESC Z (it answers ESC [? 1 ; 2 c, "I am a VT100 with advanced video option") and ESC ^ ... ESC \ with the same meanings as indicated above. It accepts ESC (, ESC), ESC *, ESC + followed by 0, A, B for the DEC special character and line drawing set, UK, and US-ASCII, respectively.

The user can configure **xterm**(1) to respond to VT220-specific control sequences, and it will identify itself as a VT52, VT100, and up depending on the way it is configured and initialized.

It accepts ESC] (OSC) for the setting of certain resources. In addition to the ECMA-48 string terminator (ST), **xterm**(1) accepts a BEL to terminate an OSC string. These are a few of the OSC control sequences recognized by **xterm**(1):

ESC] 0 ; *txt* ST	Set icon name and window title to *txt*.
ESC] 1 ; *txt* ST	Set icon name to *txt*.
ESC] 2 ; *txt* ST	Set window title to *txt*.
ESC] 4 ; *num*; *txt* ST	Set ANSI color *num* to *txt*.
ESC] 10 ; *txt* ST	Set dynamic text color to *txt*.
ESC] 4 6 ; *name* ST	Change log file to *name* (normally disabled by a compile-time option).
ESC] 5 0 ; *fn* ST	Set font to *fn*.

It recognizes the following with slightly modified meaning (saving more state, behaving closer to VT100/VT220):

ESC 7 DECSC	Save cursor
ESC 8 DECRC	Restore cursor

It also recognizes

ESC F		Cursor to lower left corner of screen (if enabled by*xterm*(1)'s **hpLowerleftBugCompat** resource).
ESC l		Memory lock (per HP terminals). Locks memory above the cursor.
ESC m		Memory unlock (per HP terminals).
ESC n	LS2	Invoke the G2 character set.
ESC o	LS3	Invoke the G3 character set.
ESC |	LS3R	Invoke the G3 character set as GR.
ESC }	LS2R	Invoke the G2 character set as GR.
ESC ~	LS1R	Invoke the G1 character set as GR.

It also recognizes ESC % and provides a more complete UTF-8 implementation than Linux console.

CSI Sequences

Old versions of **xterm**(1), for example, from X11R5, interpret the blink SGR as a bold SGR. Later versions which implemented ANSI colors, for example, XFree86 3.1.2A in 1995, improved this by allowing the blink attribute to be displayed as a color. Modern versions of xterm implement blink SGR as blinking text and still allow colored text as an alternate rendering of SGRs. Stock X11R6 versions did not recognize the color-setting SGRs until the X11R6.8 release, which incorporated XFree86 xterm. All ECMA-48 CSI sequences recognized by Linux are also recognized by *xterm*, however **xterm**(1) implements several ECMA-48 and DEC control sequences not recognized by Linux.

The **xterm**(1) program recognizes all of the DEC Private Mode sequences listed above, but none of the Linux private-mode sequences. For discussion of **xterm**(1)'s own private-mode sequences, refer to the *Xterm Control Sequences* document by Edward Moy, Stephen Gildea, and Thomas E. Dickey available with the X distribution. That document, though terse, is much longer than this manual page. For a chronological overview,

details changes to xterm.

The *vttest* program

demonstrates many of these control sequences. The **xterm**(1) source distribution also contains sample scripts which exercise other features.

NOTES

ESC 8 (DECRC) is not able to restore the character set changed with ESC %.

BUGS

In Linux 2.0.23, CSI is broken, and NUL is not ignored inside escape sequences.

Some older kernel versions (after Linux 2.0) interpret 8-bit control sequences. These "C1 controls" use codes between 128 and 159 to replace ESC [, ESC] and similar two-byte control sequence initiators. There are fragments of that in modern kernels (either overlooked or broken by changes to support UTF-8), but the implementation is incomplete and should be regarded as unreliable.

Linux "private mode" sequences do not follow the rules in ECMA-48 for private mode control sequences. In particular, those ending with] do not use a standard terminating character. The OSC (set palette) sequence is a greater problem, since **xterm**(1) may interpret this as a control sequence which requires a string terminator (ST). Unlike the **setterm**(1) sequences which will be ignored (since they are invalid control sequences), the palette sequence will make **xterm**(1) appear to hang (though pressing the return-key will fix that). To accommodate applications which have been hardcoded to use Linux control sequences, set the **xterm**(1) resource **brokenLinuxOSC** to true.

An older version of this document implied that Linux recognizes the ECMA-48 control sequence for invisible text. It is ignored.

SEE ALSO

ioctl_console(2), *charsets*(7)

NAME

cpuid − x86 CPUID access device

DESCRIPTION

CPUID provides an interface for querying information about the x86 CPU.

This device is accessed by *lseek*(2) or *pread*(2) to the appropriate CPUID level and reading in chunks of 16 bytes. A larger read size means multiple reads of consecutive levels.

The lower 32 bits of the file position is used as the incoming *%eax*, and the upper 32 bits of the file position as the incoming *%ecx*, the latter is intended for "counting" *eax* levels like *eax=4*.

This driver uses */dev/cpu/CPUNUM/cpuid*, where *CPUNUM* is the minor number, and on an SMP box will direct the access to CPU *CPUNUM* as listed in */proc/cpuinfo*.

This file is protected so that it can be read only by the user *root*, or members of the group *root*.

NOTES

The CPUID instruction can be directly executed by a program using inline assembler. However this device allows convenient access to all CPUs without changing process affinity.

Most of the information in *cpuid* is reported by the kernel in cooked form either in */proc/cpuinfo* or through subdirectories in */sys/devices/system/cpu*. Direct CPUID access through this device should only be used in exceptional cases.

The *cpuid* driver is not auto-loaded. On modular kernels you might need to use the following command to load it explicitly before use:

```
$ modprobe cpuid
```

There is no support for CPUID functions that require additional input registers.

Early i486 CPUs do not support the CPUID instruction; opening this device for those CPUs fails with EIO.

SEE ALSO

cpuid(1)

Intel Corporation, Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference, A-M, 3-180 CPUID reference.

Intel Corporation, Intel Processor Identification and the CPUID Instruction, Application note 485.

NAME

dsp56k − DSP56001 interface device

SYNOPSIS

#include <asm/dsp56k.h>

ssize_t read(int *fd***, void ****data***, size_t** *length***);**
ssize_t write(int *fd***, void ****data***, size_t** *length***);**

int ioctl(int *fd***, DSP56K_UPLOAD, struct dsp56k_upload ****program***);**
int ioctl(int *fd***, DSP56K_SET_TX_WSIZE, int** *wsize***);**
int ioctl(int *fd***, DSP56K_SET_RX_WSIZE, int** *wsize***);**
int ioctl(int *fd***, DSP56K_HOST_FLAGS, struct dsp56k_host_flags *** *flags***);**
int ioctl(int *fd***, DSP56K_HOST_CMD, int** *cmd***);**

CONFIGURATION

The *dsp56k* device is a character device with major number 55 and minor number 0.

DESCRIPTION

The Motorola DSP56001 is a fully programmable 24-bit digital signal processor found in Atari Falcon030-compatible computers. The *dsp56k* special file is used to control the DSP56001, and to send and receive data using the bidirectional handshaked host port.

To send a data stream to the signal processor, use *write*(2) to the device, and *read*(2) to receive processed data. The data can be sent or received in 8, 16, 24, or 32-bit quantities on the host side, but will always be seen as 24-bit quantities in the DSP56001.

The following *ioctl*(2) calls are used to control the *dsp56k* device:

DSP56K_UPLOAD
: resets the DSP56001 and uploads a program. The third *ioctl*(2) argument must be a pointer to a *struct dsp56k_upload* with members *bin* pointing to a DSP56001 binary program, and *len* set to the length of the program, counted in 24-bit words.

DSP56K_SET_TX_WSIZE
: sets the transmit word size. Allowed values are in the range 1 to 4, and is the number of bytes that will be sent at a time to the DSP56001. These data quantities will either be padded with bytes containing zero, or truncated to fit the native 24-bit data format of the DSP56001.

DSP56K_SET_RX_WSIZE
: sets the receive word size. Allowed values are in the range 1 to 4, and is the number of bytes that will be received at a time from the DSP56001. These data quantities will either truncated, or padded with a null byte ('\0'), to fit the native 24-bit data format of the DSP56001.

DSP56K_HOST_FLAGS
: read and write the host flags. The host flags are four general-purpose bits that can be read by both the hosting computer and the DSP56001. Bits 0 and 1 can be written by the host, and bits 2 and 3 can be written by the DSP56001.

  To access the host flags, the third *ioctl*(2) argument must be a pointer to a *struct dsp56k_host_flags*. If bit 0 or 1 is set in the *dir* member, the corresponding bit in *out* will be written to the host flags. The state of all host flags will be returned in the lower four bits of the *status* member.

DSP56K_HOST_CMD
: sends a host command. Allowed values are in the range 0 to 31, and is a user-defined command handled by the program running in the DSP56001.

FILES

/dev/dsp56k

SEE ALSO

linux/include/asm−m68k/dsp56k.h, *linux/drivers/char/dsp56k.c*, , DSP56000/DSP56001 Digital Signal Processor User's Manual

NAME

fd − floppy disk device

CONFIGURATION

Floppy drives are block devices with major number 2. Typically they are owned by root:floppy (i.e., user root, group floppy) and have either mode 0660 (access checking via group membership) or mode 0666 (everybody has access). The minor numbers encode the device type, drive number, and controller number. For each device type (that is, combination of density and track count) there is a base minor number. To this base number, add the drive's number on its controller and 128 if the drive is on the secondary controller. In the following device tables, *n* represents the drive number.

Warning: if you use formats with more tracks than supported by your drive, you may cause it mechanical damage. Trying once if more tracks than the usual 40/80 are supported should not damage it, but no warranty is given for that. If you are not sure, don't create device entries for those formats, so as to prevent their usage.

Drive-independent device files which automatically detect the media format and capacity:

Name	Base minor #
fd*n*	0

5.25 inch double-density device files:

Name	Capacity KiB	Cyl.	Sect.	Heads	Base minor #
fd*n***d360**	360	40	9	2	4

5.25 inch high-density device files:

Name	Capacity KiB	Cyl.	Sect.	Heads	Base minor #
fd*n***h360**	360	40	9	2	20
fd*n***h410**	410	41	10	2	48
fd*n***h420**	420	42	10	2	64
fd*n***h720**	720	80	9	2	24
fd*n***h880**	880	80	11	2	80
fd*n***h1200**	1200	80	15	2	8
fd*n***h1440**	1440	80	18	2	40
fd*n***h1476**	1476	82	18	2	56
fd*n***h1494**	1494	83	18	2	72
fd*n***h1600**	1600	80	20	2	92

3.5 inch double-density device files:

Name	Capacity KiB	Cyl.	Sect.	Heads	Base minor #
fd*n***u360**	360	80	9	1	12
fd*n***u720**	720	80	9	2	16
fd*n***u800**	800	80	10	2	120
fd*n***u1040**	1040	80	13	2	84
fd*n***u1120**	1120	80	14	2	88

3.5 inch high-density device files:

Name	Capacity KiB	Cyl.	Sect.	Heads	Base minor #
fd*n***u360**	360	40	9	2	12
fd*n***u720**	720	80	9	2	16
fd*n***u820**	820	82	10	2	52
fd*n***u830**	830	83	10	2	68
fd*n***u1440**	1440	80	18	2	28
fd*n***u1600**	1600	80	20	2	124
fd*n***u1680**	1680	80	21	2	44
fd*n***u1722**	1722	82	21	2	60
fd*n***u1743**	1743	83	21	2	76

fd*n*u1760	1760	80	22	2	96
fd*n*u1840	1840	80	23	2	116
fd*n*u1920	1920	80	24	2	100

3.5 inch extra-density device files:

Name	Capacity KiB	Cyl.	Sect.	Heads	Base minor #
fd*n*u2880	2880	80	36	2	32
fd*n*CompaQ	2880	80	36	2	36
fd*n*u3200	3200	80	40	2	104
fd*n*u3520	3520	80	44	2	108
fd*n*u3840	3840	80	48	2	112

DESCRIPTION

fd special files access the floppy disk drives in raw mode. The following *ioctl*(2) calls are supported by **fd** devices:

FDCLRPRM
clears the media information of a drive (geometry of disk in drive).

FDSETPRM
sets the media information of a drive. The media information will be lost when the media is changed.

FDDEFPRM
sets the media information of a drive (geometry of disk in drive). The media information will not be lost when the media is changed. This will disable autodetection. In order to reenable autodetection, you have to issue an **FDCLRPRM**.

FDGETDRVTYP
returns the type of a drive (name parameter). For formats which work in several drive types, **FDGETDRVTYP** returns a name which is appropriate for the oldest drive type which supports this format.

FDFLUSH
invalidates the buffer cache for the given drive.

FDSETMAXERRS
sets the error thresholds for reporting errors, aborting the operation, recalibrating, resetting, and reading sector by sector.

FDSETMAXERRS
gets the current error thresholds.

FDGETDRVTYP
gets the internal name of the drive.

FDWERRORCLR
clears the write error statistics.

FDWERRORGET
reads the write error statistics. These include the total number of write errors, the location and disk of the first write error, and the location and disk of the last write error. Disks are identified by a generation number which is incremented at (almost) each disk change.

FDTWADDLE
Switch the drive motor off for a few microseconds. This might be needed in order to access a disk whose sectors are too close together.

FDSETDRVPRM
sets various drive parameters.

FDGETDRVPRM
reads these parameters back.

FDGETDRVSTAT
gets the cached drive state (disk changed, write protected et al.)

FDPOLLDRVSTAT
polls the drive and return its state.

FDGETFDCSTAT
gets the floppy controller state.

FDRESET
resets the floppy controller under certain conditions.

FDRAWCMD
sends a raw command to the floppy controller.

For more precise information, consult also the <*linux/fd.h*> and <*linux/fdreg.h*> include files, as well as the *floppycontrol*(1) manual page.

FILES

*/dev/fd**

NOTES

The various formats permit reading and writing many types of disks. However, if a floppy is formatted with an inter-sector gap that is too small, performance may drop, to the point of needing a few seconds to access an entire track. To prevent this, use interleaved formats.

It is not possible to read floppies which are formatted using GCR (group code recording), which is used by Apple II and Macintosh computers (800k disks).

Reading floppies which are hard sectored (one hole per sector, with the index hole being a little skewed) is not supported. This used to be common with older 8-inch floppies.

SEE ALSO

chown(1), *floppycontrol*(1), *getfdprm*(1), *mknod*(1), *superformat*(1), *mount*(8), *setfdprm*(8)

NAME

full − always full device

CONFIGURATION

If your system does not have */dev/full* created already, it can be created with the following commands:

```
mknod -m 666 /dev/full c 1 7
chown root:root /dev/full
```

DESCRIPTION

The file */dev/full* has major device number 1 and minor device number 7.

Writes to the */dev/full* device fail with an **ENOSPC** error. This can be used to test how a program handles disk-full errors.

Reads from the */dev/full* device will return \0 characters.

Seeks on */dev/full* will always succeed.

FILES

/dev/full

SEE ALSO

mknod(1), *null*(4), *zero*(4)

NAME

fuse − Filesystem in Userspace (FUSE) device

SYNOPSIS

#include <linux/fuse.h>

DESCRIPTION

This device is the primary interface between the FUSE filesystem driver and a user-space process wishing to provide the filesystem (referred to in the rest of this manual page as the *filesystem daemon*). This manual page is intended for those interested in understanding the kernel interface itself. Those implementing a FUSE filesystem may wish to make use of a user-space library such as *libfuse* that abstracts away the low-level interface.

At its core, FUSE is a simple client-server protocol, in which the Linux kernel is the client and the daemon is the server. After obtaining a file descriptor for this device, the daemon may *read*(2) requests from that file descriptor and is expected to *write*(2) back its replies. It is important to note that a file descriptor is associated with a unique FUSE filesystem. In particular, opening a second copy of this device, will not allow access to resources created through the first file descriptor (and vice versa).

The basic protocol

Every message that is read by the daemon begins with a header described by the following structure:

```
struct fuse_in_header {
    uint32_t len;       /* Total length of the data,
                           including this header */
    uint32_t opcode;    /* The kind of operation (see below) */
    uint64_t unique;    /* A unique identifier for this request */
    uint64_t nodeid;    /* ID of the filesystem object
                           being operated on */
    uint32_t uid;       /* UID of the requesting process */
    uint32_t gid;       /* GID of the requesting process */
    uint32_t pid;       /* PID of the requesting process */
    uint32_t padding;
};
```

The header is followed by a variable-length data portion (which may be empty) specific to the requested operation (the requested operation is indicated by *opcode*).

The daemon should then process the request and if applicable send a reply (almost all operations require a reply; if they do not, this is documented below), by performing a *write*(2) to the file descriptor. All replies must start with the following header:

```
struct fuse_out_header {
    uint32_t len;       /* Total length of data written to
                           the file descriptor */
    int32_t  error;     /* Any error that occurred (0 if none) */
    uint64_t unique;    /* The value from the
                           corresponding request */
};
```

This header is also followed by (potentially empty) variable-sized data depending on the executed request. However, if the reply is an error reply (i.e., *error* is set), then no further payload data should be sent, independent of the request.

Exchanged messages

This section should contain documentation for each of the messages in the protocol. This manual page is currently incomplete, so not all messages are documented. For each message, first the struct sent by the kernel is given, followed by a description of the semantics of the message.

FUSE_INIT

```
struct fuse_init_in {
    uint32_t major;
    uint32_t minor;
    uint32_t max_readahead; /* Since protocol v7.6 */
    uint32_t flags;         /* Since protocol v7.6 */
```

```
};
```

This is the first request sent by the kernel to the daemon. It is used to negotiate the protocol version and other filesystem parameters. Note that the protocol version may affect the layout of any structure in the protocol (including this structure). The daemon must thus remember the negotiated version and flags for each session. As of the writing of this man page, the highest supported kernel protocol version is *7.26*.

Users should be aware that the descriptions in this manual page may be incomplete or incorrect for older or more recent protocol versions.

The reply for this request has the following format:

```
struct fuse_init_out {
    uint32_t major;
    uint32_t minor;
    uint32_t max_readahead;   /* Since v7.6 */
    uint32_t flags;           /* Since v7.6; some flags bits
                                 were introduced later */
    uint16_t max_background;  /* Since v7.13 */
    uint16_t congestion_threshold;  /* Since v7.13 */
    uint32_t max_write;       /* Since v7.5 */
    uint32_t time_gran;       /* Since v7.6 */
    uint32_t unused[9];
};
```

If the major version supported by the kernel is larger than that supported by the daemon, the reply shall consist of only *uint32_t major* (following the usual header), indicating the largest major version supported by the daemon. The kernel will then issue a new **FUSE_INIT** request conforming to the older version. In the reverse case, the daemon should quietly fall back to the kernel's major version.

The negotiated minor version is considered to be the minimum of the minor versions provided by the daemon and the kernel and both parties should use the protocol corresponding to said minor version.

FUSE_GETATTR

```
struct fuse_getattr_in {
    uint32_t getattr_flags;
    uint32_t dummy;
    uint64_t fh;      /* Set only if
                         (getattr_flags & FUSE_GETATTR_FH)
};
```

The requested operation is to compute the attributes to be returned by *stat*(2) and similar operations for the given filesystem object. The object for which the attributes should be computed is indicated either by *header->nodeid* or, if the **FUSE_GETATTR_FH** flag is set, by the file handle *fh*. The latter case of operation is analogous to *fstat*(2).

For performance reasons, these attributes may be cached in the kernel for a specified duration of time. While the cache timeout has not been exceeded, the attributes will be served from the cache and will not cause additional **FUSE_GETATTR** requests.

The computed attributes and the requested cache timeout should then be returned in the following structure:

```
struct fuse_attr_out {
    /* Attribute cache duration (seconds + nanoseconds) */
    uint64_t attr_valid;
    uint32_t attr_valid_nsec;
    uint32_t dummy;
    struct fuse_attr {
        uint64_t ino;
        uint64_t size;
        uint64_t blocks;
```

```
            uint64_t atime;
            uint64_t mtime;
            uint64_t ctime;
            uint32_t atimensec;
            uint32_t mtimensec;
            uint32_t ctimensec;
            uint32_t mode;
            uint32_t nlink;
            uint32_t uid;
            uint32_t gid;
            uint32_t rdev;
            uint32_t blksize;
            uint32_t padding;
        } attr;
    };
```

FUSE_ACCESS

```
    struct fuse_access_in {
        uint32_t mask;
        uint32_t padding;
    };
```

If the *default_permissions* mount options is not used, this request may be used for permissions checking. No reply data is expected, but errors may be indicated as usual by setting the *error* field in the reply header (in particular, access denied errors may be indicated by returning **-EACCES**).

FUSE_OPEN and **FUSE_OPENDIR**

```
    struct fuse_open_in {
        uint32_t flags;     /* The flags that were passed
                               to the open(2) */
        uint32_t unused;
    };
```

The requested operation is to open the node indicated by *header->nodeid*. The exact semantics of what this means will depend on the filesystem being implemented. However, at the very least the filesystem should validate that the requested *flags* are valid for the indicated resource and then send a reply with the following format:

```
    struct fuse_open_out {
        uint64_t fh;
        uint32_t open_flags;
        uint32_t padding;
    };
```

The *fh* field is an opaque identifier that the kernel will use to refer to this resource The *open_flags* field is a bit mask of any number of the flags that indicate properties of this file handle to the kernel:

FOPEN_DIRECT_IO
: Bypass page cache for this open file.

FOPEN_KEEP_CACHE
: Don't invalidate the data cache on open.

FOPEN_NONSEEKABLE
: The file is not seekable.

FUSE_READ and **FUSE_READDIR**

```
    struct fuse_read_in {
        uint64_t fh;
        uint64_t offset;
        uint32_t size;
        uint32_t read_flags;
```

```
    uint64_t lock_owner;
    uint32_t flags;
    uint32_t padding;
};
```

The requested action is to read up to *size* bytes of the file or directory, starting at *offset*. The bytes should be returned directly following the usual reply header.

FUSE_INTERRUPT

```
struct fuse_interrupt_in {
    uint64_t unique;
};
```

The requested action is to cancel the pending operation indicated by *unique*. This request requires no response. However, receipt of this message does not by itself cancel the indicated operation. The kernel will still expect a reply to said operation (e.g., an *EINTR* error or a short read). At most one **FUSE_INTERRUPT** request will be issued for a given operation. After issuing said operation, the kernel will wait uninterruptibly for completion of the indicated request.

FUSE_LOOKUP

Directly following the header is a filename to be looked up in the directory indicated by *header->nodeid*. The expected reply is of the form:

```
struct fuse_entry_out {
    uint64_t nodeid;            /* Inode ID */
    uint64_t generation;        /* Inode generation */
    uint64_t entry_valid;
    uint64_t attr_valid;
    uint32_t entry_valid_nsec;
    uint32_t attr_valid_nsec;
    struct fuse_attr attr;
};
```

The combination of *nodeid* and *generation* must be unique for the filesystem's lifetime.

The interpretation of timeouts and *attr* is as for **FUSE_GETATTR**.

FUSE_FLUSH

```
struct fuse_flush_in {
    uint64_t fh;
    uint32_t unused;
    uint32_t padding;
    uint64_t lock_owner;
};
```

The requested action is to flush any pending changes to the indicated file handle. No reply data is expected. However, an empty reply message still needs to be issued once the flush operation is complete.

FUSE_RELEASE and **FUSE_RELEASEDIR**

```
struct fuse_release_in {
    uint64_t fh;
    uint32_t flags;
    uint32_t release_flags;
    uint64_t lock_owner;
};
```

These are the converse of **FUSE_OPEN** and **FUSE_OPENDIR** respectively. The daemon may now free any resources associated with the file handle *fh* as the kernel will no longer refer to it. There is no reply data associated with this request, but a reply still needs to be issued once the request has been completely processed.

FUSE_STATFS

This operation implements *statfs*(2) for this filesystem. There is no input data associated with this request. The expected reply data has the following structure:

```
struct fuse_kstatfs {
    uint64_t blocks;
    uint64_t bfree;
    uint64_t bavail;
    uint64_t files;
    uint64_t ffree;
    uint32_t bsize;
    uint32_t namelen;
    uint32_t frsize;
    uint32_t padding;
    uint32_t spare[6];
};

struct fuse_statfs_out {
    struct fuse_kstatfs st;
};
```

For the interpretation of these fields, see *statfs*(2).

ERRORS

E2BIG Returned from *read*(2) operations when the kernel's request is too large for the provided buffer and the request was **FUSE_SETXATTR**.

EINVAL
Returned from *write*(2) if validation of the reply failed. Not all mistakes in replies will be caught by this validation. However, basic mistakes, such as short replies or an incorrect *unique* value, are detected.

EIO Returned from *read*(2) operations when the kernel's request is too large for the provided buffer.

Note: There are various ways in which incorrect use of these interfaces can cause operations on the provided filesystem's files and directories to fail with **EIO**. Among the possible incorrect uses are:

- changing *mode & S_IFMT* for an inode that has previously been reported to the kernel; or
- giving replies to the kernel that are shorter than what the kernel expected.

ENODEV
Returned from *read*(2) and *write*(2) if the FUSE filesystem was unmounted.

EPERM
Returned from operations on a */dev/fuse* file descriptor that has not been mounted.

STANDARDS

Linux.

NOTES

The following messages are not yet documented in this manual page:

FUSE_BATCH_FORGET
FUSE_BMAP
FUSE_CREATE
FUSE_DESTROY
FUSE_FALLOCATE
FUSE_FORGET
FUSE_FSYNC
FUSE_FSYNCDIR
FUSE_GETLK
FUSE_GETXATTR
FUSE_IOCTL
FUSE_LINK
FUSE_LISTXATTR
FUSE_LSEEK
FUSE_MKDIR

FUSE_MKNOD
FUSE_NOTIFY_REPLY
FUSE_POLL
FUSE_READDIRPLUS
FUSE_READLINK
FUSE_REMOVEXATTR
FUSE_RENAME
FUSE_RENAME2
FUSE_RMDIR
FUSE_SETATTR
FUSE_SETLK
FUSE_SETLKW
FUSE_SYMLINK
FUSE_UNLINK
FUSE_WRITE

SEE ALSO

fusermount(1), *mount.fuse*(8)

NAME

hd − MFM/IDE hard disk devices

DESCRIPTION

The **hd*** devices are block devices to access MFM/IDE hard disk drives in raw mode. The master drive on the primary IDE controller (major device number 3) is **hda**; the slave drive is **hdb**. The master drive of the second controller (major device number 22) is **hdc** and the slave is **hdd**.

General IDE block device names have the form **hd***X*, or **hd***XP*, where *X* is a letter denoting the physical drive, and *P* is a number denoting the partition on that physical drive. The first form, **hd***X*, is used to address the whole drive. Partition numbers are assigned in the order the partitions are discovered, and only nonempty, nonextended partitions get a number. However, partition numbers 1–4 are given to the four partitions described in the MBR (the "primary" partitions), regardless of whether they are unused or extended. Thus, the first logical partition will be **hd***X***5**. Both DOS-type partitioning and BSD-disklabel partitioning are supported. You can have at most 63 partitions on an IDE disk.

For example, */dev/hda* refers to all of the first IDE drive in the system; and */dev/hdb3* refers to the third DOS "primary" partition on the second one.

They are typically created by:

```
mknod −m 660 /dev/hda b 3 0
mknod −m 660 /dev/hda1 b 3 1
mknod −m 660 /dev/hda2 b 3 2
...
mknod −m 660 /dev/hda8 b 3 8
mknod −m 660 /dev/hdb b 3 64
mknod −m 660 /dev/hdb1 b 3 65
mknod −m 660 /dev/hdb2 b 3 66
...
mknod −m 660 /dev/hdb8 b 3 72
chown root:disk /dev/hd*
```

FILES

*/dev/hd**

SEE ALSO

chown(1), *mknod*(1), *sd*(4), *mount*(8)

NAME

hpsa − HP Smart Array SCSI driver

SYNOPSIS

modprobe hpsa [hpsa_allow_any=1]

DESCRIPTION

hpsa is a SCSI driver for HP Smart Array RAID controllers.

Options

hpsa_allow_any=1: This option allows the driver to attempt to operate on any HP Smart Array hardware RAID controller, even if it is not explicitly known to the driver. This allows newer hardware to work with older drivers. Typically this is used to allow installation of operating systems from media that predates the RAID controller, though it may also be used to enable **hpsa** to drive older controllers that would normally be handled by the *cciss*(4) driver. These older boards have not been tested and are not supported with **hpsa**, and *cciss*(4) should still be used for these.

Supported hardware

The **hpsa** driver supports the following Smart Array boards:

Smart Array P700M
Smart Array P212
Smart Array P410
Smart Array P410i
Smart Array P411
Smart Array P812
Smart Array P712m
Smart Array P711m
StorageWorks P1210m

Since Linux 4.14, the following Smart Array boards are also supported:

Smart Array 5300
Smart Array 5312
Smart Array 532
Smart Array 5i
Smart Array 6400
Smart Array 6400 EM
Smart Array 641
Smart Array 642
Smart Array 6i
Smart Array E200
Smart Array E200i
Smart Array E200i
Smart Array E200i
Smart Array E200i
Smart Array E500
Smart Array P400
Smart Array P400i
Smart Array P600
Smart Array P700m
Smart Array P800

Configuration details

To configure HP Smart Array controllers, use the HP Array Configuration Utility (either *hpacuxe*(8) or *hpacucli*(8)) or the Offline ROM-based Configuration Utility (ORCA) run from the Smart Array's option ROM at boot time.

FILES

Device nodes

Logical drives are accessed via the SCSI disk driver (**sd**(4)), tape drives via the SCSI tape driver (**st**(4)), and the RAID controller via the SCSI generic driver (**sg**(4)), with device nodes named */dev/sd**, */dev/st**, and */dev/sg**, respectively.

HPSA-specific host attribute files in /sys

/sys/class/scsi_host/host/rescan*

This is a write-only attribute. Writing to this attribute will cause the driver to scan for new, changed, or removed devices (e.g., hot-plugged tape drives, or newly configured or deleted logical drives, etc.) and notify the SCSI midlayer of any changes detected. Normally a rescan is triggered automatically by HP's Array Configuration Utility (either the GUI or the command-line variety); thus, for logical drive changes, the user should not normally have to use this attribute. This attribute may be useful when hot plugging devices like tape drives, or entire storage boxes containing preconfigured logical drives.

/sys/class/scsi_host/host/firmware_revision*

This attribute contains the firmware version of the Smart Array.

For example:

```
# cd /sys/class/scsi_host/host4
# cat firmware_revision
7.14
```

HPSA-specific disk attribute files in /sys

/sys/class/scsi_disk/c:b:t:l/device/unique_id

This attribute contains a 32 hex-digit unique ID for each logical drive.

For example:

```
# cd /sys/class/scsi_disk/4:0:0:0/device
# cat unique_id
600508B1001044395355323037570F77
```

/sys/class/scsi_disk/c:b:t:l/device/raid_level

This attribute contains the RAID level of each logical drive.

For example:

```
# cd /sys/class/scsi_disk/4:0:0:0/device
# cat raid_level
RAID 0
```

/sys/class/scsi_disk/c:b:t:l/device/lunid

This attribute contains the 16 hex-digit (8 byte) LUN ID by which a logical drive or physical device can be addressed. *c*:*b*:*t*:*l* are the controller, bus, target, and lun of the device.

For example:

```
# cd /sys/class/scsi_disk/4:0:0:0/device
# cat lunid
0x0000004000000000
```

Supported ioctl() operations

For compatibility with applications written for the *cciss*(4) driver, many, but not all of the ioctls supported by the *cciss*(4) driver are also supported by the **hpsa** driver. The data structures used by these ioctls are described in the Linux kernel source file *include/linux/cciss_ioctl.h*.

CCISS_DEREGDISK
CCISS_REGNEWDISK
CCISS_REGNEWD

These three ioctls all do exactly the same thing, which is to cause the driver to rescan for new devices. This does exactly the same thing as writing to the hpsa-specific host "rescan" attribute.

CCISS_GETPCIINFO

Returns PCI domain, bus, device, and function and "board ID" (PCI subsystem ID).

CCISS_GETDRIVVER

Returns driver version in three bytes encoded as:

```
(major_version << 16) | (minor_version << 8) |
    (subminor_version)
```

CCISS_PASSTHRU
CCISS_BIG_PASSTHRU
Allows "BMIC" and "CISS" commands to be passed through to the Smart Array. These are used extensively by the HP Array Configuration Utility, SNMP storage agents, and so on. See *cciss_vol_status* at for some examples.

SEE ALSO

cciss(4), *sd*(4), *st*(4), *cciss_vol_status*(8), *hpacucli*(8), *hpacuxe*(8)

, and *Documentation/scsi/hpsa.txt* and *Documentation/ABI/testing/sysfs−bus−pci−devices−cciss* in the Linux kernel source tree

NAME

initrd − boot loader initialized RAM disk

CONFIGURATION

/dev/initrd is a read-only block device assigned major number 1 and minor number 250. Typically */dev/initrd* is owned by root:disk with mode 0400 (read access by root only). If the Linux system does not have */dev/initrd* already created, it can be created with the following commands:

```
mknod −m 400 /dev/initrd b 1 250
chown root:disk /dev/initrd
```

Also, support for both "RAM disk" and "Initial RAM disk" (e.g., **CONFIG_BLK_DEV_RAM=y** and **CONFIG_BLK_DEV_INITRD=y**) must be compiled directly into the Linux kernel to use */dev/initrd*. When using */dev/initrd*, the RAM disk driver cannot be loaded as a module.

DESCRIPTION

The special file */dev/initrd* is a read-only block device. This device is a RAM disk that is initialized (e.g., loaded) by the boot loader before the kernel is started. The kernel then can use */dev/initrd*'s contents for a two-phase system boot-up.

In the first boot-up phase, the kernel starts up and mounts an initial root filesystem from the contents of */dev/initrd* (e.g., RAM disk initialized by the boot loader). In the second phase, additional drivers or other modules are loaded from the initial root device's contents. After loading the additional modules, a new root filesystem (i.e., the normal root filesystem) is mounted from a different device.

Boot-up operation

When booting up with **initrd**, the system boots as follows:

(1) The boot loader loads the kernel program and */dev/initrd*'s contents into memory.

(2) On kernel startup, the kernel uncompresses and copies the contents of the device */dev/initrd* onto device */dev/ram0* and then frees the memory used by */dev/initrd*.

(3) The kernel then read-write mounts the device */dev/ram0* as the initial root filesystem.

(4) If the indicated normal root filesystem is also the initial root filesystem (e.g., */dev/ram0*) then the kernel skips to the last step for the usual boot sequence.

(5) If the executable file */linuxrc* is present in the initial root filesystem, */linuxrc* is executed with UID 0. (The file */linuxrc* must have executable permission. The file */linuxrc* can be any valid executable, including a shell script.)

(6) If */linuxrc* is not executed or when */linuxrc* terminates, the normal root filesystem is mounted. (If */linuxrc* exits with any filesystems mounted on the initial root filesystem, then the behavior of the kernel is **UNSPECIFIED**. See the NOTES section for the current kernel behavior.)

(7) If the normal root filesystem has a directory */initrd*, the device */dev/ram0* is moved from / to */initrd*. Otherwise, if the directory */initrd* does not exist, the device */dev/ram0* is unmounted. (When moved from / to */initrd*, */dev/ram0* is not unmounted and therefore processes can remain running from */dev/ram0*. If directory */initrd* does not exist on the normal root filesystem and any processes remain running from */dev/ram0* when */linuxrc* exits, the behavior of the kernel is **UNSPECIFIED**. See the NOTES section for the current kernel behavior.)

(8) The usual boot sequence (e.g., invocation of */sbin/init*) is performed on the normal root filesystem.

Options

The following boot loader options, when used with **initrd**, affect the kernel's boot-up operation:

initrd= *filename*

Specifies the file to load as the contents of */dev/initrd*. For **LOADLIN** this is a command-line option. For **LILO** you have to use this command in the **LILO** configuration file */etc/lilo.config*. The filename specified with this option will typically be a gzipped filesystem image.

noinitrd

This boot option disables the two-phase boot-up operation. The kernel performs the usual boot sequence as if */dev/initrd* was not initialized. With this option, any contents of */dev/initrd* loaded into memory by the boot loader contents are preserved. This option permits the contents of */dev/initrd* to be any data and need not be limited to a filesystem image. However,

device */dev/initrd* is read-only and can be read only one time after system startup.

root=*device-name*
Specifies the device to be used as the normal root filesystem. For **LOADLIN** this is a command-line option. For **LILO** this is a boot time option or can be used as an option line in the **LILO** configuration file */etc/lilo.config*. The device specified by this option must be a mountable device having a suitable root filesystem.

Changing the normal root filesystem

By default, the kernel's settings (e.g., set in the kernel file with *rdev*(8) or compiled into the kernel file), or the boot loader option setting is used for the normal root filesystems. For an NFS-mounted normal root filesystem, one has to use the **nfs_root_name** and **nfs_root_addrs** boot options to give the NFS settings. For more information on NFS-mounted root see the kernel documentation file *Documentation/filesystems/nfs/nfsroot.txt* (or *Documentation/filesystems/nfsroot.txt* before Linux 2.6.33). For more information on setting the root filesystem see also the **LILO** and **LOADLIN** documentation.

It is also possible for the */linuxrc* executable to change the normal root device. For */linuxrc* to change the normal root device, */proc* must be mounted. After mounting */proc*, */linuxrc* changes the normal root device by writing into the proc files */proc/sys/kernel/real−root−dev*, */proc/sys/kernel/nfs−root−name*, and */proc/sys/kernel/nfs−root−addrs*. For a physical root device, the root device is changed by having */linuxrc* write the new root filesystem device number into */proc/sys/kernel/real−root−dev*. For an NFS root filesystem, the root device is changed by having */linuxrc* write the NFS setting into files */proc/sys/kernel/nfs−root−name* and */proc/sys/kernel/nfs−root−addrs* and then writing 0xff (e.g., the pseudo-NFS-device number) into file */proc/sys/kernel/real−root−dev*. For example, the following shell command line would change the normal root device to */dev/hdb1*:

```
echo 0x365 >/proc/sys/kernel/real-root-dev
```

For an NFS example, the following shell command lines would change the normal root device to the NFS directory */var/nfsroot* on a local networked NFS server with IP number 193.8.232.7 for a system with IP number 193.8.232.2 and named "idefix":

```
echo /var/nfsroot >/proc/sys/kernel/nfs-root-name
echo 193.8.232.2:193.8.232.7::255.255.255.0:idefix \
    >/proc/sys/kernel/nfs-root-addrs
echo 255 >/proc/sys/kernel/real-root-dev
```

Note: The use of */proc/sys/kernel/real−root−dev* to change the root filesystem is obsolete. See the Linux kernel source file *Documentation/admin−guide/initrd.rst* (or *Documentation/initrd.txt* before Linux 4.10) as well as *pivot_root*(2) and *pivot_root*(8) for information on the modern method of changing the root filesystem.

Usage

The main motivation for implementing **initrd** was to allow for modular kernel configuration at system installation.

A possible system installation scenario is as follows:

(1) The loader program boots from floppy or other media with a minimal kernel (e.g., support for */dev/ram*, */dev/initrd*, and the ext2 filesystem) and loads */dev/initrd* with a gzipped version of the initial filesystem.

(2) The executable */linuxrc* determines what is needed to (1) mount the normal root filesystem (i.e., device type, device drivers, filesystem) and (2) the distribution media (e.g., CD-ROM, network, tape, ...). This can be done by asking the user, by auto-probing, or by using a hybrid approach.

(3) The executable */linuxrc* loads the necessary modules from the initial root filesystem.

(4) The executable */linuxrc* creates and populates the root filesystem. (At this stage the normal root filesystem does not have to be a completed system yet.)

(5) The executable */linuxrc* sets */proc/sys/kernel/real−root−dev*, unmounts */proc*, the normal root filesystem and any other filesystems it has mounted, and then terminates.

(6) The kernel then mounts the normal root filesystem.

(7) Now that the filesystem is accessible and intact, the boot loader can be installed.

(8) The boot loader is configured to load into */dev/initrd* a filesystem with the set of modules that was used to bring up the system. (e.g., device */dev/ram0* can be modified, then unmounted, and finally, the image is written from */dev/ram0* to a file.)

(9) The system is now bootable and additional installation tasks can be performed.

The key role of */dev/initrd* in the above is to reuse the configuration data during normal system operation without requiring initial kernel selection, a large generic kernel or, recompiling the kernel.

A second scenario is for installations where Linux runs on systems with different hardware configurations in a single administrative network. In such cases, it may be desirable to use only a small set of kernels (ideally only one) and to keep the system-specific part of configuration information as small as possible. In this case, create a common file with all needed modules. Then, only the */linuxrc* file or a file executed by */linuxrc* would be different.

A third scenario is more convenient recovery disks. Because information like the location of the root filesystem partition is not needed at boot time, the system loaded from */dev/initrd* can use a dialog and/or auto-detection followed by a possible sanity check.

Last but not least, Linux distributions on CD-ROM may use **initrd** for easy installation from the CD-ROM. The distribution can use **LOADLIN** to directly load */dev/initrd* from CD-ROM without the need of any floppies. The distribution could also use a **LILO** boot floppy and then bootstrap a bigger RAM disk via */dev/initrd* from the CD-ROM.

FILES

/dev/initrd
/dev/ram0
/linuxrc
/initrd

NOTES

- With the current kernel, any filesystems that remain mounted when */dev/ram0* is moved from / to */initrd* continue to be accessible. However, the */proc/mounts* entries are not updated.
- With the current kernel, if directory */initrd* does not exist, then */dev/ram0* will **not** be fully unmounted if */dev/ram0* is used by any process or has any filesystem mounted on it. If */dev/ram0* is **not** fully unmounted, then */dev/ram0* will remain in memory.
- Users of */dev/initrd* should not depend on the behavior given in the above notes. The behavior may change in future versions of the Linux kernel.

SEE ALSO

chown(1), *mknod*(1), *ram*(4), *freeramdisk*(8), *rdev*(8)

Documentation/admin−guide/initrd.rst (or *Documentation/initrd.txt* before Linux 4.10) in the Linux kernel source tree, the LILO documentation, the LOADLIN documentation, the SYSLINUX documentation

NAME

lirc − lirc devices

DESCRIPTION

The */dev/lirc** character devices provide a low-level bidirectional interface to infra-red (IR) remotes. Most of these devices can receive, and some can send. When receiving or sending data, the driver works in two different modes depending on the underlying hardware.

Some hardware (typically TV-cards) decodes the IR signal internally and provides decoded button presses as scancode values. Drivers for this kind of hardware work in **LIRC_MODE_SCANCODE** mode. Such hardware usually does not support sending IR signals. Furthermore, such hardware can only decode a limited set of IR protocols, usually only the protocol of the specific remote which is bundled with, for example, a TV-card.

Other hardware provides a stream of pulse/space durations. Such drivers work in **LIRC_MODE_MODE2** mode. Such hardware can be used with (almost) any kind of remote. This type of hardware can also be used in **LIRC_MODE_SCANCODE** mode, in which case the kernel IR decoders will decode the IR. These decoders can be written in extended BPF (see *bpf*(2)) and attached to the **lirc** device. Sometimes, this kind of hardware also supports sending IR data.

The **LIRC_GET_FEATURES** ioctl (see below) allows probing for whether receiving and sending is supported, and in which modes, amongst other features.

Reading input with the LIRC_MODE_MODE2 mode

In the **LIRC_MODE_MODE2 mode**, the data returned by *read*(2) provides 32-bit values representing a space or a pulse duration. The time of the duration (microseconds) is encoded in the lower 24 bits. Pulse (also known as flash) indicates a duration of infrared light being detected, and space (also known as gap) indicates a duration with no infrared. If the duration of space exceeds the inactivity timeout, a special timeout package is delivered, which marks the end of a message. The upper 8 bits indicate the type of package:

LIRC_MODE2_SPACE
Value reflects a space duration (microseconds).

LIRC_MODE2_PULSE
Value reflects a pulse duration (microseconds).

LIRC_MODE2_FREQUENCY
Value reflects a frequency (Hz); see the **LIRC_SET_MEASURE_CARRIER_MODE** ioctl.

LIRC_MODE2_TIMEOUT
Value reflects a space duration (microseconds). The package reflects a timeout; see the **LIRC_SET_REC_TIMEOUT_REPORTS** ioctl.

LIRC_MODE2_OVERFLOW
The IR receiver encountered an overflow, and as a result data is missing (since Linux 5.18).

Reading input with the LIRC_MODE_SCANCODE mode

In the **LIRC_MODE_SCANCODE** mode, the data returned by *read*(2) reflects decoded button presses, in the struct *lirc_scancode*. The scancode is stored in the *scancode* field, and the IR protocol is stored in *rc_proto*. This field has one the values of the *enum rc_proto*.

Writing output with the LIRC_MODE_PULSE mode

The data written to the character device using *write*(2) is a pulse/space sequence of integer values. Pulses and spaces are only marked implicitly by their position. The data must start and end with a pulse, thus it must always include an odd number of samples. The *write*(2) function blocks until the data has been transmitted by the hardware. If more data is provided than the hardware can send, the *write*(2) call fails with the error **EINVAL**.

Writing output with the LIRC_MODE_SCANCODE mode

The data written to the character devices must be a single struct *lirc_scancode*. The *scancode* and *rc_proto* fields must filled in, all other fields must be 0. The kernel IR encoders will convert the scancode to pulses and spaces. The protocol or scancode is invalid, or the **lirc** device cannot transmit.

IOCTL COMMANDS

#include <linux/lirc.h> /* But see BUGS */

int ioctl(int fd, int cmd, int *val);

The following *ioctl*(2) operations are provided by the **lirc** character device to probe or change specific **lirc** hardware settings.

Always Supported Commands

*/dev/lirc** devices always support the following commands:

LIRC_GET_FEATURES (*void*)

Returns a bit mask of combined features bits; see FEATURES.

If a device returns an error code for **LIRC_GET_FEATURES**, it is safe to assume it is not a **lirc** device.

Optional Commands

Some **lirc** devices support the commands listed below. Unless otherwise stated, these fail with the error **ENOTTY** if the operation isn't supported, or with the error **EINVAL** if the operation failed, or invalid arguments were provided. If a driver does not announce support of certain features, invoking the corresponding ioctls will fail with the error **ENOTTY**.

LIRC_GET_REC_MODE (*void*)

If the **lirc** device has no receiver, this operation fails with the error **ENOTTY**. Otherwise, it returns the receive mode, which will be one of:

LIRC_MODE_MODE2

The driver returns a sequence of pulse/space durations.

LIRC_MODE_SCANCODE

The driver returns struct *lirc_scancode* values, each of which represents a decoded button press.

LIRC_SET_REC_MODE (*int*)

Set the receive mode. *val* is either **LIRC_MODE_SCANCODE** or **LIRC_MODE_MODE2**. If the **lirc** device has no receiver, this operation fails with the error **ENOTTY.**

LIRC_GET_SEND_MODE (*void*)

Return the send mode. **LIRC_MODE_PULSE** or **LIRC_MODE_SCANCODE** is supported. If the **lirc** device cannot send, this operation fails with the error **ENOTTY.**

LIRC_SET_SEND_MODE (*int*)

Set the send mode. *val* is either **LIRC_MODE_SCANCODE** or **LIRC_MODE_PULSE**. If the **lirc** device cannot send, this operation fails with the error **ENOTTY**.

LIRC_SET_SEND_CARRIER (*int*)

Set the modulation frequency. The argument is the frequency (Hz).

LIRC_SET_SEND_DUTY_CYCLE (*int*)

Set the carrier duty cycle. *val* is a number in the range [0,100] which describes the pulse width as a percentage of the total cycle. Currently, no special meaning is defined for 0 or 100, but the values are reserved for future use.

LIRC_GET_MIN_TIMEOUT(*void***)**
LIRC_GET_MAX_TIMEOUT(*void***)**

Some devices have internal timers that can be used to detect when there has been no IR activity for a long time. This can help *lircd*(8) in detecting that an IR signal is finished and can speed up the decoding process. These operations return integer values with the minimum/maximum timeout that can be set (microseconds). Some devices have a fixed timeout. For such drivers, **LIRC_GET_MIN_TIMEOUT** and **LIRC_GET_MAX_TIMEOUT** will fail with the error **ENOTTY**.

LIRC_SET_REC_TIMEOUT (*int*)

Set the integer value for IR inactivity timeout (microseconds). To be accepted, the value must be within the limits defined by **LIRC_GET_MIN_TIMEOUT** and **LIRC_GET_MAX_TIMEOUT**. A value of 0 (if supported by the hardware) disables all hardware timeouts and data should be reported as soon as possible. If the exact value cannot be set, then the next possible value *greater* than the given value should be set.

LIRC_GET_REC_TIMEOUT (*void*)
Return the current inactivity timeout (microseconds). Available since Linux 4.18.

LIRC_SET_REC_TIMEOUT_REPORTS (*int*)
Enable (*val* is 1) or disable (*val* is 0) timeout packages in **LIRC_MODE_MODE2**. The behavior of this operation has varied across kernel versions:

- Since Linux 5.17: timeout packages are always enabled and this ioctl is a no-op.
- Since Linux 4.16: timeout packages are enabled by default. Each time the **lirc** device is opened, the **LIRC_SET_REC_TIMEOUT** operation can be used to disable (and, if desired, to later re-enable) the timeout on the file descriptor.
- In Linux 4.15 and earlier: timeout packages are disabled by default, and enabling them (via **LIRC_SET_REC_TIMEOUT**) on any file descriptor associated with the **lirc** device has the effect of enabling timeouts for all file descriptors referring to that device (until timeouts are disabled again).

LIRC_SET_REC_CARRIER (*int*)
Set the upper bound of the receive carrier frequency (Hz). See **LIRC_SET_REC_CARRIER_RANGE**.

LIRC_SET_REC_CARRIER_RANGE (*int*)
Sets the lower bound of the receive carrier frequency (Hz). For this to take affect, first set the lower bound using the **LIRC_SET_REC_CARRIER_RANGE** ioctl, and then the upper bound using the **LIRC_SET_REC_CARRIER** ioctl.

LIRC_SET_MEASURE_CARRIER_MODE (*int*)
Enable (*val* is 1) or disable (*val* is 0) the measure mode. If enabled, from the next key press on, the driver will send **LIRC_MODE2_FREQUENCY** packets. By default, this should be turned off.

LIRC_GET_REC_RESOLUTION (*void*)
Return the driver resolution (microseconds).

LIRC_SET_TRANSMITTER_MASK (*int*)
Enable the set of transmitters specified in *val*, which contains a bit mask where each enabled transmitter is a 1. The first transmitter is encoded by the least significant bit, and so on. When an invalid bit mask is given, for example a bit is set even though the device does not have so many transmitters, this operation returns the number of available transmitters and does nothing otherwise.

LIRC_SET_WIDEBAND_RECEIVER (*int*)
Some devices are equipped with a special wide band receiver which is intended to be used to learn the output of an existing remote. This ioctl can be used to enable (*val* equals 1) or disable (*val* equals 0) this functionality. This might be useful for devices that otherwise have narrow band receivers that prevent them to be used with certain remotes. Wide band receivers may also be more precise. On the other hand, their disadvantage usually is reduced range of reception.

Note: wide band receiver may be implicitly enabled if you enable carrier reports. In that case, it will be disabled as soon as you disable carrier reports. Trying to disable a wide band receiver while carrier reports are active will do nothing.

FEATURES

the **LIRC_GET_FEATURES** ioctl returns a bit mask describing features of the driver. The following bits may be returned in the mask:

LIRC_CAN_REC_MODE2
The driver is capable of receiving using **LIRC_MODE_MODE2**.

LIRC_CAN_REC_SCANCODE
The driver is capable of receiving using **LIRC_MODE_SCANCODE**.

LIRC_CAN_SET_SEND_CARRIER
The driver supports changing the modulation frequency using **LIRC_SET_SEND_CARRIER**.

LIRC_CAN_SET_SEND_DUTY_CYCLE
The driver supports changing the duty cycle using **LIRC_SET_SEND_DUTY_CYCLE**.

LIRC_CAN_SET_TRANSMITTER_MASK
The driver supports changing the active transmitter(s) using **LIRC_SET_TRANSMITTER_MASK**.

LIRC_CAN_SET_REC_CARRIER
The driver supports setting the receive carrier frequency using **LIRC_SET_REC_CARRIER**. Any **lirc** device since the drivers were merged in Linux 2.6.36 must have **LIRC_CAN_SET_REC_CARRIER_RANGE** set if **LIRC_CAN_SET_REC_CARRIER** feature is set.

LIRC_CAN_SET_REC_CARRIER_RANGE
The driver supports **LIRC_SET_REC_CARRIER_RANGE**. The lower bound of the carrier must first be set using the **LIRC_SET_REC_CARRIER_RANGE** ioctl, before using the **LIRC_SET_REC_CARRIER** ioctl to set the upper bound.

LIRC_CAN_GET_REC_RESOLUTION
The driver supports **LIRC_GET_REC_RESOLUTION**.

LIRC_CAN_SET_REC_TIMEOUT
The driver supports **LIRC_SET_REC_TIMEOUT**.

LIRC_CAN_MEASURE_CARRIER
The driver supports measuring of the modulation frequency using **LIRC_SET_MEASURE_CARRIER_MODE**.

LIRC_CAN_USE_WIDEBAND_RECEIVER
The driver supports learning mode using **LIRC_SET_WIDEBAND_RECEIVER**.

LIRC_CAN_SEND_PULSE
The driver supports sending using **LIRC_MODE_PULSE** or **LIRC_MODE_SCANCODE**

BUGS

Using these devices requires the kernel source header file *lirc.h*. This file is not available before Linux 4.6. Users of older kernels could use the file bundled in .

SEE ALSO

ir−ctl(1), **lircd**(8), **bpf**(2)

NAME

loop, loop-control − loop devices

SYNOPSIS

#include <linux/loop.h>

DESCRIPTION

The loop device is a block device that maps its data blocks not to a physical device such as a hard disk or optical disk drive, but to the blocks of a regular file in a filesystem or to another block device. This can be useful for example to provide a block device for a filesystem image stored in a file, so that it can be mounted with the *mount*(8) command. You could do

```
$ dd if=/dev/zero of=file.img bs=1MiB count=10
$ sudo losetup /dev/loop4 file.img
$ sudo mkfs -t ext4 /dev/loop4
$ sudo mkdir /myloopdev
$ sudo mount /dev/loop4 /myloopdev
```

See *losetup*(8) for another example.

A transfer function can be specified for each loop device for encryption and decryption purposes.

The following *ioctl*(2) operations are provided by the loop block device:

LOOP_SET_FD
Associate the loop device with the open file whose file descriptor is passed as the (third) *ioctl*(2) argument.

LOOP_CLR_FD
Disassociate the loop device from any file descriptor.

LOOP_SET_STATUS
Set the status of the loop device using the (third) *ioctl*(2) argument. This argument is a pointer to a *loop_info* structure, defined in <*linux/loop.h*> as:

```
struct loop_info {
    int           lo_number;      /* ioctl r/o */
    dev_t         lo_device;      /* ioctl r/o */
    unsigned long lo_inode;       /* ioctl r/o */
    dev_t         lo_rdevice;     /* ioctl r/o */
    int           lo_offset;
    int           lo_encrypt_type;
    int           lo_encrypt_key_size;  /* ioctl w/o */
    int           lo_flags;       /* ioctl r/w (r/o before
                                     Linux 2.6.25) */
    char          lo_name[LO_NAME_SIZE];
    unsigned char lo_encrypt_key[LO_KEY_SIZE];
                                  /* ioctl w/o */
    unsigned long lo_init[2];
    char          reserved[4];
};
```

The encryption type (*lo_encrypt_type*) should be one of **LO_CRYPT_NONE**, **LO_CRYPT_XOR**, **LO_CRYPT_DES**, **LO_CRYPT_FISH2**, **LO_CRYPT_BLOW**, **LO_CRYPT_CAST128**, **LO_CRYPT_IDEA**, **LO_CRYPT_DUMMY**, **LO_CRYPT_SKIPJACK**, or (since Linux 2.6.0) **LO_CRYPT_CRYPTOAPI**.

The *lo_flags* field is a bit mask that can include zero or more of the following:

LO_FLAGS_READ_ONLY
The loopback device is read-only.

LO_FLAGS_AUTOCLEAR (since Linux 2.6.25)
The loopback device will autodestruct on last close.

LO_FLAGS_PARTSCAN (since Linux 3.2)
Allow automatic partition scanning.

LO_FLAGS_DIRECT_IO (since Linux 4.10)
Use direct I/O mode to access the backing file.

The only *lo_flags* that can be modified by **LOOP_SET_STATUS** are **LO_FLAGS_AUTOCLEAR** and **LO_FLAGS_PARTSCAN**.

LOOP_GET_STATUS
Get the status of the loop device. The (third) *ioctl*(2) argument must be a pointer to a *struct loop_info*.

LOOP_CHANGE_FD (since Linux 2.6.5)
Switch the backing store of the loop device to the new file identified file descriptor specified in the (third) *ioctl*(2) argument, which is an integer. This operation is possible only if the loop device is read-only and the new backing store is the same size and type as the old backing store.

LOOP_SET_CAPACITY (since Linux 2.6.30)
Resize a live loop device. One can change the size of the underlying backing store and then use this operation so that the loop driver learns about the new size. This operation takes no argument.

LOOP_SET_DIRECT_IO (since Linux 4.10)
Set DIRECT I/O mode on the loop device, so that it can be used to open backing file. The (third) *ioctl*(2) argument is an unsigned long value. A nonzero represents direct I/O mode.

LOOP_SET_BLOCK_SIZE (since Linux 4.14)
Set the block size of the loop device. The (third) *ioctl*(2) argument is an unsigned long value. This value must be a power of two in the range [512,pagesize]; otherwise, an **EINVAL** error results.

LOOP_CONFIGURE (since Linux 5.8)
Setup and configure all loop device parameters in a single step using the (third) *ioctl*(2) argument. This argument is a pointer to a *loop_config* structure, defined in <*linux/loop.h*> as:

```
struct loop_config {
    __u32           fd;
    __u32           block_size;
    struct loop_info64  info;
    __u64           __reserved[8];
};
```

In addition to doing what **LOOP_SET_STATUS** can do, **LOOP_CONFIGURE** can also be used to do the following:

- set the correct block size immediately by setting *loop_config.block_size*;
- explicitly request direct I/O mode by setting **LO_FLAGS_DIRECT_IO** in *loop_config.info.lo_flags*; and
- explicitly request read-only mode by setting **LO_FLAGS_READ_ONLY** in *loop_config.info.lo_flags*.

Since Linux 2.6, there are two new *ioctl*(2) operations:

LOOP_SET_STATUS64
LOOP_GET_STATUS64
These are similar to **LOOP_SET_STATUS** and **LOOP_GET_STATUS** described above but use the *loop_info64* structure, which has some additional fields and a larger range for some other fields:

```
struct loop_info64 {
    uint64_t lo_device;           /* ioctl r/o */
    uint64_t lo_inode;            /* ioctl r/o */
    uint64_t lo_rdevice;          /* ioctl r/o */
    uint64_t lo_offset;
    uint64_t lo_sizelimit;  /* bytes, 0 == max available */
    uint32_t lo_number;           /* ioctl r/o */
    uint32_t lo_encrypt_type;
```

```
            uint32_t lo_encrypt_key_size; /* ioctl w/o */
            uint32_t lo_flags; i          /* ioctl r/w (r/o before
                                             Linux 2.6.25) */
            uint8_t  lo_file_name[LO_NAME_SIZE];
            uint8_t  lo_crypt_name[LO_NAME_SIZE];
            uint8_t  lo_encrypt_key[LO_KEY_SIZE]; /* ioctl w/o */
            uint64_t lo_init[2];
        };
```

/dev/loop-control

Since Linux 3.1, the kernel provides the */dev/loop−control* device, which permits an application to dynamically find a free device, and to add and remove loop devices from the system. To perform these operations, one first opens */dev/loop−control* and then employs one of the following *ioctl*(2) operations:

LOOP_CTL_GET_FREE

Allocate or find a free loop device for use. On success, the device number is returned as the result of the call. This operation takes no argument.

LOOP_CTL_ADD

Add the new loop device whose device number is specified as a long integer in the third *ioctl*(2) argument. On success, the device index is returned as the result of the call. If the device is already allocated, the call fails with the error **EEXIST**.

LOOP_CTL_REMOVE

Remove the loop device whose device number is specified as a long integer in the third *ioctl*(2) argument. On success, the device number is returned as the result of the call. If the device is in use, the call fails with the error **EBUSY**.

FILES

*/dev/loop**

The loop block special device files.

EXAMPLES

The program below uses the */dev/loop−control* device to find a free loop device, opens the loop device, opens a file to be used as the underlying storage for the device, and then associates the loop device with the backing store. The following shell session demonstrates the use of the program:

```
$ dd if=/dev/zero of=file.img bs=1MiB count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB) copied, 0.00609385 s, 1.7 GB/s
$ sudo ./mnt_loop file.img
loopname = /dev/loop5
```

Program source

```
#include <fcntl.h>
#include <linux/loop.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

int
main(int argc, char *argv[])
{
    int loopctlfd, loopfd, backingfile;
    long devnr;
    char loopname[4096];
```

```
        if (argc != 2) {
            fprintf(stderr, "Usage: %s backing-file\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        loopctlfd = open("/dev/loop-control", O_RDWR);
        if (loopctlfd == -1)
            errExit("open: /dev/loop-control");

        devnr = ioctl(loopctlfd, LOOP_CTL_GET_FREE);
        if (devnr == -1)
            errExit("ioctl-LOOP_CTL_GET_FREE");

        sprintf(loopname, "/dev/loop%ld", devnr);
        printf("loopname = %s\n", loopname);

        loopfd = open(loopname, O_RDWR);
        if (loopfd == -1)
            errExit("open: loopname");

        backingfile = open(argv[1], O_RDWR);
        if (backingfile == -1)
            errExit("open: backing-file");

        if (ioctl(loopfd, LOOP_SET_FD, backingfile) == -1)
            errExit("ioctl-LOOP_SET_FD");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

losetup(8), *mount*(8)

NAME

lp − line printer devices

SYNOPSIS

#include <linux/lp.h>

CONFIGURATION

lp[0–2] are character devices for the parallel line printers; they have major number 6 and minor number 0–2. The minor numbers correspond to the printer port base addresses 0x03bc, 0x0378, and 0x0278. Usually they have mode 220 and are owned by user *root* and group *lp*. You can use printer ports either with polling or with interrupts. Interrupts are recommended when high traffic is expected, for example, for laser printers. For typical dot matrix printers, polling will usually be enough. The default is polling.

DESCRIPTION

The following *ioctl*(2) calls are supported:

int ioctl(int fd**, LPTIME, int** arg**)**

Sets the amount of time that the driver sleeps before rechecking the printer when the printer's buffer appears to be filled to *arg*. If you have a fast printer, decrease this number; if you have a slow printer, then increase it. This is in hundredths of a second, the default 2 being 0.02 seconds. It influences only the polling driver.

int ioctl(int fd**, LPCHAR, int** arg**)**

Sets the maximum number of busy-wait iterations which the polling driver does while waiting for the printer to get ready for receiving a character to *arg*. If printing is too slow, increase this number; if the system gets too slow, decrease this number. The default is 1000. It influences only the polling driver.

int ioctl(int fd**, LPABORT, int** arg**)**

If *arg* is 0, the printer driver will retry on errors, otherwise it will abort. The default is 0.

int ioctl(int fd**, LPABORTOPEN, int** arg**)**

If *arg* is 0, *open*(2) will be aborted on error, otherwise error will be ignored. The default is to ignore it.

int ioctl(int fd**, LPCAREFUL, int** arg**)**

If *arg* is 0, then the out-of-paper, offline, and error signals are required to be false on all writes, otherwise they are ignored. The default is to ignore them.

int ioctl(int fd**, LPWAIT, int** arg**)**

Sets the number of busy waiting iterations to wait before strobing the printer to accept a just-written character, and the number of iterations to wait before turning the strobe off again, to *arg*. The specification says this time should be 0.5 microseconds, but experience has shown the delay caused by the code is already enough. For that reason, the default value is 0. This is used for both the polling and the interrupt driver.

int ioctl(int fd**, LPSETIRQ, int** arg**)**

This *ioctl*(2) requires superuser privileges. It takes an *int* containing the new IRQ as argument. As a side effect, the printer will be reset. When *arg* is 0, the polling driver will be used, which is also default.

int ioctl(int fd**, LPGETIRQ, int ***arg**)**

Stores the currently used IRQ in *arg*.

int ioctl(int fd**, LPGETSTATUS, int ***arg**)**

Stores the value of the status port in *arg*. The bits have the following meaning:

LP_PBUSY	inverted busy input, active high
LP_PACK	unchanged acknowledge input, active low
LP_POUTPA	unchanged out-of-paper input, active high
LP_PSELECD	unchanged selected input, active high
LP_PERRORP	unchanged error input, active low

Refer to your printer manual for the meaning of the signals. Note that undocumented bits may also be set, depending on your printer.

int ioctl(int fd**, LPRESET)**
Resets the printer. No argument is used.

FILES

*/dev/lp**

SEE ALSO

chmod(1), *chown*(1), *mknod*(1), *lpcntl*(8), *tunelp*(8)

NAME

mem, kmem, port − system memory, kernel memory and system ports

DESCRIPTION

/dev/mem is a character device file that is an image of the main memory of the computer. It may be used, for example, to examine (and even patch) the system.

Byte addresses in */dev/mem* are interpreted as physical memory addresses. References to nonexistent locations cause errors to be returned.

Examining and patching is likely to lead to unexpected results when read-only or write-only bits are present.

Since Linux 2.6.26, and depending on the architecture, the **CONFIG_STRICT_DEVMEM** kernel configuration option limits the areas which can be accessed through this file. For example: on x86, RAM access is not allowed but accessing memory-mapped PCI regions is.

It is typically created by:

```
mknod −m 660 /dev/mem c 1 1
chown root:kmem /dev/mem
```

The file */dev/kmem* is the same as */dev/mem*, except that the kernel virtual memory rather than physical memory is accessed. Since Linux 2.6.26, this file is available only if the **CONFIG_DEVKMEM** kernel configuration option is enabled.

It is typically created by:

```
mknod −m 640 /dev/kmem c 1 2
chown root:kmem /dev/kmem
```

/dev/port is similar to */dev/mem*, but the I/O ports are accessed.

It is typically created by:

```
mknod −m 660 /dev/port c 1 4
chown root:kmem /dev/port
```

FILES

/dev/mem
/dev/kmem
/dev/port

SEE ALSO

chown(1), *mknod*(1), *ioperm*(2)

NAME

mouse − serial mouse interface

CONFIGURATION

Serial mice are connected to a serial RS232/V24 dialout line, see *ttyS*(4) for a description.

DESCRIPTION

Introduction

The pinout of the usual 9 pin plug as used for serial mice is:

pin	name	used for
2	RX	Data
3	TX	−12 V, Imax = 10 mA
4	DTR	+12 V, Imax = 10 mA
7	RTS	+12 V, Imax = 10 mA
5	GND	Ground

This is the specification, in fact 9 V suffices with most mice.

The mouse driver can recognize a mouse by dropping RTS to low and raising it again. About 14 ms later the mouse will send 0x4D ('M') on the data line. After a further 63 ms, a Microsoft-compatible 3-button mouse will send 0x33 ('3').

The relative mouse movement is sent as *dx* (positive means right) and *dy* (positive means down). Various mice can operate at different speeds. To select speeds, cycle through the speeds 9600, 4800, 2400, and 1200 bit/s, each time writing the two characters from the table below and waiting 0.1 seconds. The following table shows available speeds and the strings that select them:

bit/s	string
9600	*q
4800	*p
2400	*o
1200	*n

The first byte of a data packet can be used for synchronization purposes.

Microsoft protocol

The **Microsoft** protocol uses 1 start bit, 7 data bits, no parity and one stop bit at the speed of 1200 bits/sec. Data is sent to RxD in 3-byte packets. The *dx* and *dy* movements are sent as two's-complement, *lb* (*rb*) are set when the left (right) button is pressed:

byte	d6	d5	d4	d3	d2	d1	d0
1	1	lb	rb	dy7	dy6	dx7	dx6
2	0	dx5	dx4	dx3	dx2	dx1	dx0
3	0	dy5	dy4	dy3	dy2	dy1	dy0

3-button Microsoft protocol

Original Microsoft mice only have two buttons. However, there are some three button mice which also use the Microsoft protocol. Pressing or releasing the middle button is reported by sending a packet with zero movement and no buttons pressed. (Thus, unlike for the other two buttons, the status of the middle button is not reported in each packet.)

Logitech protocol

Logitech serial 3-button mice use a different extension of the Microsoft protocol: when the middle button is up, the above 3-byte packet is sent. When the middle button is down a 4-byte packet is sent, where the 4th byte has value 0x20 (or at least has the 0x20 bit set). In particular, a press of the middle button is reported as 0,0,0,0x20 when no other buttons are down.

Mousesystems protocol

The **Mousesystems** protocol uses 1 start bit, 8 data bits, no parity, and two stop bits at the speed of 1200 bits/sec. Data is sent to RxD in 5-byte packets. *dx* is sent as the sum of the two two's-complement values, *dy* is send as negated sum of the two two's-complement values. *lb* (*mb*, *rb*) are cleared when the left (middle, right) button is pressed:

byte	d7	d6	d5	d4	d3	d2	d1	d0
1	1	0	0	0	0	lb	mb	rb

2	0	dxa6	dxa5	dxa4	dxa3	dxa2	dxa1	dxa0
3	0	dya6	dya5	dya4	dya3	dya2	dya1	dya0
4	0	dxb6	dxb5	dxb4	dxb3	dxb2	dxb1	dxb0
5	0	dyb6	dyb5	dyb4	dyb3	dyb2	dyb1	dyb0

Bytes 4 and 5 describe the change that occurred since bytes 2 and 3 were transmitted.

Sun protocol

The **Sun** protocol is the 3-byte version of the above 5-byte Mousesystems protocol: the last two bytes are not sent.

MM protocol

The **MM** protocol uses 1 start bit, 8 data bits, odd parity, and one stop bit at the speed of 1200 bits/sec. Data is sent to RxD in 3-byte packets. *dx* and *dy* are sent as single signed values, the sign bit indicating a negative value. *lb* (*mb*, *rb*) are set when the left (middle, right) button is pressed:

byte	d7	d6	d5	d4	d3	d2	d1	d0
1	1	0	0	dxs	dys	lb	mb	rb
2	0	dx6	dx5	dx4	dx3	dx2	dx1	dx0
3	0	dy6	dy5	dy4	dy3	dy2	dy1	dy0

FILES

/dev/mouse

A commonly used symbolic link pointing to a mouse device.

SEE ALSO

ttyS(4), *gpm*(8)

NAME

msr − x86 CPU MSR access device

DESCRIPTION

/dev/cpu/CPUNUM/msr provides an interface to read and write the model-specific registers (MSRs) of an x86 CPU. *CPUNUM* is the number of the CPU to access as listed in */proc/cpuinfo*.

The register access is done by opening the file and seeking to the MSR number as offset in the file, and then reading or writing in chunks of 8 bytes. An I/O transfer of more than 8 bytes means multiple reads or writes of the same register.

This file is protected so that it can be read and written only by the user *root*, or members of the group *root*.

NOTES

The *msr* driver is not auto-loaded. On modular kernels you might need to use the following command to load it explicitly before use:

```
$ modprobe msr
```

SEE ALSO

Intel Corporation Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B Appendix B, for an overview of the Intel CPU MSRs.

NAME

null, zero − data sink

DESCRIPTION

Data written to the */dev/null* and */dev/zero* special files is discarded.

Reads from */dev/null* always return end of file (i.e., *read*(2) returns 0), whereas reads from */dev/zero* always return bytes containing zero ('\0' characters).

These devices are typically created by:

```
mknod −m 666 /dev/null c 1 3
mknod −m 666 /dev/zero c 1 5
chown root:root /dev/null /dev/zero
```

FILES

/dev/null
/dev/zero

NOTES

If these devices are not writable and readable for all users, many programs will act strangely.

Since Linux 2.6.31, reads from */dev/zero* are interruptible by signals. (This change was made to help with bad latencies for large reads from */dev/zero*.)

SEE ALSO

chown(1), *mknod*(1), *full*(4)

NAME

ptmx, pts − pseudoterminal master and slave

DESCRIPTION

The file */dev/ptmx* (the pseudoterminal multiplexor device) is a character file with major number 5 and minor number 2, usually with mode 0666 and ownership root:root. It is used to create a pseudoterminal master and slave pair.

When a process opens */dev/ptmx*, it gets a file descriptor for a pseudoterminal master and a pseudoterminal slave device is created in the */dev/pts* directory. Each file descriptor obtained by opening */dev/ptmx* is an independent pseudoterminal master with its own associated slave, whose path can be found by passing the file descriptor to *ptsname*(3).

Before opening the pseudoterminal slave, you must pass the master's file descriptor to *grantpt*(3) and *unlockpt*(3).

Once both the pseudoterminal master and slave are open, the slave provides processes with an interface that is identical to that of a real terminal.

Data written to the slave is presented on the master file descriptor as input. Data written to the master is presented to the slave as input.

In practice, pseudoterminals are used for implementing terminal emulators such as *xterm*(1), in which data read from the pseudoterminal master is interpreted by the application in the same way a real terminal would interpret the data, and for implementing remote-login programs such as *sshd*(8), in which data read from the pseudoterminal master is sent across the network to a client program that is connected to a terminal or terminal emulator.

Pseudoterminals can also be used to send input to programs that normally refuse to read input from pipes (such as *su*(1), and *passwd*(1)).

FILES

/dev/ptmx, */dev/pts/**

NOTES

The Linux support for the above (known as UNIX 98 pseudoterminal naming) is done using the *devpts* filesystem, which should be mounted on */dev/pts*.

SEE ALSO

getpt(3), *grantpt*(3), *ptsname*(3), *unlockpt*(3), *pty*(7)

NAME

ram − ram disk device

DESCRIPTION

The *ram* device is a block device to access the ram disk in raw mode.

It is typically created by:

```
mknod −m 660 /dev/ram b 1 1
chown root:disk /dev/ram
```

FILES

/dev/ram

SEE ALSO

chown(1), *mknod*(1), *mount*(8)

NAME

random, urandom − kernel random number source devices

SYNOPSIS

#include <linux/random.h>

int ioctl(*fd***, RND***request***,** *param***);**

DESCRIPTION

The character special files */dev/random* and */dev/urandom* (present since Linux 1.3.30) provide an interface to the kernel's random number generator. The file */dev/random* has major device number 1 and minor device number 8. The file */dev/urandom* has major device number 1 and minor device number 9.

The random number generator gathers environmental noise from device drivers and other sources into an entropy pool. The generator also keeps an estimate of the number of bits of noise in the entropy pool. From this entropy pool, random numbers are created.

Linux 3.17 and later provides the simpler and safer *getrandom*(2) interface which requires no special files; see the *getrandom*(2) manual page for details.

When read, the */dev/urandom* device returns random bytes using a pseudorandom number generator seeded from the entropy pool. Reads from this device do not block (i.e., the CPU is not yielded), but can incur an appreciable delay when requesting large amounts of data.

When read during early boot time, */dev/urandom* may return data prior to the entropy pool being initialized. If this is of concern in your application, use *getrandom*(2) or */dev/random* instead.

The */dev/random* device is a legacy interface which dates back to a time where the cryptographic primitives used in the implementation of */dev/urandom* were not widely trusted. It will return random bytes only within the estimated number of bits of fresh noise in the entropy pool, blocking if necessary. */dev/random* is suitable for applications that need high quality randomness, and can afford indeterminate delays.

When the entropy pool is empty, reads from */dev/random* will block until additional environmental noise is gathered. Since Linux 5.6, the **O_NONBLOCK** flag is ignored as */dev/random* will no longer block except during early boot process. In earlier versions, if *open*(2) is called for */dev/random* with the **O_NONBLOCK** flag, a subsequent *read*(2) will not block if the requested number of bytes is not available. Instead, the available bytes are returned. If no byte is available, *read*(2) will return −1 and *errno* will be set to **EAGAIN**.

The **O_NONBLOCK** flag has no effect when opening */dev/urandom*. When calling *read*(2) for the device */dev/urandom*, reads of up to 256 bytes will return as many bytes as are requested and will not be interrupted by a signal handler. Reads with a buffer over this limit may return less than the requested number of bytes or fail with the error **EINTR**, if interrupted by a signal handler.

Since Linux 3.16, a *read*(2) from */dev/urandom* will return at most 32 MB. A *read*(2) from */dev/random* will return at most 512 bytes (340 bytes before Linux 2.6.12).

Writing to */dev/random* or */dev/urandom* will update the entropy pool with the data written, but this will not result in a higher entropy count. This means that it will impact the contents read from both files, but it will not make reads from */dev/random* faster.

Usage

The */dev/random* interface is considered a legacy interface, and */dev/urandom* is preferred and sufficient in all use cases, with the exception of applications which require randomness during early boot time; for these applications, *getrandom*(2) must be used instead, because it will block until the entropy pool is initialized.

If a seed file is saved across reboots as recommended below, the output is cryptographically secure against attackers without local root access as soon as it is reloaded in the boot sequence, and perfectly adequate for network encryption session keys. (All major Linux distributions have saved the seed file across reboots since 2000 at least.) Since reads from */dev/random* may block, users will usually want to open it in nonblocking mode (or perform a read with timeout), and provide some sort of user notification if the desired entropy is not immediately available.

Configuration

If your system does not have */dev/random* and */dev/urandom* created already, they can be created with the following commands:

```
mknod −m 666 /dev/random c 1 8
mknod −m 666 /dev/urandom c 1 9
chown root:root /dev/random /dev/urandom
```

When a Linux system starts up without much operator interaction, the entropy pool may be in a fairly predictable state. This reduces the actual amount of noise in the entropy pool below the estimate. In order to counteract this effect, it helps to carry entropy pool information across shut-downs and start-ups. To do this, add the lines to an appropriate script which is run during the Linux system start-up sequence:

```
echo "Initializing random number generator..."
random_seed=/var/run/random−seed
# Carry a random seed from start−up to start−up
# Load and then save the whole entropy pool
if [ −f $random_seed ]; then
    cat $random_seed >/dev/urandom
else
    touch $random_seed
fi
chmod 600 $random_seed
poolfile=/proc/sys/kernel/random/poolsize
[ −r $poolfile ] && bits=$(cat $poolfile) || bits=4096
bytes=$(expr $bits / 8)
dd if=/dev/urandom of=$random_seed count=1 bs=$bytes
```

Also, add the following lines in an appropriate script which is run during the Linux system shutdown:

```
# Carry a random seed from shut−down to start−up
# Save the whole entropy pool
echo "Saving random seed..."
random_seed=/var/run/random−seed
touch $random_seed
chmod 600 $random_seed
poolfile=/proc/sys/kernel/random/poolsize
[ −r $poolfile ] && bits=$(cat $poolfile) || bits=4096
bytes=$(expr $bits / 8)
dd if=/dev/urandom of=$random_seed count=1 bs=$bytes
```

In the above examples, we assume Linux 2.6.0 or later, where */proc/sys/kernel/random/poolsize* returns the size of the entropy pool in bits (see below).

/proc interfaces

The files in the directory */proc/sys/kernel/random* (present since Linux 2.3.16) provide additional information about the */dev/random* device:

entropy_avail

This read-only file gives the available entropy, in bits. This will be a number in the range 0 to 4096.

poolsize

This file gives the size of the entropy pool. The semantics of this file vary across kernel versions:

Linux 2.4:

This file gives the size of the entropy pool in *bytes*. Normally, this file will have the value 512, but it is writable, and can be changed to any value for which an algorithm is available. The choices are 32, 64, 128, 256, 512, 1024, or 2048.

Linux 2.6 and later:

This file is read-only, and gives the size of the entropy pool in *bits*. It contains the value 4096.

read_wakeup_threshold
: This file contains the number of bits of entropy required for waking up processes that sleep waiting for entropy from */dev/random*. The default is 64.

write_wakeup_threshold
: This file contains the number of bits of entropy below which we wake up processes that do a *select*(2) or *poll*(2) for write access to */dev/random*. These values can be changed by writing to the files.

uuid and *boot_id*
: These read-only files contain random strings like 6fd5a44b−35f4−4ad4−a9b9−6b9be13e1fe9. The former is generated afresh for each read, the latter was generated once.

ioctl(2) interface

The following *ioctl*(2) requests are defined on file descriptors connected to either */dev/random* or */dev/urandom*. All requests performed will interact with the input entropy pool impacting both */dev/random* and */dev/urandom*. The **CAP_SYS_ADMIN** capability is required for all requests except **RNDGETENTCNT**.

RNDGETENTCNT
: Retrieve the entropy count of the input pool, the contents will be the same as the *entropy_avail* file under proc. The result will be stored in the int pointed to by the argument.

RNDADDTOENTCNT
: Increment or decrement the entropy count of the input pool by the value pointed to by the argument.

RNDGETPOOL
: Removed in Linux 2.6.9.

RNDADDENTROPY
: Add some additional entropy to the input pool, incrementing the entropy count. This differs from writing to */dev/random* or */dev/urandom*, which only adds some data but does not increment the entropy count. The following structure is used:

```
struct rand_pool_info {
    int    entropy_count;
    int    buf_size;
    __u32  buf[0];
};
```

Here *entropy_count* is the value added to (or subtracted from) the entropy count, and *buf* is the buffer of size *buf_size* which gets added to the entropy pool.

RNDZAPENTCNT
RNDCLEARPOOL
: Zero the entropy count of all pools and add some system data (such as wall clock) to the pools.

FILES

/dev/random
/dev/urandom

NOTES

For an overview and comparison of the various interfaces that can be used to obtain randomness, see *random*(7).

BUGS

During early boot time, reads from */dev/urandom* may return data prior to the entropy pool being initialized.

SEE ALSO

mknod(1), *getrandom*(2), *random*(7)

RFC 1750, "Randomness Recommendations for Security"

NAME

rtc − real-time clock

SYNOPSIS

#include <linux/rtc.h>

int ioctl(*fd*, **RTC_***request*, *param***);**

DESCRIPTION

This is the interface to drivers for real-time clocks (RTCs).

Most computers have one or more hardware clocks which record the current "wall clock" time. These are called "Real Time Clocks" (RTCs). One of these usually has battery backup power so that it tracks the time even while the computer is turned off. RTCs often provide alarms and other interrupts.

All i386 PCs, and ACPI-based systems, have an RTC that is compatible with the Motorola MC146818 chip on the original PC/AT. Today such an RTC is usually integrated into the mainboard's chipset (south bridge), and uses a replaceable coin-sized backup battery.

Non-PC systems, such as embedded systems built around system-on-chip processors, use other implementations. They usually won't offer the same functionality as the RTC from a PC/AT.

RTC vs system clock

RTCs should not be confused with the system clock, which is a software clock maintained by the kernel and used to implement *gettimeofday*(2) and *time*(2), as well as setting timestamps on files, and so on. The system clock reports seconds and microseconds since a start point, defined to be the POSIX Epoch: 1970-01-01 00:00:00 +0000 (UTC). (One common implementation counts timer interrupts, once per "jiffy", at a frequency of 100, 250, or 1000 Hz.) That is, it is supposed to report wall clock time, which RTCs also do.

A key difference between an RTC and the system clock is that RTCs run even when the system is in a low power state (including "off"), and the system clock can't. Until it is initialized, the system clock can only report time since system boot ... not since the POSIX Epoch. So at boot time, and after resuming from a system low power state, the system clock will often be set to the current wall clock time using an RTC. Systems without an RTC need to set the system clock using another clock, maybe across the network or by entering that data manually.

RTC functionality

RTCs can be read and written with *hwclock*(8), or directly with the *ioctl*(2) requests listed below.

Besides tracking the date and time, many RTCs can also generate interrupts

- on every clock update (i.e., once per second);
- at periodic intervals with a frequency that can be set to any power-of-2 multiple in the range 2 Hz to 8192 Hz;
- on reaching a previously specified alarm time.

Each of those interrupt sources can be enabled or disabled separately. On many systems, the alarm interrupt can be configured as a system wakeup event, which can resume the system from a low power state such as Suspend-to-RAM (STR, called S3 in ACPI systems), Hibernation (called S4 in ACPI systems), or even "off" (called S5 in ACPI systems). On some systems, the battery backed RTC can't issue interrupts, but another one can.

The */dev/rtc* (or */dev/rtc0*, */dev/rtc1*, etc.) device can be opened only once (until it is closed) and it is read-only. On *read*(2) and *select*(2) the calling process is blocked until the next interrupt from that RTC is received. Following the interrupt, the process can read a long integer, of which the least significant byte contains a bit mask encoding the types of interrupt that occurred, while the remaining 3 bytes contain the number of interrupts since the last *read*(2).

ioctl(2) interface

The following *ioctl*(2) requests are defined on file descriptors connected to RTC devices:

RTC_RD_TIME

Returns this RTC's time in the following structure:

```
struct rtc_time {
    int tm_sec;
```

```
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;     /* unused */
        int tm_yday;     /* unused */
        int tm_isdst;    /* unused */
    };
```

The fields in this structure have the same meaning and ranges as for the *tm* structure described in *gmtime*(3). A pointer to this structure should be passed as the third *ioctl*(2) argument.

RTC_SET_TIME
: Sets this RTC's time to the time specified by the *rtc_time* structure pointed to by the third *ioctl*(2) argument. To set the RTC's time the process must be privileged (i.e., have the **CAP_SYS_TIME** capability).

RTC_ALM_READ
RTC_ALM_SET
: Read and set the alarm time, for RTCs that support alarms. The alarm interrupt must be separately enabled or disabled using the **RTC_AIE_ON**, **RTC_AIE_OFF** requests. The third *ioctl*(2) argument is a pointer to an *rtc_time* structure. Only the *tm_sec*, *tm_min*, and *tm_hour* fields of this structure are used.

RTC_IRQP_READ
RTC_IRQP_SET
: Read and set the frequency for periodic interrupts, for RTCs that support periodic interrupts. The periodic interrupt must be separately enabled or disabled using the **RTC_PIE_ON**, **RTC_PIE_OFF** requests. The third *ioctl*(2) argument is an *unsigned long* * or an *unsigned long*, respectively. The value is the frequency in interrupts per second. The set of allowable frequencies is the multiples of two in the range 2 to 8192. Only a privileged process (i.e., one having the **CAP_SYS_RESOURCE** capability) can set frequencies above the value specified in */proc/sys/dev/rtc/max−user−freq*. (This file contains the value 64 by default.)

RTC_AIE_ON
RTC_AIE_OFF
: Enable or disable the alarm interrupt, for RTCs that support alarms. The third *ioctl*(2) argument is ignored.

RTC_UIE_ON
RTC_UIE_OFF
: Enable or disable the interrupt on every clock update, for RTCs that support this once-per-second interrupt. The third *ioctl*(2) argument is ignored.

RTC_PIE_ON
RTC_PIE_OFF
: Enable or disable the periodic interrupt, for RTCs that support these periodic interrupts. The third *ioctl*(2) argument is ignored. Only a privileged process (i.e., one having the **CAP_SYS_RESOURCE** capability) can enable the periodic interrupt if the frequency is currently set above the value specified in */proc/sys/dev/rtc/max−user−freq*.

RTC_EPOCH_READ
RTC_EPOCH_SET
: Many RTCs encode the year in an 8-bit register which is either interpreted as an 8-bit binary number or as a BCD number. In both cases, the number is interpreted relative to this RTC's Epoch. The RTC's Epoch is initialized to 1900 on most systems but on Alpha and MIPS it might also be initialized to 1952, 1980, or 2000, depending on the value of an RTC register for the year. With some RTCs, these operations can be used to read or to set the RTC's Epoch, respectively. The third *ioctl*(2) argument is an *unsigned long* * or an *unsigned long*, respectively, and the value returned (or assigned) is the Epoch. To set the RTC's Epoch the process must be privileged (i.e., have the **CAP_SYS_TIME** capability).

RTC_WKALM_RD
RTC_WKALM_SET
Some RTCs support a more powerful alarm interface, using these ioctls to read or write the RTC's alarm time (respectively) with this structure:

```
struct rtc_wkalrm {
    unsigned char enabled;
    unsigned char pending;
    struct rtc_time time;
};
```

The *enabled* flag is used to enable or disable the alarm interrupt, or to read its current status; when using these calls, **RTC_AIE_ON** and **RTC_AIE_OFF** are not used. The *pending* flag is used by **RTC_WKALM_RD** to report a pending interrupt (so it's mostly useless on Linux, except when talking to the RTC managed by EFI firmware). The *time* field is as used with **RTC_ALM_READ** and **RTC_ALM_SET** except that the *tm_mday*, *tm_mon*, and *tm_year* fields are also valid. A pointer to this structure should be passed as the third *ioctl*(2) argument.

FILES

/dev/rtc
/dev/rtc0
/dev/rtc1
. . . RTC special character device files.

/proc/driver/rtc
status of the (first) RTC.

NOTES

When the kernel's system time is synchronized with an external reference using *adjtimex*(2) it will update a designated RTC periodically every 11 minutes. To do so, the kernel has to briefly turn off periodic interrupts; this might affect programs using that RTC.

An RTC's Epoch has nothing to do with the POSIX Epoch which is used only for the system clock.

If the year according to the RTC's Epoch and the year register is less than 1970 it is assumed to be 100 years later, that is, between 2000 and 2069.

Some RTCs support "wildcard" values in alarm fields, to support scenarios like periodic alarms at fifteen minutes after every hour, or on the first day of each month. Such usage is nonportable; portable user-space code expects only a single alarm interrupt, and will either disable or reinitialize the alarm after receiving it.

Some RTCs support periodic interrupts with periods that are multiples of a second rather than fractions of a second; multiple alarms; programmable output clock signals; nonvolatile memory; and other hardware capabilities that are not currently exposed by this API.

SEE ALSO

date(1), *adjtimex*(2), *gettimeofday*(2), *settimeofday*(2), *stime*(2), *time*(2), *gmtime*(3), *time*(7), *hwclock*(8)

Documentation/rtc.txt in the Linux kernel source tree

NAME

sd − driver for SCSI disk drives

SYNOPSIS

#include <linux/hdreg.h> /* for HDIO_GETGEO */
#include <linux/fs.h> /* for BLKGETSIZE and BLKRRPART */

CONFIGURATION

The block device name has the following form: **sd***lp*, where *l* is a letter denoting the physical drive, and *p* is a number denoting the partition on that physical drive. Often, the partition number, *p*, will be left off when the device corresponds to the whole drive.

SCSI disks have a major device number of 8, and a minor device number of the form (16 * *drive_number*) + *partition_number*, where *drive_number* is the number of the physical drive in order of detection, and *partition_number* is as follows:

- partition 0 is the whole drive
- partitions 1–4 are the DOS "primary" partitions
- partitions 5–8 are the DOS "extended" (or "logical") partitions

For example, */dev/sda* will have major 8, minor 0, and will refer to all of the first SCSI drive in the system; and */dev/sdb3* will have major 8, minor 19, and will refer to the third DOS "primary" partition on the second SCSI drive in the system.

At this time, only block devices are provided. Raw devices have not yet been implemented.

DESCRIPTION

The following *ioctl*s are provided:

HDIO_GETGEO

Returns the BIOS disk parameters in the following structure:

```
struct hd_geometry {
    unsigned char  heads;
    unsigned char  sectors;
    unsigned short cylinders;
    unsigned long  start;
};
```

A pointer to this structure is passed as the *ioctl*(2) parameter.

The information returned in the parameter is the disk geometry of the drive *as understood by DOS!* This geometry is *not* the physical geometry of the drive. It is used when constructing the drive's partition table, however, and is needed for convenient operation of *fdisk*(1), *efdisk*(1), and *lilo*(1)If the geometry information is not available, zero will be returned for all of the parameters.

BLKGETSIZE

Returns the device size in sectors. The *ioctl*(2) parameter should be a pointer to a *long*.

BLKRRPART

Forces a reread of the SCSI disk partition tables. No parameter is needed.

The SCSI *ioctl*(2) operations are also supported. If the *ioctl*(2) parameter is required, and it is NULL, then *ioctl*(2) fails with the error **EINVAL**.

FILES

/dev/sd[a−h]
the whole device

/dev/sd[a−h][0−8]
individual block partitions

NAME

sk98lin − Marvell/SysKonnect Gigabit Ethernet driver v6.21

SYNOPSIS

insmod sk98lin.o [**Speed_A**=*i*,*j*,...] [**Speed_B**=*i*,*j*,...] [**AutoNeg_A**=*i*,*j*,...] [**AutoNeg_B**=*i*,*j*,...] [**DupCap_A**=*i*,*j*,...] [**DupCap_B**=*i*,*j*,...] [**FlowCtrl_A**=*i*,*j*,...] [**FlowCtrl_B**=*i*,*j*,...] [**Role_A**=*i*,*j*,...] [**Role_B**=*i*,*j*,...] [**ConType**=*i*,*j*,...] [**Moderation**=*i*,*j*,...] [**IntsPerSec**=*i*,*j*,...] [**PrefPort**=*i*,*j*,...] [**RlmtMode**=*i*,*j*,...]

DESCRIPTION

Note: This obsolete driver was removed in Linux 2.6.26.

sk98lin is the Gigabit Ethernet driver for Marvell and SysKonnect network adapter cards. It supports SysKonnect SK-98xx/SK-95xx compliant Gigabit Ethernet Adapter and any Yukon compliant chipset.

When loading the driver using insmod, parameters for the network adapter cards might be stated as a sequence of comma separated commands. If for instance two network adapters are installed and AutoNegotiation on Port A of the first adapter should be ON, but on the Port A of the second adapter switched OFF, one must enter:

```
insmod sk98lin.o AutoNeg_A=On,Off
```

After **sk98lin** is bound to one or more adapter cards and the */proc* filesystem is mounted on your system, a dedicated statistics file will be created in the folder */proc/net/sk98lin* for all ports of the installed network adapter cards. Those files are named *eth[x]*, where *x* is the number of the interface that has been assigned to a dedicated port by the system.

If loading is finished, any desired IP address can be assigned to the respective *eth[x]* interface using the *ifconfig*(8) command. This causes the adapter to connect to the Ethernet and to display a status message on the console saying "ethx: network connection up using port y" followed by the configured or detected connection parameters.

The **sk98lin** also supports large frames (also called jumbo frames). Using jumbo frames can improve throughput tremendously when transferring large amounts of data. To enable large frames, the MTU (maximum transfer unit) size for an interface is to be set to a high value. The default MTU size is 1500 and can be changed up to 9000 (bytes). Setting the MTU size can be done when assigning the IP address to the interface or later by using the *ifconfig*(8) command with the mtu parameter. If for instance eth0 needs an IP address and a large frame MTU size, the following two commands might be used:

```
ifconfig eth0 10.1.1.1
ifconfig eth0 mtu 9000
```

Those two commands might even be combined into one:

```
ifconfig eth0 10.1.1.1 mtu 9000
```

Note that large frames can be used only if permitted by your network infrastructure. This means, that any switch being used in your Ethernet must also support large frames. Quite some switches support large frames, but need to be configured to do so. Most of the times, their default setting is to support only standard frames with an MTU size of 1500 (bytes). In addition to the switches inside the network, all network adapters that are to be used must also be enabled regarding jumbo frames. If an adapter is not set to receive large frames, it will simply drop them.

Switching back to the standard Ethernet frame size can be done by using the *ifconfig*(8) command again:

```
ifconfig eth0 mtu 1500
```

The Marvell/SysKonnect Gigabit Ethernet driver for Linux is able to support VLAN and Link Aggregation according to IEEE standards 802.1, 802.1q, and 802.3ad. Those features are available only after installation of open source modules which can be found on the Internet:

VLAN:
Link Aggregation:

Note that Marvell/SysKonnect does not offer any support for these open source modules and does not take the responsibility for any kind of failures or problems arising when using these modules.

Parameters

Speed_A=*i,j,...*
This parameter is used to set the speed capabilities of port A of an adapter card. It is valid only for Yukon copper adapters. Possible values are: *10*, *100*, *1000*, or *Auto*; *Auto* is the default. Usually, the speed is negotiated between the two ports during link establishment. If this fails, a port can be forced to a specific setting with this parameter.

Speed_B=*i,j,...*
This parameter is used to set the speed capabilities of port B of an adapter card. It is valid only for Yukon copper adapters. Possible values are: *10*, *100*, *1000*, or *Auto*; *Auto* is the default. Usually, the speed is negotiated between the two ports during link establishment. If this fails, a port can be forced to a specific setting with this parameter.

AutoNeg_A=*i,j,...*
Enables or disables the use of autonegotiation of port A of an adapter card. Possible values are: *On*, *Off*, or *Sense*; *On* is the default. The *Sense* mode automatically detects whether the link partner supports auto-negotiation or not.

AutoNeg_B=*i,j,...*
Enables or disables the use of autonegotiation of port B of an adapter card. Possible values are: *On*, *Off*, or *Sense*; *On* is the default. The *Sense* mode automatically detects whether the link partner supports auto-negotiation or not.

DupCap_A=*i,j,...*
This parameter indicates the duplex mode to be used for port A of an adapter card. Possible values are: *Half*, *Full*, or *Both*; *Both* is the default. This parameter is relevant only if AutoNeg_A of port A is not set to *Sense*. If AutoNeg_A is set to *On*, all three values of DupCap_A (*Half*, *Full*, or *Both*) might be stated. If AutoNeg_A is set to *Off*, only DupCap_A values *Full* and *Half* are allowed. This DupCap_A parameter is useful if your link partner does not support all possible duplex combinations.

DupCap_B=*i,j,...*
This parameter indicates the duplex mode to be used for port B of an adapter card. Possible values are: *Half*, *Full*, or *Both*; *Both* is the default. This parameter is relevant only if AutoNeg_B of port B is not set to *Sense*. If AutoNeg_B is set to *On*, all three values of DupCap_B (*Half*, *Full*, or *Both*) might be stated. If AutoNeg_B is set to *Off*, only DupCap_B values *Full* and *Half* are allowed. This DupCap_B parameter is useful if your link partner does not support all possible duplex combinations.

FlowCtrl_A=*i,j,...*
This parameter can be used to set the flow control capabilities the port reports during auto-negotiation. Possible values are: *Sym*, *SymOrRem*, *LocSend*, or *None*; *SymOrRem* is the default. The different modes have the following meaning:

Sym = Symmetric
Both link partners are allowed to send PAUSE frames.

SymOrRem = SymmetricOrRemote
Both or only remote partner are allowed to send PAUSE frames.

LocSend = LocalSend
Only local link partner is allowed to send PAUSE frames.

None = None
No link partner is allowed to send PAUSE frames.

Note that this parameter is ignored if AutoNeg_A is set to *Off*.

FlowCtrl_B=*i,j,...*
This parameter can be used to set the flow control capabilities the port reports during auto-negotiation. Possible values are: *Sym*, *SymOrRem*, *LocSend*, or *None*; *SymOrRem* is the default. The different modes have the following meaning:

Sym = Symmetric
Both link partners are allowed to send PAUSE frames.

SymOrRem = SymmetricOrRemote
Both or only remote partner are allowed to send PAUSE frames.

LocSend = LocalSend
Only local link partner is allowed to send PAUSE frames.

None = None
No link partner is allowed to send PAUSE frames.

Note that this parameter is ignored if AutoNeg_B is set to *Off* .

Role_A=*i,j,...*
This parameter is valid only for 1000Base-T adapter cards. For two 1000Base-T ports to communicate, one must take the role of the master (providing timing information), while the other must be the slave. Possible values are: *Auto*, *Master*, or *Slave*; *Auto* is the default. Usually, the role of a port is negotiated between two ports during link establishment, but if that fails the port A of an adapter card can be forced to a specific setting with this parameter.

Role_B=*i,j,...*
This parameter is valid only for 1000Base-T adapter cards. For two 1000Base-T ports to communicate, one must take the role of the master (providing timing information), while the other must be the slave. Possible values are: *Auto*, *Master*, or *Slave*; *Auto* is the default. Usually, the role of a port is negotiated between two ports during link establishment, but if that fails the port B of an adapter card can be forced to a specific setting with this parameter.

ConType=*i,j,...*
This parameter is a combination of all five per-port parameters within one single parameter. This simplifies the configuration of both ports of an adapter card. The different values of this variable reflect the most meaningful combinations of port parameters. Possible values and their corresponding combination of per-port parameters:

ConType	**DupCap**	**AutoNeg**	**FlowCtrl**	**Role**	**Speed**
Auto	Both	On	SymOrRem	Auto	Auto
100FD	Full	Off	None	Auto	100
100HD	Half	Off	None	Auto	100
10FD	Full	Off	None	Auto	10
10HD	Half	Off	None	Auto	10

Stating any other port parameter together with this *ConType* parameter will result in a merged configuration of those settings. This is due to the fact, that the per-port parameters (e.g., *Speed_A*) have a higher priority than the combined variable *ConType*.

Moderation=*i,j,...*
Interrupt moderation is employed to limit the maximum number of interrupts the driver has to serve. That is, one or more interrupts (which indicate any transmit or receive packet to be processed) are queued until the driver processes them. When queued interrupts are to be served, is determined by the *IntsPerSec* parameter, which is explained later below. Possible moderation modes are: *None*, *Static*, or *Dynamic*; *None* is the default. The different modes have the following meaning:

None No interrupt moderation is applied on the adapter card. Therefore, each transmit or receive interrupt is served immediately as soon as it appears on the interrupt line of the adapter card.

Static Interrupt moderation is applied on the adapter card. All transmit and receive interrupts are queued until a complete moderation interval ends. If such a moderation interval ends, all queued interrupts are processed in one big bunch without any delay. The term *Static* reflects the fact, that interrupt moderation is always enabled, regardless how much network load is currently passing via a particular interface. In addition, the duration of the moderation interval has a fixed length that never changes while the driver is operational.

Dynamic Interrupt moderation might be applied on the adapter card, depending on the load of the system. If the driver detects that the system load is too high, the driver tries to shield the system against too much network load by enabling interrupt moderation. If—at a later time—the CPU utilization decreases again (or if the network load is negligible), the interrupt moderation will automatically be disabled.

Interrupt moderation should be used when the driver has to handle one or more interfaces with a high network load, which—as a consequence—leads also to a high CPU utilization. When moderation is applied in such high network load situations, CPU load might be reduced by 20–30% on slow computers.

Note that the drawback of using interrupt moderation is an increase of the round-trip-time (RTT), due to the queuing and serving of interrupts at dedicated moderation times.

IntsPerSec=*i,j,...*

This parameter determines the length of any interrupt moderation interval. Assuming that static interrupt moderation is to be used, an *IntsPerSec* parameter value of 2000 will lead to an interrupt moderation interval of 500 microseconds. Possible values for this parameter are in the range of 30...40000 (interrupts per second). The default value is 2000.

This parameter is used only if either static or dynamic interrupt moderation is enabled on a network adapter card. This parameter is ignored if no moderation is applied.

Note that the duration of the moderation interval is to be chosen with care. At first glance, selecting a very long duration (e.g., only 100 interrupts per second) seems to be meaningful, but the increase of packet-processing delay is tremendous. On the other hand, selecting a very short moderation time might compensate the use of any moderation being applied.

PrefPort=*i,j,...*

This parameter is used to force the preferred port to A or B (on dual-port network adapters). The preferred port is the one that is used if both ports A and B are detected as fully functional. Possible values are: *A* or *B*; *A* is the default.

RlmtMode=*i,j,...*

RLMT monitors the status of the port. If the link of the active port fails, RLMT switches immediately to the standby link. The virtual link is maintained as long as at least one "physical" link is up. This parameters states how RLMT should monitor both ports. Possible values are: *CheckLinkState*, *CheckLocalPort*, *CheckSeg*, or *DualNet*; *CheckLinkState* is the default. The different modes have the following meaning:

CheckLinkState Check link state only: RLMT uses the link state reported by the adapter hardware for each individual port to determine whether a port can be used for all network traffic or not.

CheckLocalPort In this mode, RLMT monitors the network path between the two ports of an adapter by regularly exchanging packets between them. This mode requires a network configuration in which the two ports are able to "see" each other (i.e., there must not be any router between the ports).

CheckSeg Check local port and segmentation: This mode supports the same functions as the CheckLocalPort mode and additionally checks network segmentation between the ports. Therefore, this mode is to be used only if Gigabit Ethernet switches are installed on the network that have been configured to use the Spanning Tree protocol.

DualNet In this mode, ports A and B are used as separate devices. If you have a dual port adapter, port A will be configured as *eth[x]* and port B as *eth[x+1]*. Both ports can be used independently with distinct IP addresses. The preferred port setting is not used. RLMT is turned off.

Note that RLMT modes *CheckLocalPort* and *CheckLinkState* are designed to operate in configurations where a network path between the ports on one adapter exists. Moreover, they are not designed to work where adapters are connected back-to-back.

FILES

/proc/net/sk98lin/eth[x]

The statistics file of a particular interface of an adapter card. It contains generic information about the adapter card plus a detailed summary of all transmit and receive counters.

/usr/src/linux/Documentation/networking/sk98lin.txt

This is the *README* file of the *sk98lin* driver. It contains a detailed installation HOWTO and describes all parameters of the driver. It denotes also common problems and provides the solution to them.

BUGS

Report any bugs to linux@syskonnect.de

SEE ALSO

ifconfig(8), *insmod*(8), *modprobe*(8)

NAME

smartpqi − Microchip Smart Storage SCSI driver

SYNOPSIS

modprobe smartpqi [**disable_device_id_wildcards=**{**0**|**1**}] [**disable_heartbeat=**{**0**|**1**}]
[**disable_ctrl_shutdown=**{**0**|**1**}] [**lockup_action=**{**none**|**reboot**|**panic**}]
[**expose_ld_first=**{**0**|**1**}] [**hide_vsep=**{**0**|**1**}]
[**disable_managed_interrupts=**{**0**|**1**}] [**ctrl_ready_timeout=**{**0**|[**30**,**1800**]}]

DESCRIPTION

smartpqi is a SCSI driver for Microchip Smart Storage controllers.

Supported *ioctl*() operations

For compatibility with applications written for the *cciss*(4) and *hpsa*(4) drivers, many, but not all of the *ioctl*(2) operations supported by the **hpsa** driver are also supported by the **smartpqi** driver. The data structures used by these operations are described in the Linux kernel source file *include/linux/cciss_ioctl.h*.

CCISS_DEREGDISK
CCISS_REGNEWDISK
CCISS_REGNEWD
: These operations all do exactly the same thing, which is to cause the driver to re-scan for new devices. This does exactly the same thing as writing to the **smartpqi**-specific host *rescan* attribute.

CCISS_GETPCIINFO
: This operation returns the PCI domain, bus, device, and function and "board ID" (PCI subsystem ID).

CCISS_GETDRIVVER
: This operation returns the driver version in four bytes, encoded as:

```
(major_version << 28) | (minor_version << 24) |
        (release << 16) | revision
```

CCISS_PASSTHRU
: Allows BMIC and CISS commands to be passed through to the controller.

Boot options

disable_device_id_wildcards={**0**|**1**}
: Disables support for device ID wildcards. The default value is 0 (wildcards are enabled).

disable_heartbeat={**0**|**1**}
: Disables support for the controller's heartbeat check. This parameter is used for debugging purposes. The default value is 0 (the controller's heartbeat check is enabled).

disable_ctrl_shutdown={**0**|**1**}
: Disables support for shutting down the controller in the event of a controller lockup. The default value is 0 (controller will be shut down).

lockup_action={**none**|**reboot**|**panic**}
: Specifies the action the driver takes when a controller lockup is detected. The default action is **none**.

parameter	action
none	take controller offline only
reboot	reboot the system
panic	panic the system

expose_ld_first={**0**|**1**}
: This option exposes logical devices to the OS before physical devices. The default value is 0 (physical devices exposed first).

hide_vsep={**0**|**1**}
: This option disables exposure of the virtual SEP to the OS. The default value is 0 (virtual SEP is exposed).

disable_managed_interrupts={**0**|**1**}
Disables driver utilization of Linux kernel managed interrupts for controllers. The managed interrupts feature automatically distributes interrupts to all available CPUs and assigns SMP affinity. The default value is 0 (managed interrupts enabled).

ctrl_ready_timeout={**0**|[**30**,**1800**]}
This option specifies the timeout in seconds for the driver to wait for the controller to be ready. The valid range is 0 or [**30**, **1800**]. The default value is 0, which causes the driver to use a timeout of 180 seconds.

FILES

Device nodes

Disk drives are accessed via the SCSI disk driver (*sd*), tape drives via the SCSI tape driver (*st*), and the RAID controller via the SCSI generic driver (*sg*), with device nodes named */dev/sd**, */dev/st**, and */dev/sg**, respectively.

SmartPQI-specific host attribute files in */sys*

/sys/class/scsi_host/host/rescan*
The host *rescan* attribute is a write-only attribute. Writing to this attribute will cause the driver to scan for new, changed, or removed devices (e.g., hot-plugged tape drives, or newly configured or deleted logical volumes) and notify the SCSI mid-layer of any changes detected. Usually this action is triggered automatically by configuration changes, so the user should not normally have to write to this file. Doing so may be useful when hot-plugging devices such as tape drives or entire storage boxes containing pre-configured logical volumes.

/sys/class/scsi_host/host/lockup_action*
The host *lockup_action* attribute is a read/write attribute. This attribute will cause the driver to perform a specific action in the unlikely event that a controller lockup has been detected. See **OPTIONS** above for an explanation of the *lockup_action* values.

/sys/class/scsi_host/host/driver_version*
The *driver_version* attribute is read-only. This attribute contains the smartpqi driver version.

For example:

```
$ cat /sys/class/scsi_host/host1/driver_version
1.1.2-126
```

/sys/class/scsi_host/host/firmware_version*
The *firmware_version* attribute is read-only. This attribute contains the controller firmware version.

For example:

```
$ cat /sys/class/scsi_host/host1/firmware_version
1.29-112
```

/sys/class/scsi_host/host/model*
The *model* attribute is read-only. This attribute contains the product identification string of the controller.

For example:

```
$ cat /sys/class/scsi_host/host1/model
1100-16i
```

/sys/class/scsi_host/host/serial_number*
The *serial_number* attribute is read-only. This attribute contains the unique identification number of the controller.

For example:

```
$ cat /sys/class/scsi_host/host1/serial_number
6A316373777
```

/sys/class/scsi_host/host/vendor*
The *vendor* attribute is read-only. This attribute contains the vendor identification string of the controller.

For example:

```
$ cat /sys/class/scsi_host/host1/vendor
Adaptec
```

/sys/class/scsi_host/host/enable_stream_detection*
The *enable_stream_detection* attribute is read-write. This attribute enables/disables stream detection in the driver. Enabling stream detection can improve sequential write performance for ioaccel-enabled volumes. See the **ssd_smart_path_enabled** disk attribute section for details on ioaccel-enabled volumes. The default value is 1 (stream detection enabled).

Enable example:

```
$ echo 1 > /sys/class/scsi_host/host1/enable_stream_detection
```

/sys/class/scsi_host/host/enable_r5_writes*
The *enable_r5_writes* attribute is read-write. This attribute enables/disables RAID 5 write operations for ioaccel-enabled volumes. Enabling can improve sequential write performance. See the **ssd_smart_path_enabled** disk attribute section for details on ioaccel-enabled volumes. The default value is 1 (RAID 5 writes enabled).

Enable example:

```
$ echo 1 > /sys/class/scsi_host/host1/enable_r5_writes
```

/sys/class/scsi_host/host/enable_r6_writes*
The *enable_r6_writes* attribute is read-write. This attribute enables/disables RAID 6 write operations for ioaccel-enabled volumes. Enabling can improve sequential write performance. See the **ssd_smart_path_enabled** disk attribute section for details on ioaccel-enabled volumes. The default value is 1 (RAID 6 writes enabled).

Enable example:

```
$ echo 1 > /sys/class/scsi_host/host1/enable_r6_writes
```

SmartPQI-specific disk attribute files in */sys*
In the file specifications below, *c* stands for the number of the appropriate SCSI controller, *b* is the bus number, *t* the target number, and *l* is the logical unit number (LUN).

*/sys/class/scsi_disk/*c:b:t:l*/device/raid_level*
The *raid_level* attribute is read-only. This attribute contains the RAID level of the logical volume.

For example:

```
$ cat /sys/class/scsi_disk/4:0:0:0/device/raid_level
RAID 0
```

*/sys/class/scsi_disk/*c:b:t:l*/device/sas_address*
The *sas_address* attribute is read-only. This attribute contains the SAS address of the device.

For example:

```
$ cat /sys/class/scsi_disk/1:0:3:0/device/sas_address
0x5001173d028543a2
```

*/sys/class/scsi_disk/*c:b:t:l*/device/ssd_smart_path_enabled*
The *ssd_smart_path_enabled* attribute is read-only. This attribute is for ioaccel-enabled volumes. (Ioaccel is an alternative driver submission path that allows the driver to send I/O requests directly to backend SCSI devices, bypassing the controller firmware. This results in an increase in performance. This method is used for HBA disks and for logical volumes comprised of SSDs.) Contains 1 if ioaccel is enabled for the volume and 0 otherwise.

For example:

```
$ cat /sys/class/scsi_disk/1:0:3:0/device/ssd_smart_path_enabled
```

*/sys/class/scsi_disk/*c:b:t:l*/device/lunid*
The *lunid* attribute is read-only. This attribute contains the SCSI LUN ID for the device.

For example:

```
$ cat /sys/class/scsi_disk/13:1:0:3/device/lunid
0x0300004000000000
```

/sys/class/scsi_disk/ c:b:t:l*/device/unique_id*
: The *unique_id* attribute is read-only. This attribute contains a 16-byte ID that uniquely identifies the device within the controller.

  For example:

```
$ cat /sys/class/scsi_disk/13:1:0:3/device/unique_id
600508B1001C6D4723A8E98D704FDB94
```

/sys/class/scsi_disk/ c:b:t:l*/device/path_info*
: The *path_info* attribute is read-only. This attribute contains the *c*:*b*:*t*:*l* of the device along with the device type and whether the device is Active or Inactive. If the device is an HBA device, *path_info* will also display the PORT, BOX, and BAY the device is plugged into.

  For example:

```
$ cat /sys/class/scsi_disk/13:1:0:3/device/path_info
[13:1:0:3]    Direct-Access     Active

$ cat /sys/class/scsi_disk/12:0:9:0/device/path_info
[12:0:9:0]  Direct-Access   PORT: C1 BOX: 1 BAY: 14 Inactive
[12:0:9:0]  Direct-Access   PORT: C0 BOX: 1 BAY: 14 Active
```

/sys/class/scsi_disk/ c:b:t:l*/device/raid_bypass_cnt*
: The *raid_bypass_cnt* attribute is read-only. This attribute contains the number of I/O requests that have gone through the ioaccel path for ioaccel-enabled volumes. See the **ssd_smart_path_enabled** disk attribute section for details on ioaccel-enabled volumes.

  For example:

```
$ cat /sys/class/scsi_disk/13:1:0:3/device/raid_bypass_cnt
0x300
```

/sys/class/scsi_disk/ c:b:t:l*/device/sas_ncq_prio_enable*
: The *sas_ncq_prio_enable* attribute is read/write. This attribute enables SATA NCQ priority support. This attribute works only when device has NCQ support and controller firmware can handle IO with NCQ priority attribute.

  For example:

```
$ echo 1 > /sys/class/scsi_disk/13:1:0:3/device/sas_ncq_prio_enable
```

VERSIONS

The **smartpqi** driver was added in Linux 4.9.

NOTES

Configuration

To configure a Microchip Smart Storage controller, refer to the User Guide for the controller, which can be found by searching for the specific controller at .

HISTORY

/sys/class/scsi_host/host/version* was replaced by two sysfs entries:

/sys/class/scsi_host/host/driver_version*

/sys/class/scsi_host/host/firmware_version*

SEE ALSO

cciss(4), *hpsa*(4), *sd*(4), *st*(4), *sg*(4)

Documentation/ABI/testing/sysfs-bus-pci-devices-cciss in the Linux kernel source tree.

NAME

st − SCSI tape device

SYNOPSIS

#include <sys/mtio.h>

int ioctl(int *fd***, int** *request* **[, (void *)***arg3***]);**
int ioctl(int *fd***, MTIOCTOP, (struct mtop *)***mt_cmd***);**
int ioctl(int *fd***, MTIOCGET, (struct mtget *)***mt_status***);**
int ioctl(int *fd***, MTIOCPOS, (struct mtpos *)***mt_pos***);**

DESCRIPTION

The **st** driver provides the interface to a variety of SCSI tape devices. Currently, the driver takes control of all detected devices of type "sequential-access". The **st** driver uses major device number 9.

Each device uses eight minor device numbers. The lowermost five bits in the minor numbers are assigned sequentially in the order of detection. In the 2.6 kernel, the bits above the eight lowermost bits are concatenated to the five lowermost bits to form the tape number. The minor numbers can be grouped into two sets of four numbers: the principal (auto-rewind) minor device numbers, *n*, and the "no-rewind" device numbers, (*n* + 128). Devices opened using the principal device number will be sent a **REWIND** command when they are closed. Devices opened using the "no-rewind" device number will not. (Note that using an auto-rewind device for positioning the tape with, for instance, mt does not lead to the desired result: the tape is rewound after the mt command and the next command starts from the beginning of the tape).

Within each group, four minor numbers are available to define devices with different characteristics (block size, compression, density, etc.) When the system starts up, only the first device is available. The other three are activated when the default characteristics are defined (see below). (By changing compile-time constants, it is possible to change the balance between the maximum number of tape drives and the number of minor numbers for each drive. The default allocation allows control of 32 tape drives. For instance, it is possible to control up to 64 tape drives with two minor numbers for different options.)

Devices are typically created by:

```
mknod −m 666 /dev/st0 c 9 0
mknod −m 666 /dev/st0l c 9 32
mknod −m 666 /dev/st0m c 9 64
mknod −m 666 /dev/st0a c 9 96
mknod −m 666 /dev/nst0 c 9 128
mknod −m 666 /dev/nst0l c 9 160
mknod −m 666 /dev/nst0m c 9 192
mknod −m 666 /dev/nst0a c 9 224
```

There is no corresponding block device.

The driver uses an internal buffer that has to be large enough to hold at least one tape block. Before Linux 2.1.121, the buffer is allocated as one contiguous block. This limits the block size to the largest contiguous block of memory the kernel allocator can provide. The limit is currently 128 kB for 32-bit architectures and 256 kB for 64-bit architectures. In newer kernels the driver allocates the buffer in several parts if necessary. By default, the maximum number of parts is 16. This means that the maximum block size is very large (2 MB if allocation of 16 blocks of 128 kB succeeds).

The driver's internal buffer size is determined by a compile-time constant which can be overridden with a kernel startup option. In addition to this, the driver tries to allocate a larger temporary buffer at run time if necessary. However, run-time allocation of large contiguous blocks of memory may fail and it is advisable not to rely too much on dynamic buffer allocation before Linux 2.1.121 (this applies also to demand-loading the driver with kerneld or kmod).

The driver does not specifically support any tape drive brand or model. After system start-up the tape device options are defined by the drive firmware. For example, if the drive firmware selects fixed-block mode, the tape device uses fixed-block mode. The options can be changed with explicit *ioctl*(2) calls and remain in effect when the device is closed and reopened. Setting the options affects both the auto-rewind and the nonrewind device.

Different options can be specified for the different devices within the subgroup of four. The options

take effect when the device is opened. For example, the system administrator can define one device that writes in fixed-block mode with a certain block size, and one which writes in variable-block mode (if the drive supports both modes).

The driver supports **tape partitions** if they are supported by the drive. (Note that the tape partitions have nothing to do with disk partitions. A partitioned tape can be seen as several logical tapes within one medium.) Partition support has to be enabled with an *ioctl*(2). The tape location is preserved within each partition across partition changes. The partition used for subsequent tape operations is selected with an *ioctl*(2). The partition switch is executed together with the next tape operation in order to avoid unnecessary tape movement. The maximum number of partitions on a tape is defined by a compile-time constant (originally four). The driver contains an *ioctl*(2) that can format a tape with either one or two partitions.

Device */dev/tape* is usually created as a hard or soft link to the default tape device on the system.

Starting from Linux 2.6.2, the driver exports in the sysfs directory */sys/class/scsi_tape* the attached devices and some parameters assigned to the devices.

Data transfer

The driver supports operation in both fixed-block mode and variable-block mode (if supported by the drive). In fixed-block mode the drive writes blocks of the specified size and the block size is not dependent on the byte counts of the write system calls. In variable-block mode one tape block is written for each write call and the byte count determines the size of the corresponding tape block. Note that the blocks on the tape don't contain any information about the writing mode: when reading, the only important thing is to use commands that accept the block sizes on the tape.

In variable-block mode the read byte count does not have to match the tape block size exactly. If the byte count is larger than the next block on tape, the driver returns the data and the function returns the actual block size. If the block size is larger than the byte count, an error is returned.

In fixed-block mode the read byte counts can be arbitrary if buffering is enabled, or a multiple of the tape block size if buffering is disabled. Before Linux 2.1.121 allow writes with arbitrary byte count if buffering is enabled. In all other cases (before Linux 2.1.121 with buffering disabled or newer kernel) the write byte count must be a multiple of the tape block size.

In Linux 2.6, the driver tries to use direct transfers between the user buffer and the device. If this is not possible, the driver's internal buffer is used. The reasons for not using direct transfers include improper alignment of the user buffer (default is 512 bytes but this can be changed by the HBA driver), one or more pages of the user buffer not reachable by the SCSI adapter, and so on.

A filemark is automatically written to tape if the last tape operation before close was a write.

When a filemark is encountered while reading, the following happens. If there are data remaining in the buffer when the filemark is found, the buffered data is returned. The next read returns zero bytes. The following read returns data from the next file. The end of recorded data is signaled by returning zero bytes for two consecutive read calls. The third read returns an error.

Ioctls

The driver supports three *ioctl*(2) requests. Requests not recognized by the **st** driver are passed to the **SCSI** driver. The definitions below are from */usr/include/linux/mtio.h*:

MTIOCTOP — perform a tape operation

This request takes an argument of type *(struct mtop *)*. Not all drives support all operations. The driver returns an **EIO** error if the drive rejects an operation.

```
/* Structure for MTIOCTOP - mag tape op command: */
struct mtop {
    short   mt_op;       /* operations defined below */
    int     mt_count;    /* how many of them */
};
```

Magnetic tape operations for normal tape use:

MTBSF

Backward space over *mt_count* filemarks.

MTBSFM
Backward space over *mt_count* filemarks. Reposition the tape to the EOT side of the last filemark.

MTBSR
Backward space over *mt_count* records (tape blocks).

MTBSS
Backward space over *mt_count* setmarks.

MTCOMPRESSION
Enable compression of tape data within the drive if *mt_count* is nonzero and disable compression if *mt_count* is zero. This command uses the MODE page 15 supported by most DATs.

MTEOM
Go to the end of the recorded media (for appending files).

MTERASE
Erase tape. With Linux 2.6, short erase (mark tape empty) is performed if the argument is zero. Otherwise, long erase (erase all) is done.

MTFSF
Forward space over *mt_count* filemarks.

MTFSFM
Forward space over *mt_count* filemarks. Reposition the tape to the BOT side of the last filemark.

MTFSR
Forward space over *mt_count* records (tape blocks).

MTFSS
Forward space over *mt_count* setmarks.

MTLOAD
Execute the SCSI load command. A special case is available for some HP autoloaders. If *mt_count* is the constant **MT_ST_HPLOADER_OFFSET** plus a number, the number is sent to the drive to control the autoloader.

MTLOCK
Lock the tape drive door.

MTMKPART
Format the tape into one or two partitions. If *mt_count* is positive, it gives the size of partition 1 and partition 0 contains the rest of the tape. If *mt_count* is zero, the tape is formatted into one partition. From Linux 4.6, a negative *mt_count* specifies the size of partition 0 and the rest of the tape contains partition 1. The physical ordering of partitions depends on the drive. This command is not allowed for a drive unless the partition support is enabled for the drive (see **MT_ST_CAN_PARTITIONS** below).

MTNOP
No op—flushes the driver's buffer as a side effect. Should be used before reading status with **MTIOCGET**.

MTOFFL
Rewind and put the drive off line.

MTRESET
Reset drive.

MTRETEN
Re-tension tape.

MTREW
Rewind.

MTSEEK
Seek to the tape block number specified in *mt_count*. This operation requires either a SCSI-2 drive that supports the **LOCATE** command (device-specific address) or a Tandberg-

compatible SCSI-1 drive (Tandberg, Archive Viper, Wangtek, ...). The block number should be one that was previously returned by **MTIOCPOS** if device-specific addresses are used.

MTSETBLK
Set the drive's block length to the value specified in *mt_count*. A block length of zero sets the drive to variable block size mode.

MTSETDENSITY
Set the tape density to the code in *mt_count*. The density codes supported by a drive can be found from the drive documentation.

MTSETPART
The active partition is switched to *mt_count*. The partitions are numbered from zero. This command is not allowed for a drive unless the partition support is enabled for the drive (see **MT_ST_CAN_PARTITIONS** below).

MTUNLOAD
Execute the SCSI unload command (does not eject the tape).

MTUNLOCK
Unlock the tape drive door.

MTWEOF
Write *mt_count* filemarks.

MTWSM
Write *mt_count* setmarks.

Magnetic tape operations for setting of device options (by the superuser):

MTSETDRVBUFFER
Set various drive and driver options according to bits encoded in *mt_count*. These consist of the drive's buffering mode, a set of Boolean driver options, the buffer write threshold, defaults for the block size and density, and timeouts (only since Linux 2.1). A single operation can affect only one item in the list below (the Booleans counted as one item.)

A value having zeros in the high-order 4 bits will be used to set the drive's buffering mode. The buffering modes are:

0 The drive will not report **GOOD** status on write commands until the data blocks are actually written to the medium.

1 The drive may report **GOOD** status on write commands as soon as all the data has been transferred to the drive's internal buffer.

2 The drive may report **GOOD** status on write commands as soon as (a) all the data has been transferred to the drive's internal buffer, and (b) all buffered data from different initiators has been successfully written to the medium.

To control the write threshold the value in *mt_count* must include the constant **MT_ST_WRITE_THRESHOLD** bitwise ORed with a block count in the low 28 bits. The block count refers to 1024-byte blocks, not the physical block size on the tape. The threshold cannot exceed the driver's internal buffer size (see DESCRIPTION, above).

To set and clear the Boolean options the value in *mt_count* must include one of the constants **MT_ST_BOOLEANS**, **MT_ST_SETBOOLEANS**, **MT_ST_CLEARBOOLEANS**, or **MT_ST_DEFBOOLEANS** bitwise ORed with whatever combination of the following options is desired. Using **MT_ST_BOOLEANS** the options can be set to the values defined in the corresponding bits. With **MT_ST_SETBOOLEANS** the options can be selectively set and with **MT_ST_DEFBOOLEANS** selectively cleared.

The default options for a tape device are set with **MT_ST_DEFBOOLEANS**. A nonactive tape device (e.g., device with minor 32 or 160) is activated when the default options for it are defined the first time. An activated device inherits from the device activated at start-up the options not set explicitly.

The Boolean options are:

MT_ST_BUFFER_WRITES (Default: true)
Buffer all write operations in fixed-block mode. If this option is false and the drive uses a fixed block size, then all write operations must be for a multiple of the block size. This option must be set false to write reliable multivolume archives.

MT_ST_ASYNC_WRITES (Default: true)
When this option is true, write operations return immediately without waiting for the data to be transferred to the drive if the data fits into the driver's buffer. The write threshold determines how full the buffer must be before a new SCSI write command is issued. Any errors reported by the drive will be held until the next operation. This option must be set false to write reliable multivolume archives.

MT_ST_READ_AHEAD (Default: true)
This option causes the driver to provide read buffering and read-ahead in fixed-block mode. If this option is false and the drive uses a fixed block size, then all read operations must be for a multiple of the block size.

MT_ST_TWO_FM (Default: false)
This option modifies the driver behavior when a file is closed. The normal action is to write a single filemark. If the option is true, the driver will write two filemarks and backspace over the second one.

Note: This option should not be set true for QIC tape drives since they are unable to overwrite a filemark. These drives detect the end of recorded data by testing for blank tape rather than two consecutive filemarks. Most other current drives also detect the end of recorded data and using two filemarks is usually necessary only when interchanging tapes with some other systems.

MT_ST_DEBUGGING (Default: false)
This option turns on various debugging messages from the driver (effective only if the driver was compiled with **DEBUG** defined nonzero).

MT_ST_FAST_EOM (Default: false)
This option causes the **MTEOM** operation to be sent directly to the drive, potentially speeding up the operation but causing the driver to lose track of the current file number normally returned by the **MTIOCGET** request. If **MT_ST_FAST_EOM** is false, the driver will respond to an **MTEOM** request by forward spacing over files.

MT_ST_AUTO_LOCK (Default: false)
When this option is true, the drive door is locked when the device file is opened and unlocked when it is closed.

MT_ST_DEF_WRITES (Default: false)
The tape options (block size, mode, compression, etc.) may change when changing from one device linked to a drive to another device linked to the same drive depending on how the devices are defined. This option defines when the changes are enforced by the driver using SCSI-commands and when the drives auto-detection capabilities are relied upon. If this option is false, the driver sends the SCSI-commands immediately when the device is changed. If the option is true, the SCSI-commands are not sent until a write is requested. In this case, the drive firmware is allowed to detect the tape structure when reading and the SCSI-commands are used only to make sure that a tape is written according to the correct specification.

MT_ST_CAN_BSR (Default: false)
When read-ahead is used, the tape must sometimes be spaced backward to the correct position when the device is closed and the SCSI command to space backward over records is used for this purpose. Some older drives can't process this command reliably and this option can be used to instruct the driver not to use the command. The end result is that, with read-ahead and fixed-block mode, the tape may not be correctly positioned within a file when the device is closed. With Linux 2.6, the default is true for drives supporting SCSI-3.

MT_ST_NO_BLKLIMS (Default: false)
Some drives don't accept the **READ BLOCK LIMITS** SCSI command. If this is used, the driver does not use the command. The drawback is that the driver can't

check before sending commands if the selected block size is acceptable to the drive.

MT_ST_CAN_PARTITIONS (Default: false)
This option enables support for several partitions within a tape. The option applies to all devices linked to a drive.

MT_ST_SCSI2LOGICAL (Default: false)
This option instructs the driver to use the logical block addresses defined in the SCSI-2 standard when performing the seek and tell operations (both with **MTSEEK** and **MTIOCPOS** commands and when changing tape partition). Otherwise, the device-specific addresses are used. It is highly advisable to set this option if the drive supports the logical addresses because they count also filemarks. There are some drives that support only the logical block addresses.

MT_ST_SYSV (Default: false)
When this option is enabled, the tape devices use the System V semantics. Otherwise, the BSD semantics are used. The most important difference between the semantics is what happens when a device used for reading is closed: in System V semantics the tape is spaced forward past the next filemark if this has not happened while using the device. In BSD semantics the tape position is not changed.

MT_NO_WAIT (Default: false)
Enables immediate mode (i.e., don't wait for the command to finish) for some commands (e.g., rewind).

An example:

```
struct mtop mt_cmd;
mt_cmd.mt_op = MTSETDRVBUFFER;
mt_cmd.mt_count = MT_ST_BOOLEANS |
        MT_ST_BUFFER_WRITES | MT_ST_ASYNC_WRITES;
ioctl(fd, MTIOCTOP, mt_cmd);
```

The default block size for a device can be set with **MT_ST_DEF_BLKSIZE** and the default density code can be set with **MT_ST_DEFDENSITY**. The values for the parameters are or'ed with the operation code.

With Linux 2.1.x and later, the timeout values can be set with the subcommand **MT_ST_SET_TIMEOUT** ORed with the timeout in seconds. The long timeout (used for rewinds and other commands that may take a long time) can be set with **MT_ST_SET_LONG_TIMEOUT**. The kernel defaults are very long to make sure that a successful command is not timed out with any drive. Because of this, the driver may seem stuck even if it is only waiting for the timeout. These commands can be used to set more practical values for a specific drive. The timeouts set for one device apply for all devices linked to the same drive.

Starting from Linux 2.4.19 and Linux 2.5.43, the driver supports a status bit which indicates whether the drive requests cleaning. The method used by the drive to return cleaning information is set using the **MT_ST_SEL_CLN** subcommand. If the value is zero, the cleaning bit is always zero. If the value is one, the TapeAlert data defined in the SCSI-3 standard is used (not yet implemented). Values 2–17 are reserved. If the lowest eight bits are >= 18, bits from the extended sense data are used. The bits 9–16 specify a mask to select the bits to look at and the bits 17–23 specify the bit pattern to look for. If the bit pattern is zero, one or more bits under the mask indicate the cleaning request. If the pattern is nonzero, the pattern must match the masked sense data byte.

MTIOCGET — get status
This request takes an argument of type *(struct mtget *)*.

```
/* structure for MTIOCGET - mag tape get status command */
struct mtget {
    long     mt_type;
    long     mt_resid;
    /* the following registers are device dependent */
    long     mt_dsreg;
```

```
        long     mt_gstat;
        long     mt_erreg;
        /* The next two fields are not always used */
        daddr_t  mt_fileno;
        daddr_t  mt_blkno;
    };
```

mt_type
: The header file defines many values for *mt_type*, but the current driver reports only the generic types **MT_ISSCSI1** (Generic SCSI-1 tape) and **MT_ISSCSI2** (Generic SCSI-2 tape).

mt_resid
: contains the current tape partition number.

mt_dsreg
: reports the drive's current settings for block size (in the low 24 bits) and density (in the high 8 bits). These fields are defined by **MT_ST_BLKSIZE_SHIFT**, **MT_ST_BLKSIZE_MASK**, **MT_ST_DENSITY_SHIFT**, and **MT_ST_DENSITY_MASK**.

mt_gstat
: reports generic (device independent) status information. The header file defines macros for testing these status bits:

  GMT_EOF(*x*)
  : The tape is positioned just after a filemark (always false after an **MTSEEK** operation).

  GMT_BOT(*x*)
  : The tape is positioned at the beginning of the first file (always false after an **MTSEEK** operation).

  GMT_EOT(*x*)
  : A tape operation has reached the physical End Of Tape.

  GMT_SM(*x*)
  : The tape is currently positioned at a setmark (always false after an **MTSEEK** operation).

  GMT_EOD(*x*)
  : The tape is positioned at the end of recorded data.

  GMT_WR_PROT(*x*)
  : The drive is write-protected. For some drives this can also mean that the drive does not support writing on the current medium type.

  GMT_ONLINE(*x*)
  : The last *open*(2) found the drive with a tape in place and ready for operation.

  GMT_D_6250(*x*)
  GMT_D_1600(*x*)
  GMT_D_800(*x*)
  : This "generic" status information reports the current density setting for 9-track ½" tape drives only.

  GMT_DR_OPEN(*x*)
  : The drive does not have a tape in place.

  GMT_IM_REP_EN(*x*)
  : Immediate report mode. This bit is set if there are no guarantees that the data has been physically written to the tape when the write call returns. It is set zero only when the driver does not buffer data and the drive is set not to buffer data.

  GMT_CLN(*x*)
  : The drive has requested cleaning. Implemented since Linux 2.4.19 and Linux 2.5.43.

mt_erreg
: The only field defined in *mt_erreg* is the recovered error count in the low 16 bits (as defined by **MT_ST_SOFTERR_SHIFT** and **MT_ST_SOFTERR_MASK**). Due to inconsistencies

in the way drives report recovered errors, this count is often not maintained (most drives do not by default report soft errors but this can be changed with a SCSI MODE SELECT command).

mt_fileno
reports the current file number (zero-based). This value is set to −1 when the file number is unknown (e.g., after **MTBSS** or **MTSEEK**).

mt_blkno
reports the block number (zero-based) within the current file. This value is set to −1 when the block number is unknown (e.g., after **MTBSF**, **MTBSS**, or **MTSEEK**).

MTIOCPOS — get tape position
This request takes an argument of type *(struct mtpos *)* and reports the drive's notion of the current tape block number, which is not the same as *mt_blkno* returned by **MTIOCGET**. This drive must be a SCSI-2 drive that supports the **READ POSITION** command (device-specific address) or a Tandberg-compatible SCSI-1 drive (Tandberg, Archive Viper, Wangtek, ...).

```
/* structure for MTIOCPOS - mag tape get position command */
struct mtpos {
    long mt_blkno;    /* current block number */
};
```

RETURN VALUE

EACCES
An attempt was made to write or erase a write-protected tape. (This error is not detected during *open*(2).)

EBUSY
The device is already in use or the driver was unable to allocate a buffer.

EFAULT
The command parameters point to memory not belonging to the calling process.

EINVAL
An *ioctl*(2) had an invalid argument, or a requested block size was invalid.

EIO The requested operation could not be completed.

ENOMEM
The byte count in *read*(2) is smaller than the next physical block on the tape. (Before Linux 2.2.18 and Linux 2.4.0 the extra bytes have been silently ignored.)

ENOSPC
A write operation could not be completed because the tape reached end-of-medium.

ENOSYS
Unknown *ioctl*(2).

ENXIO
During opening, the tape device does not exist.

EOVERFLOW
An attempt was made to read or write a variable-length block that is larger than the driver's internal buffer.

EROFS
Open is attempted with **O_WRONLY** or **O_RDWR** when the tape in the drive is write-protected.

FILES

*/dev/st**
the auto-rewind SCSI tape devices

*/dev/nst**
the nonrewind SCSI tape devices

NOTES

- When exchanging data between systems, both systems have to agree on the physical tape block size. The parameters of a drive after startup are often not the ones most operating systems use with these devices. Most systems use drives in variable-block mode if the drive supports that mode. This applies to most modern drives, including DATs, 8mm helical scan drives, DLTs, etc. It may be advisable to use these drives in variable-block mode also in Linux (i.e., use **MTSETBLK** or **MTSETDEFBLK** at system startup to set the mode), at least when exchanging data with a foreign system. The drawback of this is that a fairly large tape block size has to be used to get acceptable data transfer rates on the SCSI bus.
- Many programs (e.g., *tar*(1)) allow the user to specify the blocking factor on the command line. Note that this determines the physical block size on tape only in variable-block mode.
- In order to use SCSI tape drives, the basic SCSI driver, a SCSI-adapter driver and the SCSI tape driver must be either configured into the kernel or loaded as modules. If the SCSI-tape driver is not present, the drive is recognized but the tape support described in this page is not available.
- The driver writes error messages to the console/log. The SENSE codes written into some messages are automatically translated to text if verbose SCSI messages are enabled in kernel configuration.
- The driver's internal buffering allows good throughput in fixed-block mode also with small *read*(2) and *write*(2) byte counts. With direct transfers this is not possible and may cause a surprise when moving to the 2.6 kernel. The solution is to tell the software to use larger transfers (often telling it to use larger blocks). If this is not possible, direct transfers can be disabled.

SEE ALSO

mt(1)

The file *drivers/scsi/README.st* or *Documentation/scsi/st.txt* (kernel >= 2.6) in the Linux kernel source tree contains the most recent information about the driver and its configuration possibilities

NAME

tty − controlling terminal

DESCRIPTION

The file */dev/tty* is a character file with major number 5 and minor number 0, usually with mode 0666 and ownership root:tty. It is a synonym for the controlling terminal of a process, if any.

In addition to the *ioctl*(2) requests supported by the device that **tty** refers to, the *ioctl*(2) request **TIOCNOTTY** is supported.

TIOCNOTTY

Detach the calling process from its controlling terminal.

If the process is the session leader, then **SIGHUP** and **SIGCONT** signals are sent to the foreground process group and all processes in the current session lose their controlling tty.

This *ioctl*(2) call works only on file descriptors connected to */dev/tty*. It is used by daemon processes when they are invoked by a user at a terminal. The process attempts to open */dev/tty*. If the open succeeds, it detaches itself from the terminal by using **TIOCNOTTY**, while if the open fails, it is obviously not attached to a terminal and does not need to detach itself.

FILES

/dev/tty

SEE ALSO

chown(1), *mknod*(1), *ioctl*(2), *ioctl_console*(2), *ioctl_tty*(2), *termios*(3), *ttyS*(4), *vcs*(4), *pty*(7), *agetty*(8), *mingetty*(8)

NAME

ttyS − serial terminal lines

DESCRIPTION

ttyS[0−3] are character devices for the serial terminal lines.

They are typically created by:

```
mknod −m 660 /dev/ttyS0 c 4 64 # base address 0x3f8
mknod −m 660 /dev/ttyS1 c 4 65 # base address 0x2f8
mknod −m 660 /dev/ttyS2 c 4 66 # base address 0x3e8
mknod −m 660 /dev/ttyS3 c 4 67 # base address 0x2e8
chown root:tty /dev/ttyS[0−3]
```

FILES

/dev/ttyS[0−3]

SEE ALSO

chown(1), *mknod*(1), *tty*(4), *agetty*(8), *mingetty*(8), *setserial*(8)

NAME

vcs, vcsa − virtual console memory

DESCRIPTION

/dev/vcs0 is a character device with major number 7 and minor number 0, usually with mode 0644 and ownership root:tty. It refers to the memory of the currently displayed virtual console terminal.

/dev/vcs[1−63] are character devices for virtual console terminals, they have major number 7 and minor number 1 to 63, usually mode 0644 and ownership root:tty. */dev/vcsa[0−63]* are the same, but using *unsigned short*s (in host byte order) that include attributes, and prefixed with four bytes giving the screen dimensions and cursor position: *lines*, *columns*, *x*, *y*. (*x* = *y* = 0 at the top left corner of the screen.)

When a 512-character font is loaded, the 9th bit position can be fetched by applying the *ioctl*(2) **VT_GETHIFONTMASK** operation (available since Linux 2.6.18) on */dev/tty[1−63]*; the value is returned in the *unsigned short* pointed to by the third *ioctl*(2) argument.

These devices replace the screendump *ioctl*(2) operations of *ioctl_console*(2), so the system administrator can control access using filesystem permissions.

The devices for the first eight virtual consoles may be created by:

```
for x in 0 1 2 3 4 5 6 7 8; do
    mknod −m 644 /dev/vcs$x c 7 $x;
    mknod −m 644 /dev/vcsa$x c 7 $[$x+128];
done
chown root:tty /dev/vcs*
```

No *ioctl*(2) requests are supported.

FILES

/dev/vcs[0−63]
/dev/vcsa[0−63]

VERSIONS

Introduced with Linux 1.1.92.

EXAMPLES

You may do a screendump on vt3 by switching to vt1 and typing

```
cat /dev/vcs3 >foo
```

Note that the output does not contain newline characters, so some processing may be required, like in

```
fold −w 81 /dev/vcs3 | lpr
```

or (horrors)

```
setterm −dump 3 −file /proc/self/fd/1
```

The */dev/vcsa0* device is used for Braille support.

This program displays the character and screen attributes under the cursor of the second virtual console, then changes the background color there:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/vt.h>

int
main(void)
{
    int fd;
    char *device = "/dev/vcsa2";
    char *console = "/dev/tty2";
    struct {unsigned char lines, cols, x, y;} scrn;
```

```
        unsigned short s;
        unsigned short mask;
        unsigned char attrib;
        int ch;

        fd = open(console, O_RDWR);
        if (fd < 0) {
            perror(console);
            exit(EXIT_FAILURE);
        }
        if (ioctl(fd, VT_GETHIFONTMASK, &mask) < 0) {
            perror("VT_GETHIFONTMASK");
            exit(EXIT_FAILURE);
        }
        (void) close(fd);
        fd = open(device, O_RDWR);
        if (fd < 0) {
            perror(device);
            exit(EXIT_FAILURE);
        }
        (void) read(fd, &scrn, 4);
        (void) lseek(fd, 4 + 2*(scrn.y*scrn.cols + scrn.x), SEEK_SET);
        (void) read(fd, &s, 2);
        ch = s & 0xff;
        if (s & mask)
            ch |= 0x100;
        attrib = ((s & ~mask) >> 8);
        printf("ch=%#03x attrib=%#02x\n", ch, attrib);
        s ^= 0x1000;
        (void) lseek(fd, -2, SEEK_CUR);
        (void) write(fd, &s, 2);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ioctl_console(2), *tty*(4), *ttyS*(4), *gpm*(8)

NAME

veth − Virtual Ethernet Device

DESCRIPTION

The **veth** devices are virtual Ethernet devices. They can act as tunnels between network namespaces to create a bridge to a physical network device in another namespace, but can also be used as standalone network devices.

veth devices are always created in interconnected pairs. A pair can be created using the command:

```
# ip link add <p1-name> type veth peer name <p2-name>
```

In the above, *p1-name* and *p2-name* are the names assigned to the two connected end points.

Packets transmitted on one device in the pair are immediately received on the other device. When either device is down, the link state of the pair is down.

veth device pairs are useful for combining the network facilities of the kernel together in interesting ways. A particularly interesting use case is to place one end of a **veth** pair in one network namespace and the other end in another network namespace, thus allowing communication between network namespaces. To do this, one can provide the **netns** parameter when creating the interfaces:

```
# ip link add <p1−name> netns <p1−ns> type veth peer <p2−name> netns <p2−ns>
```

or, for an existing **veth** pair, move one side to the other namespace:

```
# ip link set <p2−name> netns <p2−ns>
```

ethtool(8) can be used to find the peer of a **veth** network interface, using commands something like:

```
# ip link add ve_A type veth peer name ve_B   # Create veth pair
# ethtool −S ve_A         # Discover interface index of peer
NIC statistics:
     peer_ifindex: 16
# ip link | grep '^16:'   # Look up interface
16: ve_B@ve_A: <BROADCAST,MULTICAST,M−DOWN> mtu 1500 qdisc ...
```

SEE ALSO

clone(2), *network_namespaces*(7), *ip*(8), *ip-link*(8), *ip-netns*(8)

NAME

wavelan − AT&T GIS WaveLAN ISA device driver

SYNOPSIS

insmod wavelan_cs.o [io=*B,B..***] [irq=***I,I..***] [name=***N,N..***]**

DESCRIPTION

This driver is obsolete: it was removed in Linux 2.6.35.

wavelan is the low-level device driver for the NCR / AT&T / Lucent **WaveLAN ISA** and Digital (DEC) **RoamAbout DS** wireless ethernet adapter. This driver is available as a module or might be compiled in the kernel. This driver supports multiple cards in both forms (up to 4) and allocates the next available ethernet device (eth0..eth#) for each card found, unless a device name is explicitly specified (see below). This device name will be reported in the kernel log file with the MAC address, NWID, and frequency used by the card.

Parameters

This section applies to the module form (parameters passed on the *insmod*(8) command line). If the driver is included in the kernel, use the *ether=IRQ,IO,NAME* syntax on the kernel command line.

io Specify the list of base addresses where to search for wavelan cards (setting by dip switch on the card). If you don't specify any io address, the driver will scan 0x390 and 0x3E0 addresses, which might conflict with other hardware...

irq Set the list of IRQs that each wavelan card should use (the value is saved in permanent storage for future use).

name Set the list of names to be used for each wavelan card device (name used by *ifconfig*(8)).

Wireless extensions

Use *iwconfig*(8) to manipulate wireless extensions.

NWID (or domain)

Set the network ID [*0* to *FFFF*] or disable it [*off*]. As the NWID is stored in the card Permanent Storage Area, it will be reused at any further invocation of the driver.

Frequency & channels

For the 2.4 GHz 2.00 Hardware, you are able to set the frequency by specifying one of the 10 defined channels (*2.412, 2.422, 2.425, 2.4305, 2.432, 2.442, 2.452, 2.460, 2.462* or *2.484*) or directly as a numeric value. The frequency is changed immediately and permanently. Frequency availability depends on the regulations...

Statistics spy

Set a list of MAC addresses in the driver (up to 8) and get the last quality of link for each of those (see *iwspy*(8)).

/proc/net/wireless

status is the status reported by the modem. *Link quality* reports the quality of the modulation on the air (direct sequence spread spectrum) [max = 16]. *Level* and *Noise* refer to the signal level and noise level [max = 64]. The *crypt discarded packet* and *misc discarded packet* counters are not implemented.

Private ioctl

You may use *iwpriv*(8) to manipulate private ioctls.

Quality and level threshold

Enables you to define the quality and level threshold used by the modem (packet below that level are discarded).

Histogram

This functionality makes it possible to set a number of signal level intervals and to count the number of packets received in each of those defined intervals. This distribution might be used to calculate the mean value and standard deviation of the signal level.

Specific notes

This driver fails to detect some **non-NCR/AT&T/Lucent** Wavelan cards. If this happens for you, you must look in the source code on how to add your card to the detection routine.

Some of the mentioned features are optional. You may enable or disable them by changing flags in the driver header and recompile.

SEE ALSO

wavelan_cs(4), *ifconfig*(8), *insmod*(8), *iwconfig*(8), *iwpriv*(8), *iwspy*(8)

NAME

intro − introduction to file formats and filesystems

DESCRIPTION

Section 5 of the manual describes various file formats, as well as the corresponding C structures, if any.

In addition, this section contains a number of pages that document various filesystems.

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

SEE ALSO

standards(7)

NAME

acct − process accounting file

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

If the kernel is built with the process accounting option enabled (**CONFIG_BSD_PROCESS_ACCT**), then calling *acct*(2) starts process accounting, for example:

```
acct("/var/log/pacct");
```

When process accounting is enabled, the kernel writes a record to the accounting file as each process on the system terminates. This record contains information about the terminated process, and is defined in <*sys/acct.h*> as follows:

```
#define ACCT_COMM 16

typedef u_int16_t comp_t;

struct acct {
    char ac_flag;           /* Accounting flags */
    u_int16_t ac_uid;       /* Accounting user ID */
    u_int16_t ac_gid;       /* Accounting group ID */
    u_int16_t ac_tty;       /* Controlling terminal */
    u_int32_t ac_btime;     /* Process creation time
                               (seconds since the Epoch) */
    comp_t    ac_utime;     /* User CPU time */
    comp_t    ac_stime;     /* System CPU time */
    comp_t    ac_etime;     /* Elapsed time */
    comp_t    ac_mem;       /* Average memory usage (kB) */
    comp_t    ac_io;        /* Characters transferred (unused) */
    comp_t    ac_rw;        /* Blocks read or written (unused) */
    comp_t    ac_minflt;    /* Minor page faults */
    comp_t    ac_majflt;    /* Major page faults */
    comp_t    ac_swaps;     /* Number of swaps (unused) */
    u_int32_t ac_exitcode;  /* Process termination status
                               (see wait(2)) */
    char      ac_comm[ACCT_COMM+1];
                            /* Command name (basename of last
                               executed command; null-terminated) */
    char      ac_pad[X];    /* padding bytes */
};

enum {          /* Bits that may be set in ac_flag field */
    AFORK = 0x01,           /* Has executed fork, but no exec */
    ASU   = 0x02,           /* Used superuser privileges */
    ACORE = 0x08,           /* Dumped core */
    AXSIG = 0x10            /* Killed by a signal */
};
```

The *comp_t* data type is a floating-point value consisting of a 3-bit, base-8 exponent, and a 13-bit mantissa. A value, *c*, of this type can be converted to a (long) integer as follows:

```
v = (c & 0x1fff) << (((c >> 13) & 0x7) * 3);
```

The *ac_utime*, *ac_stime*, and *ac_etime* fields measure time in "clock ticks"; divide these values by *sysconf(_SC_CLK_TCK)* to convert them to seconds.

Version 3 accounting file format

Since Linux 2.6.8, an optional alternative version of the accounting file can be produced if the **CONFIG_BSD_PROCESS_ACCT_V3** option is set when building the kernel. With this option is set, the records written to the accounting file contain additional fields, and the width of *c_uid* and *ac_gid* fields is widened from 16 to 32 bits (in line with the increased size of UID and GIDs in Linux 2.4 and later).

The records are defined as follows:

```
struct acct_v3 {
    char      ac_flag;      /* Flags */
    char      ac_version;   /* Always set to ACCT_VERSION (3) */
    u_int16_t ac_tty;       /* Controlling terminal */
    u_int32_t ac_exitcode;  /* Process termination status */
    u_int32_t ac_uid;       /* Real user ID */
    u_int32_t ac_gid;       /* Real group ID */
    u_int32_t ac_pid;       /* Process ID */
    u_int32_t ac_ppid;      /* Parent process ID */
    u_int32_t ac_btime;     /* Process creation time */
    float     ac_etime;     /* Elapsed time */
    comp_t    ac_utime;     /* User CPU time */
    comp_t    ac_stime;     /* System time */
    comp_t    ac_mem;       /* Average memory usage (kB) */
    comp_t    ac_io;        /* Characters transferred (unused) */
    comp_t    ac_rw;        /* Blocks read or written
                               (unused) */
    comp_t    ac_minflt;    /* Minor page faults */
    comp_t    ac_majflt;    /* Major page faults */
    comp_t    ac_swaps;     /* Number of swaps (unused) */
    char      ac_comm[ACCT_COMM]; /* Command name */
};
```

VERSIONS

Although it is present on most systems, it is not standardized, and the details vary somewhat between systems.

STANDARDS

None.

HISTORY

glibc 2.6.

Process accounting originated on BSD.

NOTES

Records in the accounting file are ordered by termination time of the process.

Up to and including Linux 2.6.9, a separate accounting record is written for each thread created using the NPTL threading library; since Linux 2.6.10, a single accounting record is written for the entire process on termination of the last thread in the process.

The */proc/sys/kernel/acct* file, described in *proc*(5), defines settings that control the behavior of process accounting when disk space runs low.

SEE ALSO

lastcomm(1), *acct*(2), *accton*(8), *sa*(8)

NAME

charmap − character set description file

DESCRIPTION

A character set description (charmap) defines all available characters and their encodings in a character set. *localedef*(1) can use charmaps to create locale variants for different character sets.

Syntax

The charmap file starts with a header that may consist of the following keywords:

<*code_set_name*>
: is followed by the name of the character map.

<*comment_char*>
: is followed by a character that will be used as the comment character for the rest of the file. It defaults to the number sign (#).

<*escape_char*>
: is followed by a character that should be used as the escape character for the rest of the file to mark characters that should be interpreted in a special way. It defaults to the backslash (\).

<*mb_cur_max*>
: is followed by the maximum number of bytes for a character. The default value is 1.

<*mb_cur_min*>
: is followed by the minimum number of bytes for a character. This value must be less than or equal than <*mb_cur_max*>. If not specified, it defaults to <*mb_cur_max*>.

The character set definition section starts with the keyword *CHARMAP* in the first column.

The following lines may have one of the two following forms to define the character set:

<*character*> *byte-sequence comment*
: This form defines exactly one character and its byte sequence, *comment* being optional.

<*character*>..<*character*> *byte-sequence comment*
: This form defines a character range and its byte sequence, *comment* being optional.

The character set definition section ends with the string *END CHARMAP*.

The character set definition section may optionally be followed by a section to define widths of characters.

The *WIDTH_DEFAULT* keyword can be used to define the default width for all characters not explicitly listed. The default character width is 1.

The width section for individual characters starts with the keyword *WIDTH* in the first column.

The following lines may have one of the two following forms to define the widths of the characters:

<*character*> *width*
: This form defines the width of exactly one character.

<*character*>...<*character*> *width*
: This form defines the width for all the characters in the range.

The width definition section ends with the string *END WIDTH*.

FILES

/usr/share/i18n/charmaps
: Usual default character map path.

STANDARDS

POSIX.2.

EXAMPLES

The Euro sign is defined as follows in the *UTF−8* charmap:

```
<U20AC>     /xe2/x82/xac EURO SIGN
```

SEE ALSO

iconv(1), *locale*(1), *localedef*(1), *locale*(5), *charsets*(7)

NAME

core − core dump file

DESCRIPTION

The default action of certain signals is to cause a process to terminate and produce a *core dump file*, a file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., *gdb*(1)) to inspect the state of the program at the time that it terminated. A list of the signals which cause a process to dump core can be found in *signal*(7).

A process can set its soft **RLIMIT_CORE** resource limit to place an upper limit on the size of the core dump file that will be produced if it receives a "core dump" signal; see *getrlimit*(2) for details.

There are various circumstances in which a core dump file is not produced:

- The process does not have permission to write the core file. (By default, the core file is called *core* or *core.pid*, where *pid* is the ID of the process that dumped core, and is created in the current working directory. See below for details on naming.) Writing the core file fails if the directory in which it is to be created is not writable, or if a file with the same name exists and is not writable or is not a regular file (e.g., it is a directory or a symbolic link).
- A (writable, regular) file with the same name as would be used for the core dump already exists, but there is more than one hard link to that file.
- The filesystem where the core dump file would be created is full; or has run out of inodes; or is mounted read-only; or the user has reached their quota for the filesystem.
- The directory in which the core dump file is to be created does not exist.
- The **RLIMIT_CORE** (core file size) or **RLIMIT_FSIZE** (file size) resource limits for the process are set to zero; see *getrlimit*(2) and the documentation of the shell's *ulimit* command (*limit* in *csh*(1)). However, **RLIMIT_CORE** will be ignored if the system is configured to pipe core dumps to a program.
- The binary being executed by the process does not have read permission enabled. (This is a security measure to ensure that an executable whose contents are not readable does not produce a—possibly readable—core dump containing an image of the executable.)
- The process is executing a set-user-ID (set-group-ID) program that is owned by a user (group) other than the real user (group) ID of the process, or the process is executing a program that has file capabilities (see *capabilities*(7)). (However, see the description of the *prctl*(2) **PR_SET_DUMPABLE** operation, and the description of the */proc/sys/fs/suid_dumpable* file in *proc*(5).)
- */proc/sys/kernel/core_pattern* is empty and */proc/sys/kernel/core_uses_pid* contains the value 0. (These files are described below.) Note that if */proc/sys/kernel/core_pattern* is empty and */proc/sys/kernel/core_uses_pid* contains the value 1, core dump files will have names of the form *.pid*, and such files are hidden unless one uses the *ls*(1) *−a* option.
- (Since Linux 3.7) The kernel was configured without the **CONFIG_COREDUMP** option.

In addition, a core dump may exclude part of the address space of the process if the *madvise*(2) **MADV_DONTDUMP** flag was employed.

On systems that employ *systemd*(1) as the *init* framework, core dumps may instead be placed in a location determined by *systemd*(1)See below for further details.

Naming of core dump files

By default, a core dump file is named *core*, but the */proc/sys/kernel/core_pattern* file (since Linux 2.6 and 2.4.21) can be set to define a template that is used to name core dump files. The template can contain % specifiers which are substituted by the following values when a core file is created:

%%
: A single % character.

%c
: Core file size soft resource limit of crashing process (since Linux 2.6.24).

%d
: Dump mode—same as value returned by *prctl*(2) **PR_GET_DUMPABLE** (since Linux 3.7).

%e
: The process or thread's *comm* value, which typically is the same as the executable filename (without path prefix, and truncated to a maximum of 15 characters), but may have been modified to be something different; see the discussion of */proc/*pid*/comm* and */proc/*pid*/task/*tid*/comm* in *proc*(5).

%E Pathname of executable, with slashes ('/') replaced by exclamation marks ('!') (since Linux 3.0).
%g Numeric real GID of dumped process.
%h Hostname (same as *nodename* returned by **uname**(2)).
%i TID of thread that triggered core dump, as seen in the PID namespace in which the thread resides (since Linux 3.18).
%I TID of thread that triggered core dump, as seen in the initial PID namespace (since Linux 3.18).
%p PID of dumped process, as seen in the PID namespace in which the process resides.
%P PID of dumped process, as seen in the initial PID namespace (since Linux 3.12).
%s Number of signal causing dump.
%t Time of dump, expressed as seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
%u Numeric real UID of dumped process.

A single % at the end of the template is dropped from the core filename, as is the combination of a % followed by any character other than those listed above. All other characters in the template become a literal part of the core filename. The template may include '/' characters, which are interpreted as delimiters for directory names. The maximum size of the resulting core filename is 128 bytes (64 bytes before Linux 2.6.19). The default value in this file is "core". For backward compatibility, if */proc/sys/kernel/core_pattern* does not include *%p* and */proc/sys/kernel/core_uses_pid* (see below) is nonzero, then .PID will be appended to the core filename.

Paths are interpreted according to the settings that are active for the crashing process. That means the crashing process's mount namespace (see *mount_namespaces*(7)), its current working directory (found via *getcwd*(2)), and its root directory (see *chroot*(2)).

Since Linux 2.4, Linux has also provided a more primitive method of controlling the name of the core dump file. If the */proc/sys/kernel/core_uses_pid* file contains the value 0, then a core dump file is simply named *core*. If this file contains a nonzero value, then the core dump file includes the process ID in a name of the form *core.PID*.

Since Linux 3.6, if */proc/sys/fs/suid_dumpable* is set to 2 ("suidsafe"), the pattern must be either an absolute pathname (starting with a leading '/' character) or a pipe, as defined below.

Piping core dumps to a program

Since Linux 2.6.19, Linux supports an alternate syntax for the */proc/sys/kernel/core_pattern* file. If the first character of this file is a pipe symbol (|), then the remainder of the line is interpreted as the command-line for a user-space program (or script) that is to be executed.

Since Linux 5.3.0, the pipe template is split on spaces into an argument list *before* the template parameters are expanded. In earlier kernels, the template parameters are expanded first and the resulting string is split on spaces into an argument list. This means that in earlier kernels executable names added by the *%e* and *%E* template parameters could get split into multiple arguments. So the core dump handler needs to put the executable names as the last argument and ensure it joins all parts of the executable name using spaces. Executable names with multiple spaces in them are not correctly represented in earlier kernels, meaning that the core dump handler needs to use mechanisms to find the executable name.

Instead of being written to a file, the core dump is given as standard input to the program. Note the following points:

- The program must be specified using an absolute pathname (or a pathname relative to the root directory, */*), and must immediately follow the '|' character.
- The command-line arguments can include any of the % specifiers listed above. For example, to pass the PID of the process that is being dumped, specify *%p* in an argument.
- The process created to run the program runs as user and group *root*.
- Running as *root* does not confer any exceptional security bypasses. Namely, LSMs (e.g., SELinux) are still active and may prevent the handler from accessing details about the crashed process via */proc/*pid.
- The program pathname is interpreted with respect to the initial mount namespace as it is always executed there. It is not affected by the settings (e.g., root directory, mount namespace, current working directory) of the crashing process.

- The process runs in the initial namespaces (PID, mount, user, and so on) and not in the namespaces of the crashing process. One can utilize specifiers such as *%P* to find the right */proc/*pid directory and probe/enter the crashing process's namespaces if needed.
- The process starts with its current working directory as the root directory. If desired, it is possible change to the working directory of the dumping process by employing the value provided by the *%P* specifier to change to the location of the dumping process via */proc/*pid*/cwd*.
- Command-line arguments can be supplied to the program (since Linux 2.6.24), delimited by white space (up to a total line length of 128 bytes).
- The **RLIMIT_CORE** limit is not enforced for core dumps that are piped to a program via this mechanism.

/proc/sys/kernel/core_pipe_limit

When collecting core dumps via a pipe to a user-space program, it can be useful for the collecting program to gather data about the crashing process from that process's */proc/*pid directory. In order to do this safely, the kernel must wait for the program collecting the core dump to exit, so as not to remove the crashing process's */proc/*pid files prematurely. This in turn creates the possibility that a misbehaving collecting program can block the reaping of a crashed process by simply never exiting.

Since Linux 2.6.32, the */proc/sys/kernel/core_pipe_limit* can be used to defend against this possibility. The value in this file defines how many concurrent crashing processes may be piped to user-space programs in parallel. If this value is exceeded, then those crashing processes above this value are noted in the kernel log and their core dumps are skipped.

A value of 0 in this file is special. It indicates that unlimited processes may be captured in parallel, but that no waiting will take place (i.e., the collecting program is not guaranteed access to */proc/<crashing−PID>*). The default value for this file is 0.

Controlling which mappings are written to the core dump

Since Linux 2.6.23, the Linux-specific */proc/*pid*/coredump_filter* file can be used to control which memory segments are written to the core dump file in the event that a core dump is performed for the process with the corresponding process ID.

The value in the file is a bit mask of memory mapping types (see *mmap*(2)). If a bit is set in the mask, then memory mappings of the corresponding type are dumped; otherwise they are not dumped. The bits in this file have the following meanings:

bit 0 Dump anonymous private mappings.
bit 1 Dump anonymous shared mappings.
bit 2 Dump file-backed private mappings.
bit 3 Dump file-backed shared mappings.
bit 4 (since Linux 2.6.24)
Dump ELF headers.
bit 5 (since Linux 2.6.28)
Dump private huge pages.
bit 6 (since Linux 2.6.28)
Dump shared huge pages.
bit 7 (since Linux 4.4)
Dump private DAX pages.
bit 8 (since Linux 4.4)
Dump shared DAX pages.

By default, the following bits are set: 0, 1, 4 (if the **CONFIG_CORE_DUMP_DEFAULT_ELF_HEADERS** kernel configuration option is enabled), and 5. This default can be modified at boot time using the *coredump_filter* boot option.

The value of this file is displayed in hexadecimal. (The default value is thus displayed as 33.)

Memory-mapped I/O pages such as frame buffer are never dumped, and virtual DSO (**vdso**(7)) pages are always dumped, regardless of the *coredump_filter* value.

A child process created via *fork*(2) inherits its parent's *coredump_filter* value; the *coredump_filter* value is preserved across an *execve*(2).

It can be useful to set *coredump_filter* in the parent shell before running a program, for example:

```
$ echo 0x7 > /proc/self/coredump_filter
$ ./some_program
```

This file is provided only if the kernel was built with the **CONFIG_ELF_CORE** configuration option.

Core dumps and systemd

On systems using the *systemd*(1) *init* framework, core dumps may be placed in a location determined by *systemd*(1)To do this, *systemd*(1) employs the *core_pattern* feature that allows piping core dumps to a program. One can verify this by checking whether core dumps are being piped to the *systemd-coredump*(8) program:

```
$ cat /proc/sys/kernel/core_pattern
|/usr/lib/systemd/systemd-coredump %P %u %g %s %t %c %e
```

In this case, core dumps will be placed in the location configured for *systemd-coredump*(8), typically as *lz4*(1) compressed files in the directory */var/lib/systemd/coredump/*. One can list the core dumps that have been recorded by *systemd-coredump*(8) using *coredumpctl*(1):

```
$ coredumpctl list | tail -5
Wed 2017-10-11 22:25:30 CEST  2748 1000 1000 3 present  /usr/bin/sleep
Thu 2017-10-12 06:29:10 CEST  2716 1000 1000 3 present  /usr/bin/sleep
Thu 2017-10-12 06:30:50 CEST  2767 1000 1000 3 present  /usr/bin/sleep
Thu 2017-10-12 06:37:40 CEST  2918 1000 1000 3 present  /usr/bin/cat
Thu 2017-10-12 08:13:07 CEST  2955 1000 1000 3 present  /usr/bin/cat
```

The information shown for each core dump includes the date and time of the dump, the PID, UID, and GID of the dumping process, the signal number that caused the core dump, and the pathname of the executable that was being run by the dumped process. Various options to *coredumpctl*(1) allow a specified coredump file to be pulled from the *systemd*(1) location into a specified file. For example, to extract the core dump for PID 2955 shown above to a file named *core* in the current directory, one could use:

```
$ coredumpctl dump 2955 -o core
```

For more extensive details, see the *coredumpctl*(1) manual page.

To (persistently) disable the *systemd*(1) mechanism that archives core dumps, restoring to something more like traditional Linux behavior, one can set an override for the *systemd*(1) mechanism, using something like:

```
# echo "kernel.core_pattern=core.%p" > \
                /etc/sysctl.d/50-coredump.conf
# /lib/systemd/systemd-sysctl
```

It is also possible to temporarily (i.e., until the next reboot) change the *core_pattern* setting using a command such as the following (which causes the names of core dump files to include the executable name as well as the number of the signal which triggered the core dump):

```
# sysctl -w kernel.core_pattern="%e-%s.core"
```

NOTES

The *gdb*(1) *gcore* command can be used to obtain a core dump of a running process.

In Linux versions up to and including 2.6.27, if a multithreaded process (or, more precisely, a process that shares its memory with another process by being created with the **CLONE_VM** flag of *clone*(2)) dumps core, then the process ID is always appended to the core filename, unless the process ID was already included elsewhere in the filename via a *%p* specification in */proc/sys/kernel/core_pattern*. (This is primarily useful when employing the obsolete LinuxThreads implementation, where each thread of a process has a different PID.)

EXAMPLES

The program below can be used to demonstrate the use of the pipe syntax in the */proc/sys/kernel/core_pattern* file. The following shell session demonstrates the use of this program (compiled to create an executable named *core_pattern_pipe_test*):

```
$ cc -o core_pattern_pipe_test core_pattern_pipe_test.c
$ su
Password:
```

```
# echo "|$PWD/core_pattern_pipe_test %p UID=%u GID=%g sig=%s" > \
    /proc/sys/kernel/core_pattern
# exit
$ sleep 100
^\                      # type control-backslash
Quit (core dumped)
$ cat core.info
argc=5
argc[0]=</home/mtk/core_pattern_pipe_test>
argc[1]=<20575>
argc[2]=<UID=1000>
argc[3]=<GID=100>
argc[4]=<sig=3>
Total bytes in core dump: 282624
```

Program source

```
/* core_pattern_pipe_test.c */

#define _GNU_SOURCE
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    ssize_t nread, tot;
    char buf[BUF_SIZE];
    FILE *fp;
    char cwd[PATH_MAX];

    /* Change our current working directory to that of the
       crashing process. */

    snprintf(cwd, PATH_MAX, "/proc/%s/cwd", argv[1]);
    chdir(cwd);

    /* Write output to file "core.info" in that directory. */

    fp = fopen("core.info", "w+");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    /* Display command-line arguments given to core_pattern
       pipe program. */

    fprintf(fp, "argc=%d\n", argc);
    for (size_t j = 0; j < argc; j++)
        fprintf(fp, "argc[%zu]=<%s>\n", j, argv[j]);

    /* Count bytes in standard input (the core dump). */

    tot = 0;
```

```
        while ((nread = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
            tot += nread;
        fprintf(fp, "Total bytes in core dump: %zd\n", tot);

        fclose(fp);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

bash(1), *coredumpctl*(1), *gdb*(1), *getrlimit*(2), *mmap*(2), *prctl*(2), *sigaction*(2), *elf*(5), *proc*(5), *pthreads*(7), *signal*(7), *systemd-coredump*(8)

NAME

dir_colors − configuration file for dircolors(1)

DESCRIPTION

The program *ls*(1) uses the environment variable **LS_COLORS** to determine the colors in which the filenames are to be displayed. This environment variable is usually set by a command like

eval `dircolors some_path/dir_colors`

found in a system default shell initialization file, like */etc/profile* or */etc/csh.cshrc*. (See also *dircolors*(1)Usually, the file used here is */etc/DIR_COLORS* and can be overridden by a *.dir_colors* file in one's home directory.

This configuration file consists of several statements, one per line. Anything right of a hash mark (#) is treated as a comment, if the hash mark is at the beginning of a line or is preceded by at least one whitespace. Blank lines are ignored.

The *global* section of the file consists of any statement before the first **TERM** statement. Any statement in the global section of the file is considered valid for all terminal types. Following the global section is one or more *terminal-specific* sections, preceded by one or more **TERM** statements which specify the terminal types (as given by the **TERM** environment variable) the following declarations apply to. It is always possible to override a global declaration by a subsequent terminal-specific one.

The following statements are recognized; case is insignificant:

TERM *terminal-type*
: Starts a terminal-specific section and specifies which terminal it applies to. Multiple **TERM** statements can be used to create a section which applies for several terminal types.

COLOR yes|all|no|none|tty
: (Slackware only; ignored by GNU *dircolors*(1)Specifies that colorization should always be enabled (*yes* or *all*), never enabled (*no* or *none*), or enabled only if the output is a terminal (*tty*). The default is *no*.

EIGHTBIT yes|no
: (Slackware only; ignored by GNU *dircolors*(1)Specifies that eight-bit ISO/IEC 8859 characters should be enabled by default. For compatibility reasons, this can also be specified as 1 for *yes* or 0 for *no*. The default is *no*.

OPTIONS *options*
: (Slackware only; ignored by GNU *dircolors*(1)Adds command-line options to the default **ls** command line. The options can be any valid **ls** command-line options, and should include the leading minus sign. Note that **dircolors** does not verify the validity of these options.

NORMAL *color-sequence*
: Specifies the color used for normal (nonfilename) text.

  Synonym: **NORM**.

FILE *color-sequence*
: Specifies the color used for a regular file.

DIR *color-sequence*
: Specifies the color used for directories.

LINK *color-sequence*
: Specifies the color used for a symbolic link.

  Synonyms: **LNK**, **SYMLINK**.

ORPHAN *color-sequence*
: Specifies the color used for an orphaned symbolic link (one which points to a nonexistent file). If this is unspecified, **ls** will use the **LINK** color instead.

MISSING *color-sequence*
: Specifies the color used for a missing file (a nonexistent file which nevertheless has a symbolic link pointing to it). If this is unspecified, **ls** will use the **FILE** color instead.

FIFO *color-sequence*
Specifies the color used for a FIFO (named pipe).

Synonym: **PIPE**.

SOCK *color-sequence*
Specifies the color used for a socket.

DOOR *color-sequence*
(Supported since fileutils 4.1) Specifies the color used for a door (Solaris 2.5 and later).

BLK *color-sequence*
Specifies the color used for a block device special file.

Synonym: **BLOCK**.

CHR *color-sequence*
Specifies the color used for a character device special file.

Synonym: **CHAR**.

EXEC *color-sequence*
Specifies the color used for a file with the executable attribute set.

SUID *color-sequence*
Specifies the color used for a file with the set-user-ID attribute set.

Synonym: **SETUID**.

SGID *color-sequence*
Specifies the color used for a file with the set-group-ID attribute set.

Synonym: **SETGID**.

STICKY *color-sequence*
Specifies the color used for a directory with the sticky attribute set.

STICKY_OTHER_WRITABLE *color-sequence*
Specifies the color used for an other-writable directory with the executable attribute set.

Synonym: **OWT**.

OTHER_WRITABLE *color-sequence*
Specifies the color used for an other-writable directory without the executable attribute set.

Synonym: **OWR**.

LEFTCODE *color-sequence*
Specifies the *left code* for non-ISO/IEC 6429 terminals (see below).

Synonym: **LEFT**.

RIGHTCODE *color-sequence*
Specifies the *right code* for non-ISO/IEC 6429 terminals (see below).

Synonym: **RIGHT**.

ENDCODE *color-sequence*
Specifies the *end code* for non-ISO/IEC 6429 terminals (see below).

Synonym: **END**.

*******extension color-sequence*
Specifies the color used for any file that ends in *extension*.

.*extension color-sequence*
Same as *******.extension*. Specifies the color used for any file that ends in *.extension*. Note that the period is included in the extension, which makes it impossible to specify an extension not starting with a period, such as **~** for **emacs** backup files. This form should be considered obsolete.

ISO/IEC 6429 (ANSI) color sequences
Most color-capable ASCII terminals today use ISO/IEC 6429 (ANSI) color sequences, and many common terminals without color capability, including **xterm** and the widely used and cloned DEC VT100,

will recognize ISO/IEC 6429 color codes and harmlessly eliminate them from the output or emulate them. **ls** uses ISO/IEC 6429 codes by default, assuming colorization is enabled.

ISO/IEC 6429 color sequences are composed of sequences of numbers separated by semicolons. The most common codes are:

0	to restore default color
1	for brighter colors
4	for underlined text
5	for flashing text
30	for black foreground
31	for red foreground
32	for green foreground
33	for yellow (or brown) foreground
34	for blue foreground
35	for purple foreground
36	for cyan foreground
37	for white (or gray) foreground
40	for black background
41	for red background
42	for green background
43	for yellow (or brown) background
44	for blue background
45	for purple background
46	for cyan background
47	for white (or gray) background

Not all commands will work on all systems or display devices.

ls uses the following defaults:

NORMAL	0	Normal (nonfilename) text
FILE	0	Regular file
DIR	32	Directory
LINK	36	Symbolic link
ORPHAN	undefined	Orphaned symbolic link
MISSING	undefined	Missing file
FIFO	31	Named pipe (FIFO)
SOCK	33	Socket
BLK	44;37	Block device
CHR	44;37	Character device
EXEC	35	Executable file

A few terminal programs do not recognize the default properly. If all text gets colorized after you do a directory listing, change the **NORMAL** and **FILE** codes to the numerical codes for your normal foreground and background colors.

Other terminal types (advanced configuration)

If you have a color-capable (or otherwise highlighting) terminal (or printer!) which uses a different set of codes, you can still generate a suitable setup. To do so, you will have to use the **LEFTCODE**, **RIGHTCODE**, and **ENDCODE** definitions.

When writing out a filename, **ls** generates the following output sequence: **LEFTCODE** *typecode* **RIGHTCODE** *filename* **ENDCODE**, where the *typecode* is the color sequence that depends on the type or name of file. If the **ENDCODE** is undefined, the sequence **LEFTCODE NORMAL RIGHTCODE** will be used instead. The purpose of the left- and rightcodes is merely to reduce the amount of typing necessary (and to hide ugly escape codes away from the user). If they are not appropriate for your terminal, you can eliminate them by specifying the respective keyword on a line by itself.

NOTE: If the **ENDCODE** is defined in the global section of the setup file, it *cannot* be undefined in a terminal-specific section of the file. This means any **NORMAL** definition will have no effect. A different **ENDCODE** can, however, be specified, which would have the same effect.

Escape sequences

To specify control- or blank characters in the color sequences or filename extensions, either C-style \-escaped notation or **stty**-style ^-notation can be used. The C-style notation includes the following characters:

\a	Bell (ASCII 7)
\b	Backspace (ASCII 8)
\e	Escape (ASCII 27)
\f	Form feed (ASCII 12)
\n	Newline (ASCII 10)
\r	Carriage Return (ASCII 13)
\t	Tab (ASCII 9)
\v	Vertical Tab (ASCII 11)
\?	Delete (ASCII 127)
nnn	Any character (octal notation)
\x*nnn*	Any character (hexadecimal notation)
_	Space
\\	Backslash (\)
\^	Caret (^)
\#	Hash mark (#)

Note that escapes are necessary to enter a space, backslash, caret, or any control character anywhere in the string, as well as a hash mark as the first character.

FILES

/etc/DIR_COLORS
System-wide configuration file.

~/.dir_colors
Per-user configuration file.

This page describes the **dir_colors** file format as used in the fileutils-4.1 package; other versions may differ slightly.

NOTES

The default **LEFTCODE** and **RIGHTCODE** definitions, which are used by ISO/IEC 6429 terminals are:

LEFTCODE	\e[
RIGHTCODE	m

The default **ENDCODE** is undefined.

SEE ALSO

dircolors(1), *ls*(1), *stty*(1), *xterm*(1)

NAME

elf − format of Executable and Linking Format (ELF) files

SYNOPSIS

#include <elf.h>

DESCRIPTION

The header file <*elf.h*> defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects.

An executable file using the ELF file format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The program header table and the section header table's offset in the file are defined in the ELF header. The two tables describe the rest of the particularities of the file.

This header file describes the above mentioned headers as C structures and also includes structures for dynamic sections, relocation sections and symbol tables.

Basic types

The following types are used for N-bit architectures (N=32,64, *ElfN* stands for *Elf32* or *Elf64*, *uintN_t* stands for *uint32_t* or *uint64_t*):

```
ElfN_Addr       Unsigned program address, uintN_t
ElfN_Off        Unsigned file offset, uintN_t
ElfN_Section    Unsigned section index, uint16_t
ElfN_Versym     Unsigned version symbol information, uint16_t
Elf_Byte        unsigned char
ElfN_Half       uint16_t
ElfN_Sword      int32_t
ElfN_Word       uint32_t
ElfN_Sxword     int64_t
ElfN_Xword      uint64_t
```

(Note: the *BSD terminology is a bit different. There, *Elf64_Half* is twice as large as *Elf32_Half*, and *Elf64Quarter* is used for *uint16_t*. In order to avoid confusion these types are replaced by explicit ones in the below.)

All data structures that the file format defines follow the "natural" size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so on.

ELF header (Ehdr)

The ELF header is described by the type *Elf32_Ehdr* or *Elf64_Ehdr*:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

The fields have the following meanings:

e_ident This array of bytes specifies how to interpret the file, independent of the processor or the file's remaining contents. Within this array everything is named by macros, which start with the prefix **EI_** and may contain values which start with the prefix **ELF**. The following macros are defined:

EI_MAG0
The first byte of the magic number. It must be filled with **ELFMAG0**. (0: 0x7f)

EI_MAG1
The second byte of the magic number. It must be filled with **ELFMAG1**. (1: 'E')

EI_MAG2
The third byte of the magic number. It must be filled with **ELFMAG2**. (2: 'L')

EI_MAG3
The fourth byte of the magic number. It must be filled with **ELFMAG3**. (3: 'F')

EI_CLASS
The fifth byte identifies the architecture for this binary:

ELFCLASSNONE
This class is invalid.
ELFCLASS32 This defines the 32-bit architecture. It supports machines with files and virtual address spaces up to 4 Gigabytes.
ELFCLASS64 This defines the 64-bit architecture.

EI_DATA
The sixth byte specifies the data encoding of the processor-specific data in the file. Currently, these encodings are supported:

ELFDATANONE
Unknown data format.
ELFDATA2LSB
Two's complement, little-endian.
ELFDATA2MSB
Two's complement, big-endian.

EI_VERSION
The seventh byte is the version number of the ELF specification:

EV_NONE Invalid version.
EV_CURRENT
Current version.

EI_OSABI
The eighth byte identifies the operating system and ABI to which the object is targeted. Some fields in other ELF structures have flags and values that have platform-specific meanings; the interpretation of those fields is determined by the value of this byte. For example:

ELFOSABI_NONE Same as ELFOSABI_SYSV
ELFOSABI_SYSV UNIX System V ABI
ELFOSABI_HPUX HP-UX ABI
ELFOSABI_NETBSD NetBSD ABI
ELFOSABI_LINUX Linux ABI
ELFOSABI_SOLARIS Solaris ABI
ELFOSABI_IRIX IRIX ABI
ELFOSABI_FREEBSD
FreeBSD ABI
ELFOSABI_TRU64 TRU64 UNIX ABI
ELFOSABI_ARM ARM architecture ABI
ELFOSABI_STANDALONE
Stand-alone (embedded) ABI

EI_ABIVERSION
The ninth byte identifies the version of the ABI to which the object is targeted. This field is used to distinguish among incompatible versions of an ABI. The interpretation of this version number is dependent on the ABI identified by the **EI_OSABI** field. Applications conforming to this specification use the value 0.

EI_PAD
Start of padding. These bytes are reserved and set to zero. Programs which read them should ignore them. The value for **EI_PAD** will change in the future if currently unused bytes are given meanings.

EI_NIDENT
The size of the *e_ident* array.

e_type This member of the structure identifies the object file type:

ET_NONE	An unknown type.
ET_REL	A relocatable file.
ET_EXEC	An executable file.
ET_DYN	A shared object.
ET_CORE	A core file.

e_machine
This member specifies the required architecture for an individual file. For example:

EM_NONE	An unknown machine
EM_M32	AT&T WE 32100
EM_SPARC	Sun Microsystems SPARC
EM_386	Intel 80386
EM_68K	Motorola 68000
EM_88K	Motorola 88000
EM_860	Intel 80860
EM_MIPS	MIPS RS3000 (big-endian only)
EM_PARISC	HP/PA
EM_SPARC32PLUS	SPARC with enhanced instruction set
EM_PPC	PowerPC
EM_PPC64	PowerPC 64-bit
EM_S390	IBM S/390
EM_ARM	Advanced RISC Machines
EM_SH	Renesas SuperH
EM_SPARCV9	SPARC v9 64-bit
EM_IA_64	Intel Itanium
EM_X86_64	AMD x86-64
EM_VAX	DEC Vax

e_version
This member identifies the file version:

EV_NONE	Invalid version
EV_CURRENT	Current version

e_entry This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff
This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags This member holds processor-specific flags associated with the file. Flag names take the form EF_'machine_flag'. Currently, no flags have been defined.

e_ehsize
: This member holds the ELF header's size in bytes.

e_phentsize
: This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum
: This member holds the number of entries in the program header table. Thus the product of *e_phentsize* and *e_phnum* gives the table's size in bytes. If a file has no program header, *e_phnum* holds the value zero.

  If the number of entries in the program header table is larger than or equal to **PN_XNUM** (0xffff), this member holds **PN_XNUM** (0xffff) and the real number of entries in the program header table is held in the *sh_info* member of the initial entry in section header table. Otherwise, the *sh_info* member of the initial entry contains the value zero.

  PN_XNUM
  : This is defined as 0xffff, the largest number *e_phnum* can have, specifying where the actual number of program headers is assigned.

e_shentsize
: This member holds a sections header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum
: This member holds the number of entries in the section header table. Thus the product of *e_shentsize* and *e_shnum* gives the section header table's size in bytes. If a file has no section header table, *e_shnum* holds the value of zero.

  If the number of entries in the section header table is larger than or equal to **SHN_LORESERVE** (0xff00), *e_shnum* holds the value zero and the real number of entries in the section header table is held in the *sh_size* member of the initial entry in section header table. Otherwise, the *sh_size* member of the initial entry in the section header table holds the value zero.

e_shstrndx
: This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value **SHN_UNDEF**.

  If the index of section name string table section is larger than or equal to **SHN_LORESERVE** (0xff00), this member holds **SHN_XINDEX** (0xffff) and the real index of the section name string table section is held in the *sh_link* member of the initial entry in section header table. Otherwise, the *sh_link* member of the initial entry in section header table contains the value zero.

Program header (Phdr)

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections*. Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's *e_phentsize* and *e_phnum* members. The ELF program header is described by the type *Elf32_Phdr* or *Elf64_Phdr* depending on the architecture:

```
typedef struct {
    uint32_t   p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    uint32_t   p_filesz;
    uint32_t   p_memsz;
    uint32_t   p_flags;
    uint32_t   p_align;
} Elf32_Phdr;

typedef struct {
```

```
        uint32_t   p_type;
        uint32_t   p_flags;
        Elf64_Off  p_offset;
        Elf64_Addr p_vaddr;
        Elf64_Addr p_paddr;
        uint64_t   p_filesz;
        uint64_t   p_memsz;
        uint64_t   p_align;
    } Elf64_Phdr;
```

The main difference between the 32-bit and the 64-bit program header lies in the location of the *p_flags* member in the total struct.

p_type This member of the structure indicates what kind of segment this array element describes or how to interpret the array element's information.

PT_NULL
The array element is unused and the other members' values are undefined. This lets the program header have ignored entries.

PT_LOAD
The array element specifies a loadable segment, described by *p_filesz* and *p_memsz*. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size *p_memsz* is larger than the file size *p_filesz*, the "extra" bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the *p_vaddr* member.

PT_DYNAMIC
The array element specifies dynamic linking information.

PT_INTERP
The array element specifies the location and size of a null-terminated pathname to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects). However it may not occur more than once in a file. If it is present, it must precede any loadable segment entry.

PT_NOTE
The array element specifies the location of notes (ElfN_Nhdr).

PT_SHLIB
This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

PT_PHDR
The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry.

PT_LOPROC
PT_HIPROC
Values in the inclusive range [**PT_LOPROC**, **PT_HIPROC**] are reserved for processor-specific semantics.

PT_GNU_STACK
GNU extension which is used by the Linux kernel to control the state of the stack via the flags set in the *p_flags* member.

p_offset
This member holds the offset from the beginning of the file at which the first byte of the segment resides.

p_vaddr
This member holds the virtual address at which the first byte of the segment resides in memory.

p_paddr
On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Under BSD this member is not used and must be zero.

p_filesz
This member holds the number of bytes in the file image of the segment. It may be zero.

p_memsz
This member holds the number of bytes in the memory image of the segment. It may be zero.

p_flags This member holds a bit mask of flags relevant to the segment:

PF_X An executable segment.
PF_W A writable segment.
PF_R A readable segment.

A text segment commonly has the flags **PF_X** and **PF_R**. A data segment commonly has **PF_W** and **PF_R**.

p_align
This member holds the value to which the segments are aligned in memory and in the file. Loadable process segments must have congruent values for *p_vaddr* and *p_offset*, modulo the page size. Values of zero and one mean no alignment is required. Otherwise, *p_align* should be a positive, integral power of two, and *p_vaddr* should equal *p_offset*, modulo *p_align*.

Section header (Shdr)
A file's section header table lets one locate all the file's sections. The section header table is an array of *Elf32_Shdr* or *Elf64_Shdr* structures. The ELF header's *e_shoff* member gives the byte offset from the beginning of the file to the section header table. *e_shnum* holds the number of entries the section header table contains. *e_shentsize* holds the size in bytes of each entry.

A section header table index is a subscript into this array. Some section header table indices are reserved: the initial entry and the indices between **SHN_LORESERVE** and **SHN_HIRESERVE**. The initial entry is used in ELF extensions for *e_phnum*, *e_shnum*, and *e_shstrndx*; in other cases, each field in the initial entry is set to zero. An object file does not have sections for these special indices:

SHN_UNDEF
This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference.

SHN_LORESERVE
This value specifies the lower bound of the range of reserved indices.

SHN_LOPROC
SHN_HIPROC
Values greater in the inclusive range [**SHN_LOPROC**, **SHN_HIPROC**] are reserved for processor-specific semantics.

SHN_ABS
This value specifies the absolute value for the corresponding reference. For example, a symbol defined relative to section number **SHN_ABS** has an absolute value and is not affected by relocation.

SHN_COMMON
Symbols defined relative to this section are common symbols, such as FORTRAN COMMON or unallocated C external variables.

SHN_HIRESERVE
This value specifies the upper bound of the range of reserved indices. The system reserves indices between **SHN_LORESERVE** and **SHN_HIRESERVE**, inclusive. The section header table does not contain entries for the reserved indices.

The section header has the following structure:

```
typedef struct {
```

```
    uint32_t   sh_name;
    uint32_t   sh_type;
    uint32_t   sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off  sh_offset;
    uint32_t   sh_size;
    uint32_t   sh_link;
    uint32_t   sh_info;
    uint32_t   sh_addralign;
    uint32_t   sh_entsize;
} Elf32_Shdr;

typedef struct {
    uint32_t   sh_name;
    uint32_t   sh_type;
    uint64_t   sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off  sh_offset;
    uint64_t   sh_size;
    uint32_t   sh_link;
    uint32_t   sh_info;
    uint64_t   sh_addralign;
    uint64_t   sh_entsize;
} Elf64_Shdr;
```

No real differences exist between the 32-bit and 64-bit section headers.

sh_name

This member specifies the name of the section. Its value is an index into the section header string table section, giving the location of a null-terminated string.

sh_type

This member categorizes the section's contents and semantics.

SHT_NULL

This value marks the section header as inactive. It does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS

This section holds information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB

This section holds a symbol table. Typically, **SHT_SYMTAB** provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. An object file can also contain a **SHT_DYNSYM** section.

SHT_STRTAB

This section holds a string table. An object file may have multiple string table sections.

SHT_RELA

This section holds relocation entries with explicit addends, such as type *Elf32_Rela* for the 32-bit class of object files. An object may have multiple relocation sections.

SHT_HASH

This section holds a symbol hash table. An object participating in dynamic linking must contain a symbol hash table. An object file may have only one hash table.

SHT_DYNAMIC

This section holds information for dynamic linking. An object file may have only one dynamic section.

SHT_NOTE
This section holds notes (ElfN_Nhdr).

SHT_NOBITS
A section of this type occupies no space in the file but otherwise resembles **SHT_PROGBITS**. Although this section contains no bytes, the *sh_offset* member contains the conceptual file offset.

SHT_REL
This section holds relocation offsets without explicit addends, such as type *Elf32_Rel* for the 32-bit class of object files. An object file may have multiple relocation sections.

SHT_SHLIB
This section is reserved but has unspecified semantics.

SHT_DYNSYM
This section holds a minimal set of dynamic linking symbols. An object file can also contain a **SHT_SYMTAB** section.

SHT_LOPROC
SHT_HIPROC
Values in the inclusive range [**SHT_LOPROC**, **SHT_HIPROC**] are reserved for processor-specific semantics.

SHT_LOUSER
This value specifies the lower bound of the range of indices reserved for application programs.

SHT_HIUSER
This value specifies the upper bound of the range of indices reserved for application programs. Section types between **SHT_LOUSER** and **SHT_HIUSER** may be used by the application, without conflicting with current or future system-defined section types.

sh_flags
Sections support one-bit flags that describe miscellaneous attributes. If a flag bit is set in *sh_flags*, the attribute is "on" for the section. Otherwise, the attribute is "off" or does not apply. Undefined attributes are set to zero.

SHF_WRITE
This section contains data that should be writable during process execution.

SHF_ALLOC
This section occupies memory during process execution. Some control sections do not reside in the memory image of an object file. This attribute is off for those sections.

SHF_EXECINSTR
This section contains executable machine instructions.

SHF_MASKPROC
All bits included in this mask are reserved for processor-specific semantics.

sh_addr
If this section appears in the memory image of a process, this member holds the address at which the section's first byte should reside. Otherwise, the member contains zero.

sh_offset
This member's value holds the byte offset from the beginning of the file to the first byte in the section. One section type, **SHT_NOBITS**, occupies no space in the file, and its *sh_offset* member locates the conceptual placement in the file.

sh_size This member holds the section's size in bytes. Unless the section type is **SHT_NOBITS**, the section occupies *sh_size* bytes in the file. A section of type **SHT_NOBITS** may have a nonzero size, but it occupies no space in the file.

sh_link This member holds a section header table index link, whose interpretation depends on the section type.

sh_info This member holds extra information, whose interpretation depends on the section type.

sh_addralign
Some sections have address alignment constraints. If a section holds a doubleword, the system must ensure doubleword alignment for the entire section. That is, the value of *sh_addr* must be congruent to zero, modulo the value of *sh_addralign*. Only zero and positive integral powers of two are allowed. The value 0 or 1 means that the section has no alignment constraints.

sh_entsize
Some sections hold a table of fixed-sized entries, such as a symbol table. For such a section, this member gives the size in bytes for each entry. This member contains zero if the section does not hold a table of fixed-size entries.

Various sections hold program and control information:

.bss This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. This section is of type **SHT_NOBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.comment
This section holds version control information. This section is of type **SHT_PROGBITS**. No attribute types are used.

.ctors This section holds initialized pointers to the C++ constructor functions. This section is of type **SHT_PROGBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.data This section holds initialized data that contribute to the program's memory image. This section is of type **SHT_PROGBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.data1 This section holds initialized data that contribute to the program's memory image. This section is of type **SHT_PROGBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.debug This section holds information for symbolic debugging. The contents are unspecified. This section is of type **SHT_PROGBITS**. No attribute types are used.

.dtors This section holds initialized pointers to the C++ destructor functions. This section is of type **SHT_PROGBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.dynamic
This section holds dynamic linking information. The section's attributes will include the **SHF_ALLOC** bit. Whether the **SHF_WRITE** bit is set is processor-specific. This section is of type **SHT_DYNAMIC**. See the attributes above.

.dynstr This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. This section is of type **SHT_STRTAB**. The attribute type used is **SHF_ALLOC**.

.dynsym
This section holds the dynamic linking symbol table. This section is of type **SHT_DYNSYM**. The attribute used is **SHF_ALLOC**.

.fini This section holds executable instructions that contribute to the process termination code. When a program exits normally the system arranges to execute the code in this section. This section is of type **SHT_PROGBITS**. The attributes used are **SHF_ALLOC** and **SHF_EXECINSTR**.

.gnu.version
This section holds the version symbol table, an array of *ElfN_Half* elements. This section is of type **SHT_GNU_versym**. The attribute type used is **SHF_ALLOC**.

.gnu.version_d
This section holds the version symbol definitions, a table of *ElfN_Verdef* structures. This section is of type **SHT_GNU_verdef**. The attribute type used is **SHF_ALLOC**.

.gnu.version_r
This section holds the version symbol needed elements, a table of *ElfN_Verneed* structures. This section is of type **SHT_GNU_versym**. The attribute type used is **SHF_ALLOC**.

.got This section holds the global offset table. This section is of type **SHT_PROGBITS**. The attributes are processor-specific.

.hash This section holds a symbol hash table. This section is of type **SHT_HASH**. The attribute used is **SHF_ALLOC**.

.init This section holds executable instructions that contribute to the process initialization code. When a program starts to run the system arranges to execute the code in this section before calling the main program entry point. This section is of type **SHT_PROGBITS**. The attributes used are **SHF_ALLOC** and **SHF_EXECINSTR**.

.interp This section holds the pathname of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the **SHF_ALLOC** bit. Otherwise, that bit will be off. This section is of type **SHT_PROGBITS**.

.line This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code. The contents are unspecified. This section is of type **SHT_PROGBITS**. No attribute types are used.

.note This section holds various notes. This section is of type **SHT_NOTE**. No attribute types are used.

.note.ABI-tag
This section is used to declare the expected run-time ABI of the ELF image. It may include the operating system name and its run-time versions. This section is of type **SHT_NOTE**. The only attribute used is **SHF_ALLOC**.

.note.gnu.build-id
This section is used to hold an ID that uniquely identifies the contents of the ELF image. Different files with the same build ID should contain the same executable content. See the **--build-id** option to the GNU linker (**ld** (1)) for more details. This section is of type **SHT_NOTE**. The only attribute used is **SHF_ALLOC**.

.note.GNU-stack
This section is used in Linux object files for declaring stack attributes. This section is of type **SHT_PROGBITS**. The only attribute used is **SHF_EXECINSTR**. This indicates to the GNU linker that the object file requires an executable stack.

.note.openbsd.ident
OpenBSD native executables usually contain this section to identify themselves so the kernel can bypass any compatibility ELF binary emulation tests when loading the file.

.plt This section holds the procedure linkage table. This section is of type **SHT_PROGBITS**. The attributes are processor-specific.

.relNAME
This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the **SHF_ALLOC** bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rel.text**. This section is of type **SHT_REL**.

.relaNAME
This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the **SHF_ALLOC** bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rela.text**. This section is of type **SHT_RELA**.

.rodata This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type **SHT_PROGBITS**. The attribute used is **SHF_ALLOC**.

.rodata1
: This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type **SHT_PROGBITS**. The attribute used is **SHF_ALLOC**.

.shstrtab
: This section holds section names. This section is of type **SHT_STRTAB**. No attribute types are used.

.strtab
: This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the **SHF_ALLOC** bit. Otherwise, the bit will be off. This section is of type **SHT_STRTAB**.

.symtab
: This section holds a symbol table. If the file has a loadable segment that includes the symbol table, the section's attributes will include the **SHF_ALLOC** bit. Otherwise, the bit will be off. This section is of type **SHT_SYMTAB**.

.text
: This section holds the "text", or executable instructions, of a program. This section is of type **SHT_PROGBITS**. The attributes used are **SHF_ALLOC** and **SHF_EXECINSTR**.

String and symbol tables

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null byte ('\0'). Similarly, a string table's last byte is defined to hold a null byte, ensuring null termination for all strings.

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array.

```
typedef struct {
    uint32_t      st_name;
    Elf32_Addr    st_value;
    uint32_t      st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
} Elf32_Sym;

typedef struct {
    uint32_t      st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
    Elf64_Addr    st_value;
    uint64_t      st_size;
} Elf64_Sym;
```

The 32-bit and 64-bit versions have the same members, just in a different order.

st_name
: This member holds an index into the object file's symbol string table, which holds character representations of the symbol names. If the value is nonzero, it represents a string table index that gives the symbol name. Otherwise, the symbol has no name.

st_value
: This member gives the value of the associated symbol.

st_size
: Many symbols have associated sizes. This member holds zero if the symbol has no size or an unknown size.

st_info
: This member specifies the symbol's type and binding attributes:

  STT_NOTYPE
  : The symbol's type is not defined.

STT_OBJECT
The symbol is associated with a data object.

STT_FUNC
The symbol is associated with a function or other executable code.

STT_SECTION
The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have **STB_LOCAL** bindings.

STT_FILE
By convention, the symbol's name gives the name of the source file associated with the object file. A file symbol has **STB_LOCAL** bindings, its section index is **SHN_ABS**, and it precedes the other **STB_LOCAL** symbols of the file, if it is present.

STT_LOPROC
STT_HIPROC
Values in the inclusive range [**STT_LOPROC**, **STT_HIPROC**] are reserved for processor-specific semantics.

STB_LOCAL
Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

STB_GLOBAL
Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same symbol.

STB_WEAK
Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOPROC
STB_HIPROC
Values in the inclusive range [**STB_LOPROC**, **STB_HIPROC**] are reserved for processor-specific semantics.

There are macros for packing and unpacking the binding and type fields:

ELF32_ST_BIND(*info***)**
ELF64_ST_BIND(*info***)**
Extract a binding from an *st_info* value.

ELF32_ST_TYPE(*info***)**
ELF64_ST_TYPE(*info***)**
Extract a type from an *st_info* value.

ELF32_ST_INFO(*bind***, *** type***)**
ELF64_ST_INFO(*bind***, *** type***)**
Convert a binding and a type into an *st_info* value.

st_other
This member defines the symbol visibility.

STV_DEFAULT
Default symbol visibility rules. Global and weak symbols are available to other modules; references in the local module can be interposed by definitions in other modules.

STV_INTERNAL
Processor-specific hidden class.

STV_HIDDEN
Symbol is unavailable to other modules; references in the local module always resolve to the local symbol (i.e., the symbol can't be interposed by definitions in other modules).

STV_PROTECTED
Symbol is available to other modules, but references in the local module always resolve to the local symbol.

There are macros for extracting the visibility type:

ELF32_ST_VISIBILITY(other) or *ELF64_ST_VISIBILITY*(other)

st_shndx
Every symbol table entry is "defined" in relation to some section. This member holds the relevant section header table index.

Relocation entries (Rel & Rela)
Relocation is the process of connecting symbolic references with symbolic definitions. Relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

Relocation structures that do not need an addend:

```
typedef struct {
    Elf32_Addr r_offset;
    uint32_t   r_info;
} Elf32_Rel;

typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info;
} Elf64_Rel;
```

Relocation structures that need an addend:

```
typedef struct {
    Elf32_Addr r_offset;
    uint32_t   r_info;
    int32_t    r_addend;
} Elf32_Rela;

typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info;
    int64_t    r_addend;
} Elf64_Rela;
```

r_offset
This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or shared object, the value is the virtual address of the storage unit affected by the relocation.

r_info This member gives both the symbol table index with respect to which the relocation must be made and the type of relocation to apply. Relocation types are processor-specific. When the text refers to a relocation entry's relocation type or symbol table index, it means the result of applying **ELF[32|64]_R_TYPE** or **ELF[32|64]_R_SYM**, respectively, to the entry's *r_info* member.

r_addend
This member specifies a constant addend used to compute the value to be stored into the relocatable field.

Dynamic tags (Dyn)
The *.dynamic* section contains a series of structures that hold relevant dynamic linking information. The *d_tag* member controls the interpretation of *d_un*.

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word d_val;
```

```
        Elf32_Addr d_ptr;
    } d_un;
} Elf32_Dyn;
extern Elf32_Dyn _DYNAMIC[];

typedef struct {
    Elf64_Sxword    d_tag;
    union {
        Elf64_Xword d_val;
        Elf64_Addr  d_ptr;
    } d_un;
} Elf64_Dyn;
extern Elf64_Dyn _DYNAMIC[];
```

d_tag This member may have any of the following values:

DT_NULL Marks end of dynamic section

DT_NEEDED
String table offset to name of a needed library

DT_PLTRELSZ
Size in bytes of PLT relocation entries

DT_PLTGOT
Address of PLT and/or GOT

DT_HASH Address of symbol hash table

DT_STRTAB
Address of string table

DT_SYMTAB
Address of symbol table

DT_RELA Address of Rela relocation table

DT_RELASZ
Size in bytes of the Rela relocation table

DT_RELAENT
Size in bytes of a Rela relocation table entry

DT_STRSZ Size in bytes of string table

DT_SYMENT
Size in bytes of a symbol table entry

DT_INIT Address of the initialization function

DT_FINI Address of the termination function

DT_SONAME
String table offset to name of shared object

DT_RPATH String table offset to library search path (deprecated)

DT_SYMBOLIC
Alert linker to search this shared object before the executable for symbols

DT_REL Address of Rel relocation table

DT_RELSZ Size in bytes of Rel relocation table

DT_RELENT
Size in bytes of a Rel table entry

DT_PLTREL
Type of relocation entry to which the PLT refers (Rela or Rel)

DT_DEBUG Undefined use for debugging

DT_TEXTREL
Absence of this entry indicates that no relocation entries should apply to a non-writable segment

DT_JMPREL
Address of relocation entries associated solely with the PLT

DT_BIND_NOW
Instruct dynamic linker to process all relocations before transferring control to the executable

DT_RUNPATH
String table offset to library search path

DT_LOPROC
DT_HIPROC
Values in the inclusive range [**DT_LOPROC**, **DT_HIPROC**] are reserved for processor-specific semantics

d_val This member represents integer values with various interpretations.

d_ptr This member represents program virtual addresses. When interpreting these addresses, the actual address should be computed based on the original file value and memory base address. Files do not contain relocation entries to fixup these addresses.

_DYNAMIC
Array containing all the dynamic structures in the *.dynamic* section. This is automatically populated by the linker.

Notes (Nhdr)

ELF notes allow for appending arbitrary information for the system to use. They are largely used by core files (*e_type* of **ET_CORE**), but many projects define their own set of extensions. For example, the GNU tool chain uses ELF notes to pass information from the linker to the C library.

Note sections contain a series of notes (see the *struct* definitions below). Each note is followed by the name field (whose length is defined in *n_namesz*) and then by the descriptor field (whose length is defined in *n_descsz*) and whose starting address has a 4 byte alignment. Neither field is defined in the note struct due to their arbitrary lengths.

An example for parsing out two consecutive notes should clarify their layout in memory:

```
void *memory, *name, *desc;
Elf64_Nhdr *note, *next_note;

/* The buffer is pointing to the start of the section/segment. */
note = memory;

/* If the name is defined, it follows the note. */
name = note->n_namesz == 0 ? NULL : memory + sizeof(*note);

/* If the descriptor is defined, it follows the name
   (with alignment). */

desc = note->n_descsz == 0 ? NULL :
       memory + sizeof(*note) + ALIGN_UP(note->n_namesz, 4);

/* The next note follows both (with alignment). */
next_note = memory + sizeof(*note) +
                     ALIGN_UP(note->n_namesz, 4) +
                     ALIGN_UP(note->n_descsz, 4);
```

Keep in mind that the interpretation of *n_type* depends on the namespace defined by the *n_namesz* field. If the *n_namesz* field is not set (e.g., is 0), then there are two sets of notes: one for core files and one for all other ELF types. If the namespace is unknown, then tools will usually fallback to these sets of notes as well.

```
typedef struct {
    Elf32_Word n_namesz;
    Elf32_Word n_descsz;
    Elf32_Word n_type;
} Elf32_Nhdr;

typedef struct {
    Elf64_Word n_namesz;
    Elf64_Word n_descsz;
    Elf64_Word n_type;
} Elf64_Nhdr;
```

n_namesz
: The length of the name field in bytes. The contents will immediately follow this note in memory. The name is null terminated. For example, if the name is "GNU", then *n_namesz* will be set to 4.

n_descsz
: The length of the descriptor field in bytes. The contents will immediately follow the name field in memory.

n_type Depending on the value of the name field, this member may have any of the following values:

Core files (e_type = ET_CORE)
: Notes used by all core files. These are highly operating system or architecture specific and often require close coordination with kernels, C libraries, and debuggers. These are used when the namespace is the default (i.e., *n_namesz* will be set to 0), or a fallback when the namespace is unknown.

NT_PRSTATUS	prstatus struct
NT_FPREGSET	fpregset struct
NT_PRPSINFO	prpsinfo struct
NT_PRXREG	prxregset struct
NT_TASKSTRUCT	task structure
NT_PLATFORM	String from sysinfo(SI_PLATFORM)
NT_AUXV	auxv array
NT_GWINDOWS	gwindows struct
NT_ASRS	asrset struct
NT_PSTATUS	pstatus struct
NT_PSINFO	psinfo struct
NT_PRCRED	prcred struct
NT_UTSNAME	utsname struct
NT_LWPSTATUS	lwpstatus struct
NT_LWPSINFO	lwpinfo struct
NT_PRFPXREG	fprxregset struct
NT_SIGINFO	siginfo_t (size might increase over time)
NT_FILE	Contains information about mapped files
NT_PRXFPREG	user_fxsr_struct
NT_PPC_VMX	PowerPC Altivec/VMX registers
NT_PPC_SPE	PowerPC SPE/EVR registers
NT_PPC_VSX	PowerPC VSX registers
NT_386_TLS	i386 TLS slots (struct user_desc)
NT_386_IOPERM	x86 io permission bitmap (1=deny)
NT_X86_XSTATE	x86 extended state using xsave
NT_S390_HIGH_GPRS	s390 upper register halves
NT_S390_TIMER	s390 timer register
NT_S390_TODCMP	s390 time-of-day (TOD) clock comparator register
NT_S390_TODPREG	s390 time-of-day (TOD) programmable register
NT_S390_CTRS	s390 control registers

NT_S390_PREFIX s390 prefix register
NT_S390_LAST_BREAK
s390 breaking event address
NT_S390_SYSTEM_CALL
s390 system call restart data
NT_S390_TDB s390 transaction diagnostic block
NT_ARM_VFP ARM VFP/NEON registers
NT_ARM_TLS ARM TLS register
NT_ARM_HW_BREAK
ARM hardware breakpoint registers
NT_ARM_HW_WATCH
ARM hardware watchpoint registers
NT_ARM_SYSTEM_CALL
ARM system call number

n_name = GNU
Extensions used by the GNU tool chain.

NT_GNU_ABI_TAG
Operating system (OS) ABI information. The desc field will be 4 words:

[0] OS descriptor (**ELF_NOTE_OS_LINUX**, **ELF_NOTE_OS_GNU**, and so on)‘
[1] major version of the ABI
[2] minor version of the ABI
[3] subminor version of the ABI

NT_GNU_HWCAP
Synthetic hwcap information. The desc field begins with two words:

[0] number of entries
[1] bit mask of enabled entries

Then follow variable-length entries, one byte followed by a null-terminated hwcap name string. The byte gives the bit number to test if enabled, (1U << bit) & bit mask.

NT_GNU_BUILD_ID
Unique build ID as generated by the GNU *ld*(1) **−−build−id** option. The desc consists of any nonzero number of bytes.

NT_GNU_GOLD_VERSION
The desc contains the GNU Gold linker version used.

Default/unknown namespace (e_type != ET_CORE)
These are used when the namespace is the default (i.e., *n_namesz* will be set to 0), or a fallback when the namespace is unknown.

NT_VERSION
A version string of some sort.
NT_ARCH Architecture information.

NOTES

ELF first appeared in System V. The ELF format is an adopted standard.

The extensions for *e_phnum*, *e_shnum*, and *e_shstrndx* respectively are Linux extensions. Sun, BSD, and AMD64 also support them; for further information, look under SEE ALSO.

SEE ALSO

as(1), *elfedit*(1), *gdb*(1), *ld*(1), *nm*(1), *objcopy*(1), *objdump*(1), *patchelf*(1), *readelf*(1), *size*(1), *strings*(1), *strip*(1), *execve*(2), *dl_iterate_phdr*(3), *core*(5), *ld.so*(8)

Hewlett-Packard, *Elf-64 Object File Format*.

Santa Cruz Operation, *System V Application Binary Interface*.

UNIX System Laboratories, "Object Files", *Executable and Linking Format (ELF)*.

Sun Microsystems, *Linker and Libraries Guide*.

AMD64 ABI Draft, *System V Application Binary Interface AMD64 Architecture Processor Supplement*.

NAME

erofs − the Enhanced Read-Only File System

DESCRIPTION

erofs is a create-once read-only filesystem, with support for compression and a multi-device backing store.

There are two inode formats:

- 32-byte compact with 16-bit UID/GID, 32-bit file size, and no file times
- 64-byte extended with 32-bit UID/GID, 64-bit file size, and a modification time (*st_mtim*).

Mount options

user_xattr
nouser_xattr
: Controls whether *user* extended attributes are exposed. Defaults to yes.

acl
noacl
: Controls whether POSIX *acl*(5)s are exposed. Defaults to yes.

cache_strategy=**disabled**|**readahead**|**readaround**
: Cache allocation for compressed files: never, if reading from start of file, regardless of position. Defaults to **readaround**.

dax
dax=**always**|**never**
: Direct Access control. If **always** and the source device supports DAX, uncompressed non-inlined files will be read directly, without going through the page cache. **dax** is a synonym for **always**. Defaults to unset, which is equivalent to **never**.

device=*blobdev*
: Add extra device holding some of the data. Must be given as many times and in the same order as **−−blobdev** was to *mkfs.erofs*(1)

domain_id=*did*
fsid=*id*
: Control CacheFiles on-demand read support. To be documented.

VERSIONS

erofs images are versioned through the use of feature flags; these are listed in the **−E** section of *mkfs.erofs*(1),

CONFIGURATION

Linux must be configured with the **CONFIG_EROFS_FS** option to mount EROFS filesystems. There are sub-configuration items that restrict the availability of some of the parameters above.

SEE ALSO

mkfs.erofs(1), *fsck.erofs*(1), *dump.erofs*(1)

Documentation/filesystems/erofs.txt in the Linux source.

NAME

filesystems − Linux filesystem types: ext, ext2, ext3, ext4, hpfs, iso9660, JFS, minix, msdos, ncpfs nfs, ntfs, proc, Reiserfs, smb, sysv, umsdos, vfat, XFS, xiafs

DESCRIPTION

When, as is customary, the **proc** filesystem is mounted on */proc*, you can find in the file */proc/filesystems* which filesystems your kernel currently supports; see *proc*(5) for more details. There is also a legacy *sysfs*(2) system call (whose availability is controlled by the **CONFIG_SYSFS_SYSCALL** kernel build configuration option since Linux 3.15) that enables enumeration of the currently available filesystem types regardless of */proc* availability and/or sanity.

If you need a currently unsupported filesystem, insert the corresponding kernel module or recompile the kernel.

In order to use a filesystem, you have to *mount* it; see *mount*(2) and *mount*(8)

The following list provides a short description of the available or historically available filesystems in the Linux kernel. See the kernel documentation for a comprehensive description of all options and limitations.

erofs is the Enhanced Read-Only File System, stable since Linux 5.4. See *erofs*(5).

ext is an elaborate extension of the **minix** filesystem. It has been completely superseded by the second version of the extended filesystem (**ext2**) and has been removed from the kernel (in Linux 2.1.21).

ext2 is a disk filesystem that was used by Linux for fixed disks as well as removable media. The second extended filesystem was designed as an extension of the extended filesystem (**ext**). See *ext2*(5)

ext3 is a journaling version of the **ext2** filesystem. It is easy to switch back and forth between **ext2** and **ext3**. See *ext3*(5)

ext4 is a set of upgrades to **ext3** including substantial performance and reliability enhancements, plus large increases in volume, file, and directory size limits. See *ext4*(5)

hpfs is the High Performance Filesystem, used in OS/2. This filesystem is read-only under Linux due to the lack of available documentation.

iso9660

is a CD-ROM filesystem type conforming to the ISO/IEC 9660 standard.

High Sierra

Linux supports High Sierra, the precursor to the ISO/IEC 9660 standard for CD-ROM filesystems. It is automatically recognized within the **iso9660** filesystem support under Linux.

Rock Ridge

Linux also supports the System Use Sharing Protocol records specified by the Rock Ridge Interchange Protocol. They are used to further describe the files in the **iso9660** filesystem to a UNIX host, and provide information such as long filenames, UID/GID, POSIX permissions, and devices. It is automatically recognized within the **iso9660** filesystem support under Linux.

JFS is a journaling filesystem, developed by IBM, that was integrated into Linux 2.4.24.

minix is the filesystem used in the Minix operating system, the first to run under Linux. It has a number of shortcomings, including a 64 MB partition size limit, short filenames, and a single timestamp. It remains useful for floppies and RAM disks.

msdos is the filesystem used by DOS, Windows, and some OS/2 computers. **msdos** filenames can be no longer than 8 characters, followed by an optional period and 3 character extension.

ncpfs is a network filesystem that supports the NCP protocol, used by Novell NetWare. It was removed from the kernel in Linux 4.17.

To use **ncpfs**, you need special programs, which can be found at .

nfs is the network filesystem used to access disks located on remote computers.

ntfs is the filesystem native to Microsoft Windows NT, supporting features like ACLs, journaling, encryption, and so on.

proc is a pseudo filesystem which is used as an interface to kernel data structures rather than reading and interpreting */dev/kmem*. In particular, its files do not take disk space. See *proc*(5).

Reiserfs
is a journaling filesystem, designed by Hans Reiser, that was integrated into Linux 2.4.1.

smb is a network filesystem that supports the SMB protocol, used by Windows. See .

sysv is an implementation of the System V/Coherent filesystem for Linux. It implements all of Xenix FS, System V/386 FS, and Coherent FS.

umsdos
is an extended DOS filesystem used by Linux. It adds capability for long filenames, UID/GID, POSIX permissions, and special files (devices, named pipes, etc.) under the DOS filesystem, without sacrificing compatibility with DOS.

tmpfs is a filesystem whose contents reside in virtual memory. Since the files on such filesystems typically reside in RAM, file access is extremely fast. See *tmpfs*(5).

vfat is an extended FAT filesystem used by Microsoft Windows95 and Windows NT. **vfat** adds the capability to use long filenames under the MSDOS filesystem.

XFS is a journaling filesystem, developed by SGI, that was integrated into Linux 2.4.20.

xiafs was designed and implemented to be a stable, safe filesystem by extending the Minix filesystem code. It provides the basic most requested features without undue complexity. The **xiafs** filesystem is no longer actively developed or maintained. It was removed from the kernel in Linux 2.1.21.

SEE ALSO

fuse(4), *btrfs*(5), *ext2*(5), *ext3*(5), *ext4*(5), *nfs*(5), *proc*(5), *sysfs*(5), *tmpfs*(5), *xfs*(5), *fsck*(8), *mkfs*(8), *mount*(8)

NAME

ftpusers − list of users that may not log in via the FTP daemon

DESCRIPTION

The text file **ftpusers** contains a list of users that may not log in using the File Transfer Protocol (FTP) server daemon. This file is used not merely for system administration purposes but also for improving security within a TCP/IP networked environment.

The **ftpusers** file will typically contain a list of the users that either have no business using ftp or have too many privileges to be allowed to log in through the FTP server daemon. Such users usually include root, daemon, bin, uucp, and news.

If your FTP server daemon doesn't use **ftpusers**, then it is suggested that you read its documentation to find out how to block access for certain users. Washington University FTP server Daemon (wuftpd) and Professional FTP Daemon (proftpd) are known to make use of **ftpusers**.

Format

The format of **ftpusers** is very simple. There is one account name (or username) per line. Lines starting with a # are ignored.

FILES

/etc/ftpusers

SEE ALSO

passwd(5), *proftpd*(8), *wuftpd*(8)

NAME

gai.conf − getaddrinfo(3) configuration file

DESCRIPTION

A call to *getaddrinfo*(3) might return multiple answers. According to RFC 3484 these answers must be sorted so that the answer with the highest success rate is first in the list. The RFC provides an algorithm for the sorting. The static rules are not always adequate, though. For this reason, the RFC also requires that system administrators should have the possibility to dynamically change the sorting. For the glibc implementation, this can be achieved with the */etc/gai.conf* file.

Each line in the configuration file consists of a keyword and its parameters. White spaces in any place are ignored. Lines starting with '#' are comments and are ignored.

The keywords currently recognized are:

label *netmask precedence*
: The value is added to the label table used in the RFC 3484 sorting. If any **label** definition is present in the configuration file, the default table is not used. All the label definitions of the default table which are to be maintained have to be duplicated. Following the keyword, the line has to contain a network mask and a precedence value.

precedence *netmask precedence*
: This keyword is similar to **label**, but instead the value is added to the precedence table as specified in RFC 3484. Once again, the presence of a single **precedence** line in the configuration file causes the default table to not be used.

reload <**yes**|**no**>
: This keyword controls whether a process checks whether the configuration file has been changed since the last time it was read. If the value is "**yes**", the file is reread. This might cause problems in multithreaded applications and is generally a bad idea. The default is "**no**".

scopev4 *mask value*
: Add another rule to the RFC 3484 scope table for IPv4 address. By default, the scope IDs described in section 3.2 in RFC 3438 are used. Changing these defaults should hardly ever be necessary.

FILES

/etc/gai.conf

VERSIONS

The *gai.conf* file is supported since glibc 2.5.

EXAMPLES

The default table according to RFC 3484 would be specified with the following configuration file:

```
label  ::1/128       0
label  ::/0          1
label  2002::/16     2
label ::/96          3
label ::ffff:0:0/96  4
precedence  ::1/128       50
precedence  ::/0          40
precedence  2002::/16     30
precedence ::/96          20
precedence ::ffff:0:0/96  10
```

SEE ALSO

getaddrinfo(3), RFC 3484

NAME

group − user group file

DESCRIPTION

The */etc/group* file is a text file that defines the groups on the system. There is one entry per line, with the following format:

```
group_name:password:GID:user_list
```

The fields are as follows:

group_name
: the name of the group.

password
: the (encrypted) group password. If this field is empty, no password is needed.

GID
: the numeric group ID.

user_list
: a list of the usernames that are members of this group, separated by commas.

FILES

/etc/group

BUGS

As the 4.2BSD *initgroups*(3) man page says: no one seems to keep */etc/group* up-to-date.

SEE ALSO

chgrp(1), *gpasswd*(1), *groups*(1), *login*(1), *newgrp*(1), *sg*(1), *getgrent*(3), *getgrnam*(3), *gshadow*(5), *passwd*(5), *vigr*(8)

NAME

host.conf − resolver configuration file

DESCRIPTION

The file */etc/host.conf* contains configuration information specific to the resolver library. It should contain one configuration keyword per line, followed by appropriate configuration information. The following keywords are recognized:

trim This keyword may be listed more than once. Each time it should be followed by a list of domains, separated by colons (':'), semicolons (';') or commas (','), with the leading dot. When set, the resolver library will automatically trim the given domain name from the end of any hostname resolved via DNS. This is intended for use with local hosts and domains. (Related note: *trim* will not affect hostnames gathered via NIS or the *hosts*(5) file. Care should be taken to ensure that the first hostname for each entry in the hosts file is fully qualified or unqualified, as appropriate for the local installation.)

multi Valid values are *on* and *off*. If set to *on*, the resolver library will return all valid addresses for a host that appears in the */etc/hosts* file, instead of only the first. This is *off* by default, as it may cause a substantial performance loss at sites with large hosts files.

reorder Valid values are *on* and *off*. If set to *on*, the resolver library will attempt to reorder host addresses so that local addresses (i.e., on the same subnet) are listed first when a *gethostbyname*(3) is performed. Reordering is done for all lookup methods. The default value is *off*.

ENVIRONMENT

The following environment variables can be used to allow users to override the behavior which is configured in */etc/host.conf*:

RESOLV_HOST_CONF
If set, this variable points to a file that should be read instead of */etc/host.conf*.

RESOLV_MULTI
Overrides the *multi* command.

RESOLV_REORDER
Overrides the *reorder* command.

RESOLV_ADD_TRIM_DOMAINS
A list of domains, separated by colons (':'), semicolons (';'), or commas (','), with the leading dot, which will be added to the list of domains that should be trimmed.

RESOLV_OVERRIDE_TRIM_DOMAINS
A list of domains, separated by colons (':'), semicolons (';'), or commas (','), with the leading dot, which will replace the list of domains that should be trimmed. Overrides the *trim* command.

FILES

/etc/host.conf
Resolver configuration file

/etc/resolv.conf
Resolver configuration file

/etc/hosts
Local hosts database

NOTES

The following differences exist compared to the original implementation. A new command *spoof* and a new environment variable **RESOLV_SPOOF_CHECK** can take arguments like *off*, *nowarn*, and *warn*. Line comments can appear anywhere and not only at the beginning of a line.

Historical

The *nsswitch.conf*(5) file is the modern way of controlling the order of host lookups.

In glibc 2.4 and earlier, the following keyword is recognized:

order This keyword specifies how host lookups are to be performed. It should be followed by one or more lookup methods, separated by commas. Valid methods are *bind*, *hosts*, and *nis*.

RESOLV_SERV_ORDER
Overrides the *order* command.

Since glibc 2.0.7, and up through glibc 2.24, the following keywords and environment variable have been recognized but never implemented:

nospoof
Valid values are *on* and *off*. If set to *on*, the resolver library will attempt to prevent hostname spoofing to enhance the security of **rlogin** and **rsh**. It works as follows: after performing a host address lookup, the resolver library will perform a hostname lookup for that address. If the two hostnames do not match, the query fails. The default value is *off*.

spoofalert
Valid values are *on* and *off*. If this option is set to *on* and the *nospoof* option is also set, the resolver library will log a warning of the error via the syslog facility. The default value is *off*.

spoof Valid values are *off*, *nowarn*, and *warn*. If this option is set to *off*, spoofed addresses are permitted and no warnings will be emitted via the syslog facility. If this option is set to *warn*, the resolver library will attempt to prevent hostname spoofing to enhance the security and log a warning of the error via the syslog facility. If this option is set to *nowarn*, the resolver library will attempt to prevent hostname spoofing to enhance the security but not emit warnings via the syslog facility. Setting this option to anything else is equal to setting it to *nowarn*.

RESOLV_SPOOF_CHECK
Overrides the *nospoof*, *spoofalert*, and *spoof* commands in the same way as the *spoof* command is parsed. Valid values are *off*, *nowarn*, and *warn*.

SEE ALSO
gethostbyname(3), *hosts*(5), *nsswitch.conf*(5), *resolv.conf*(5), *hostname*(7), *named*(8)

NAME

hosts − static table lookup for hostnames

SYNOPSIS

/etc/hosts

DESCRIPTION

This manual page describes the format of the */etc/hosts* file. This file is a simple text file that associates IP addresses with hostnames, one line per IP address. For each host a single line should be present with the following information:

IP_address canonical_hostname [aliases...]

The IP address can conform to either IPv4 or IPv6. Fields of the entry are separated by any number of blanks and/or tab characters. Text from a "#" character until the end of the line is a comment, and is ignored. Host names may contain only alphanumeric characters, minus signs ("−"), and periods (".."). They must begin with an alphabetic character and end with an alphanumeric character. Optional aliases provide for name changes, alternate spellings, shorter hostnames, or generic hostnames (for example, *localhost*). If required, a host may have two separate entries in this file; one for each version of the Internet Protocol (IPv4 and IPv6).

The Berkeley Internet Name Domain (BIND) Server implements the Internet name server for UNIX systems. It augments or replaces the */etc/hosts* file or hostname lookup, and frees a host from relying on */etc/hosts* being up to date and complete.

In modern systems, even though the host table has been superseded by DNS, it is still widely used for:

bootstrapping
: Most systems have a small host table containing the name and address information for important hosts on the local network. This is useful when DNS is not running, for example during system bootup.

NIS
: Sites that use NIS use the host table as input to the NIS host database. Even though NIS can be used with DNS, most NIS sites still use the host table with an entry for all local hosts as a backup.

isolated nodes
: Very small sites that are isolated from the network use the host table instead of DNS. If the local information rarely changes, and the network is not connected to the Internet, DNS offers little advantage.

FILES

/etc/hosts

NOTES

Modifications to this file normally take effect immediately, except in cases where the file is cached by applications.

Historical notes

RFC 952 gave the original format for the host table, though it has since changed.

Before the advent of DNS, the host table was the only way of resolving hostnames on the fledgling Internet. Indeed, this file could be created from the official host data base maintained at the Network Information Control Center (NIC), though local changes were often required to bring it up to date regarding unofficial aliases and/or unknown hosts. The NIC no longer maintains the hosts.txt files, though looking around at the time of writing (circa 2000), there are historical hosts.txt files on the WWW. I just found three, from 92, 94, and 95.

EXAMPLES

```
# The following lines are desirable for IPv4 capable hosts
127.0.0.1       localhost

# 127.0.1.1 is often used for the FQDN of the machine
127.0.1.1       thishost.example.org   thishost
192.168.1.10    foo.example.org        foo
192.168.1.13    bar.example.org        bar
146.82.138.7    master.debian.org      master
```

```
209.237.226.90  www.opensource.org

# The following lines are desirable for IPv6 capable hosts
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
```

SEE ALSO

hostname(1), *resolver*(3), *host.conf*(5), *resolv.conf*(5), *resolver*(5), *hostname*(7), *named*(8)

Internet RFC 952

NAME

hosts.equiv − list of hosts and users that are granted "trusted" **r** command access to your system

DESCRIPTION

The file */etc/hosts.equiv* allows or denies hosts and users to use the **r**-commands (e.g., **rlogin**, **rsh**, or **rcp**) without supplying a password.

The file uses the following format:

+|[−]hostname|+@netgroup|−@netgroup [+|[−]username|+@netgroup|−@netgroup]

The *hostname* is the name of a host which is logically equivalent to the local host. Users logged into that host are allowed to access like-named user accounts on the local host without supplying a password. The *hostname* may be (optionally) preceded by a plus (+) sign. If the plus sign is used alone, it allows any host to access your system. You can explicitly deny access to a host by preceding the *hostname* by a minus (−) sign. Users from that host must always supply additional credentials, including possibly a password. For security reasons you should always use the FQDN of the hostname and not the short hostname.

The *username* entry grants a specific user access to all user accounts (except root) without supplying a password. That means the user is NOT restricted to like-named accounts. The *username* may be (optionally) preceded by a plus (+) sign. You can also explicitly deny access to a specific user by preceding the *username* with a minus (−) sign. This says that the user is not trusted no matter what other entries for that host exist.

Netgroups can be specified by preceding the netgroup by an @ sign.

Be extremely careful when using the plus (+) sign. A simple typographical error could result in a standalone plus sign. A standalone plus sign is a wildcard character that means "any host"!

FILES

/etc/hosts.equiv

NOTES

Some systems will honor the contents of this file only when it has owner root and no write permission for anybody else. Some exceptionally paranoid systems even require that there be no other hard links to the file.

Modern systems use the Pluggable Authentication Modules library (PAM). With PAM a standalone plus sign is considered a wildcard character which means "any host" only when the word *promiscuous* is added to the auth component line in your PAM file for the particular service (e.g., **rlogin**).

EXAMPLES

Below are some example */etc/host.equiv* or *~/.rhosts* files.

Allow any user to log in from any host:

```
+
```

Allow any user from *host* with a matching local account to log in:

```
host
```

Note: the use of *+host* is never a valid syntax, including attempting to specify that any user from the host is allowed.

Allow any user from *host* to log in:

```
host +
```

Note: this is distinct from the previous example since it does not require a matching local account.

Allow *user* from *host* to log in as any non-root user:

```
host user
```

Allow all users with matching local accounts from *host* to log in except for *baduser*:

```
host −baduser
host
```

Deny all users from *host*:

```
−host
```

Note: the use of −*host* −*user* is never a valid syntax, including attempting to specify that a particular user from the host is not trusted.

Allow all users with matching local accounts on all hosts in a *netgroup*:

```
+@netgroup
```

Disallow all users on all hosts in a *netgroup*:

```
−@netgroup
```

Allow all users in a *netgroup* to log in from *host* as any non-root user:

```
host +@netgroup
```

Allow all users with matching local accounts on all hosts in a *netgroup* except *baduser*:

```
+@netgroup −baduser
+@netgroup
```

Note: the deny statements must always precede the allow statements because the file is processed sequentially until the first matching rule is found.

SEE ALSO

rhosts(5), *rlogind*(8), *rshd*(8)

NAME

issue − prelogin message and identification file

DESCRIPTION

/etc/issue is a text file which contains a message or system identification to be printed before the login prompt. It may contain various **@***char* and *char* sequences, if supported by the **getty**-type program employed on the system.

FILES

/etc/issue

SEE ALSO

motd(5), *agetty*(8), *mingetty*(8)

NAME

locale − describes a locale definition file

DESCRIPTION

The **locale** definition file contains all the information that the *localedef*(1) command needs to convert it into the binary locale database.

The definition files consist of sections which each describe a locale category in detail. See *locale*(7) for additional details for these categories.

Syntax

The locale definition file starts with a header that may consist of the following keywords:

escape_char
: is followed by a character that should be used as the escape-character for the rest of the file to mark characters that should be interpreted in a special way. It defaults to the backslash (\).

comment_char
: is followed by a character that will be used as the comment-character for the rest of the file. It defaults to the number sign (#).

The locale definition has one part for each locale category. Each part can be copied from another existing locale or can be defined from scratch. If the category should be copied, the only valid keyword in the definition is *copy* followed by the name of the locale in double quotes which should be copied. The exceptions for this rule are **LC_COLLATE** and **LC_CTYPE** where a *copy* statement can be followed by locale-specific rules and selected overrides.

When defining a locale or a category from scratch, an existing system- provided locale definition file should be used as a reference to follow common glibc conventions.

Locale category sections

The following category sections are defined by POSIX:

- **LC_CTYPE**
- **LC_COLLATE**
- **LC_MESSAGES**
- **LC_MONETARY**
- **LC_NUMERIC**
- **LC_TIME**

In addition, since glibc 2.2, the GNU C library supports the following nonstandard categories:

- **LC_ADDRESS**
- **LC_IDENTIFICATION**
- **LC_MEASUREMENT**
- **LC_NAME**
- **LC_PAPER**
- **LC_TELEPHONE**

See *locale*(7) for a more detailed description of each category.

LC_ADDRESS

The definition starts with the string *LC_ADDRESS* in the first column.

The following keywords are allowed:

postal_fmt
: followed by a string containing field descriptors that define the format used for postal addresses in the locale. The following field descriptors are recognized:

  %n Person's name, possibly constructed with the **LC_NAME** *name_fmt* keyword (since glibc 2.24).

%a Care of person, or organization.

%f Firm name.

%d Department name.

%b Building name.

%s Street or block (e.g., Japanese) name.

%h House number or designation.

%N
Insert an end-of-line if the previous descriptor's value was not an empty string; otherwise ignore.

%t Insert a space if the previous descriptor's value was not an empty string; otherwise ignore.

%r Room number, door designation.

%e Floor number.

%C Country designation, from the *country_post* keyword.

%l Local township within town or city (since glibc 2.24).

%z Zip number, postal code.

%T Town, city.

%S State, province, or prefecture.

%c Country, as taken from data record.

Each field descriptor may have an 'R' after the '%' to specify that the information is taken from a Romanized version string of the entity.

country_name
followed by the country name in the language of the current document (e.g., "Deutschland" for the **de_DE** locale).

country_post
followed by the abbreviation of the country (see CERT_MAILCODES).

country_ab2
followed by the two-letter abbreviation of the country (ISO 3166).

country_ab3
followed by the three-letter abbreviation of the country (ISO 3166).

country_num
followed by the numeric country code (ISO 3166).

country_car
followed by the international license plate country code.

country_isbn
followed by the ISBN code (for books).

lang_name
followed by the language name in the language of the current document.

lang_ab
followed by the two-letter abbreviation of the language (ISO 639).

lang_term
followed by the three-letter abbreviation of the language (ISO 639-2/T).

lang_lib
followed by the three-letter abbreviation of the language for library use (ISO 639-2/B). Applications should in general prefer *lang_term* over *lang_lib*.

The **LC_ADDRESS** definition ends with the string *END LC_ADDRESS*.

LC_CTYPE

The definition starts with the string *LC_CTYPE* in the first column.

The following keywords are allowed:

upper followed by a list of uppercase letters. The letters **A** through **Z** are included automatically. Characters also specified as **cntrl**, **digit**, **punct**, or **space** are not allowed.

lower followed by a list of lowercase letters. The letters **a** through **z** are included automatically. Characters also specified as **cntrl**, **digit**, **punct**, or **space** are not allowed.

alpha followed by a list of letters. All character specified as either **upper** or **lower** are automatically included. Characters also specified as **cntrl**, **digit**, **punct**, or **space** are not allowed.

digit followed by the characters classified as numeric digits. Only the digits **0** through **9** are allowed. They are included by default in this class.

space followed by a list of characters defined as white-space characters. Characters also specified as **upper**, **lower**, **alpha**, **digit**, **graph**, or **xdigit** are not allowed. The characters **<space>**, **<form-feed>**, **<newline>**, **<carriage-return>**, **<tab>**, and **<vertical-tab>** are automatically included.

cntrl followed by a list of control characters. Characters also specified as **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print**, or **xdigit** are not allowed.

punct followed by a list of punctuation characters. Characters also specified as **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit**, or the **<space>** character are not allowed.

graph followed by a list of printable characters, not including the **<space>** character. The characters defined as **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** are automatically included. Characters also specified as **cntrl** are not allowed.

print followed by a list of printable characters, including the **<space>** character. The characters defined as **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct**, and the **<space>** character are automatically included. Characters also specified as **cntrl** are not allowed.

xdigit followed by a list of characters classified as hexadecimal digits. The decimal digits must be included followed by one or more set of six characters in ascending order. The following characters are included by default: **0** through **9**, **a** through **f**, **A** through **F**.

blank followed by a list of characters classified as **blank**. The characters **<space>** and **<tab>** are automatically included.

charclass
followed by a list of locale-specific character class names which are then to be defined in the locale.

toupper
followed by a list of mappings from lowercase to uppercase letters. Each mapping is a pair of a lowercase and an uppercase letter separated with a **,** and enclosed in parentheses.

tolower
followed by a list of mappings from uppercase to lowercase letters. If the keyword tolower is not present, the reverse of the toupper list is used.

map totitle
followed by a list of mapping pairs of characters and letters to be used in titles (headings).

class followed by a locale-specific character class definition, starting with the class name followed by the characters belonging to the class.

charconv
followed by a list of locale-specific character mapping names which are then to be defined in the locale.

outdigit
followed by a list of alternate output digits for the locale.

map to_inpunct
followed by a list of mapping pairs of alternate digits and separators for input digits for the locale.

map to_outpunct
followed by a list of mapping pairs of alternate separators for output for the locale.

translit_start
marks the start of the transliteration rules section. The section can contain the *include* keyword in the beginning followed by locale-specific rules and overrides. Any rule specified in the locale file will override any rule copied or included from other files. In case of duplicate rule definitions in the locale file, only the first rule is used.

A transliteration rule consist of a character to be transliterated followed by a list of transliteration targets separated by semicolons. The first target which can be presented in the target character set is used, if none of them can be used the *default_missing* character will be used instead.

include in the transliteration rules section includes a transliteration rule file (and optionally a repertoire map file).

default_missing
in the transliteration rules section defines the default character to be used for transliteration where none of the targets cannot be presented in the target character set.

translit_end
marks the end of the transliteration rules.

The **LC_CTYPE** definition ends with the string *END LC_CTYPE*.

LC_COLLATE

Note that glibc does not support all POSIX-defined options, only the options described below are supported (as of glibc 2.23).

The definition starts with the string *LC_COLLATE* in the first column.

The following keywords are allowed:

coll_weight_max
followed by the number representing used collation levels. This keyword is recognized but ignored by glibc.

collating−element
followed by the definition of a collating-element symbol representing a multicharacter collating element.

collating−symbol
followed by the definition of a collating symbol that can be used in collation order statements.

define followed by **string** to be evaluated in an *ifdef* **string** / *else* / *endif* construct.

reorder−after
followed by a redefinition of a collation rule.

reorder−end
marks the end of the redefinition of a collation rule.

reorder−sections−after
followed by a script name to reorder listed scripts after.

reorder−sections−end
marks the end of the reordering of sections.

script followed by a declaration of a script.

symbol−equivalence
followed by a collating-symbol to be equivalent to another defined collating-symbol.

The collation rule definition starts with a line:

order_start
followed by a list of keywords chosen from **forward**, **backward**, or **position**. The order definition consists of lines that describe the collation order and is terminated with the keyword *order_end*.

The **LC_COLLATE** definition ends with the string *END LC_COLLATE*.

LC_IDENTIFICATION

The definition starts with the string *LC_IDENTIFICATION* in the first column.

The following keywords are allowed:

title followed by the title of the locale document (e.g., "Maori language locale for New Zealand").

source followed by the name of the organization that maintains this document.

address
followed by the address of the organization that maintains this document.

contact followed by the name of the contact person at the organization that maintains this document.

email followed by the email address of the person or organization that maintains this document.

tel followed by the telephone number (in international format) of the organization that maintains this document. As of glibc 2.24, this keyword is deprecated in favor of other contact methods.

fax followed by the fax number (in international format) of the organization that maintains this document. As of glibc 2.24, this keyword is deprecated in favor of other contact methods.

language
followed by the name of the language to which this document applies.

territory
followed by the name of the country/geographic extent to which this document applies.

audience
followed by a description of the audience for which this document is intended.

application
followed by a description of any special application for which this document is intended.

abbreviation
followed by the short name for provider of the source of this document.

revision
followed by the revision number of this document.

date followed by the revision date of this document.

In addition, for each of the categories defined by the document, there should be a line starting with the keyword *category*, followed by:

(1) a string that identifies this locale category definition,

(2) a semicolon, and

(3) one of the **LC_*** identifiers.

The **LC_IDENTIFICATION** definition ends with the string *END LC_IDENTIFICATION*.

LC_MESSAGES

The definition starts with the string *LC_MESSAGES* in the first column.

The following keywords are allowed:

yesexpr
followed by a regular expression that describes possible yes-responses.

noexpr followed by a regular expression that describes possible no-responses.

yesstr followed by the output string corresponding to "yes".

nostr followed by the output string corresponding to "no".

The **LC_MESSAGES** definition ends with the string *END LC_MESSAGES*.

LC_MEASUREMENT

The definition starts with the string *LC_MEASUREMENT* in the first column.

The following keywords are allowed:

measurement
followed by number identifying the standard used for measurement. The following values are recognized:

1 Metric.

2 US customary measurements.

The **LC_MEASUREMENT** definition ends with the string *END LC_MEASUREMENT*.

LC_MONETARY

The definition starts with the string *LC_MONETARY* in the first column.

The following keywords are allowed:

int_curr_symbol
: followed by the international currency symbol. This must be a 4-character string containing the international currency symbol as defined by the ISO 4217 standard (three characters) followed by a separator.

currency_symbol
: followed by the local currency symbol.

mon_decimal_point
: followed by the single-character string that will be used as the decimal delimiter when formatting monetary quantities.

mon_thousands_sep
: followed by the single-character string that will be used as a group separator when formatting monetary quantities.

mon_grouping
: followed by a sequence of integers separated by semicolons that describe the formatting of monetary quantities. See *grouping* below for details.

positive_sign
: followed by a string that is used to indicate a positive sign for monetary quantities.

negative_sign
: followed by a string that is used to indicate a negative sign for monetary quantities.

int_frac_digits
: followed by the number of fractional digits that should be used when formatting with the *int_curr_symbol*.

frac_digits
: followed by the number of fractional digits that should be used when formatting with the *currency_symbol*.

p_cs_precedes
: followed by an integer that indicates the placement of *currency_symbol* for a nonnegative formatted monetary quantity:

  0 the symbol succeeds the value.

  1 the symbol precedes the value.

p_sep_by_space
: followed by an integer that indicates the separation of *currency_symbol*, the sign string, and the value for a nonnegative formatted monetary quantity. The following values are recognized:

  0 No space separates the currency symbol and the value.

  1 If the currency symbol and the sign string are adjacent, a space separates them from the value; otherwise a space separates the currency symbol and the value.

  2 If the currency symbol and the sign string are adjacent, a space separates them from the value; otherwise a space separates the sign string and the value.

n_cs_precedes
: followed by an integer that indicates the placement of *currency_symbol* for a negative formatted monetary quantity. The same values are recognized as for *p_cs_precedes*.

n_sep_by_space
followed by an integer that indicates the separation of *currency_symbol*, the sign string, and the value for a negative formatted monetary quantity. The same values are recognized as for *p_sep_by_space*.

p_sign_posn
followed by an integer that indicates where the *positive_sign* should be placed for a nonnegative monetary quantity:

0 Parentheses enclose the quantity and the *currency_symbol* or *int_curr_symbol*.

1 The sign string precedes the quantity and the *currency_symbol* or the *int_curr_symbol*.

2 The sign string succeeds the quantity and the *currency_symbol* or the *int_curr_symbol*.

3 The sign string precedes the *currency_symbol* or the *int_curr_symbol*.

4 The sign string succeeds the *currency_symbol* or the *int_curr_symbol*.

n_sign_posn
followed by an integer that indicates where the *negative_sign* should be placed for a negative monetary quantity. The same values are recognized as for *p_sign_posn*.

int_p_cs_precedes
followed by an integer that indicates the placement of *int_curr_symbol* for a nonnegative internationally formatted monetary quantity. The same values are recognized as for *p_cs_precedes*.

int_n_cs_precedes
followed by an integer that indicates the placement of *int_curr_symbol* for a negative internationally formatted monetary quantity. The same values are recognized as for *p_cs_precedes*.

int_p_sep_by_space
followed by an integer that indicates the separation of *int_curr_symbol*, the sign string, and the value for a nonnegative internationally formatted monetary quantity. The same values are recognized as for *p_sep_by_space*.

int_n_sep_by_space
followed by an integer that indicates the separation of *int_curr_symbol*, the sign string, and the value for a negative internationally formatted monetary quantity. The same values are recognized as for *p_sep_by_space*.

int_p_sign_posn
followed by an integer that indicates where the *positive_sign* should be placed for a nonnegative internationally formatted monetary quantity. The same values are recognized as for *p_sign_posn*.

int_n_sign_posn
followed by an integer that indicates where the *negative_sign* should be placed for a negative internationally formatted monetary quantity. The same values are recognized as for *p_sign_posn*.

The **LC_MONETARY** definition ends with the string *END LC_MONETARY*.

LC_NAME

The definition starts with the string *LC_NAME* in the first column.

Various keywords are allowed, but only *name_fmt* is mandatory. Other keywords are needed only if there is common convention to use the corresponding salutation in this locale. The allowed keywords are as follows:

name_fmt
followed by a string containing field descriptors that define the format used for names in the locale. The following field descriptors are recognized:

%f Family name(s).

%F Family names in uppercase.

%g First given name.

%G
First given initial.

%l First given name with Latin letters.

%o Other shorter name.

%m
Additional given name(s).

%M
Initials for additional given name(s).

%p Profession.

%s Salutation, such as "Doctor".

%S Abbreviated salutation, such as "Mr." or "Dr.".

%d Salutation, using the FDCC-sets conventions.

%t If the preceding field descriptor resulted in an empty string, then the empty string, otherwise a space character.

name_gen
followed by the general salutation for any gender.

name_mr
followed by the salutation for men.

name_mrs
followed by the salutation for married women.

name_miss
followed by the salutation for unmarried women.

name_ms
followed by the salutation valid for all women.

The **LC_NAME** definition ends with the string *END LC_NAME*.

LC_NUMERIC

The definition starts with the string *LC_NUMERIC* in the first column.

The following keywords are allowed:

decimal_point
followed by the single-character string that will be used as the decimal delimiter when formatting numeric quantities.

thousands_sep
followed by the single-character string that will be used as a group separator when formatting numeric quantities.

grouping
followed by a sequence of integers separated by semicolons that describe the formatting of numeric quantities.

Each integer specifies the number of digits in a group. The first integer defines the size of the group immediately to the left of the decimal delimiter. Subsequent integers define succeeding groups to the left of the previous group. If the last integer is not −1, then the size of the previous group (if any) is repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping is performed.

The **LC_NUMERIC** definition ends with the string *END LC_NUMERIC*.

LC_PAPER

The definition starts with the string *LC_PAPER* in the first column.

The following keywords are allowed:

height followed by the height, in millimeters, of the standard paper format.

width followed by the width, in millimeters, of the standard paper format.

The **LC_PAPER** definition ends with the string *END LC_PAPER*.

LC_TELEPHONE

The definition starts with the string *LC_TELEPHONE* in the first column.

The following keywords are allowed:

tel_int_fmt
followed by a string that contains field descriptors that identify the format used to dial international numbers. The following field descriptors are recognized:

%a Area code without nationwide prefix (the prefix is often "00").

%A
Area code including nationwide prefix.

%l Local number (within area code).

%e Extension (to local number).

%c Country code.

%C Alternate carrier service code used for dialing abroad.

%t If the preceding field descriptor resulted in an empty string, then the empty string, otherwise a space character.

tel_dom_fmt
followed by a string that contains field descriptors that identify the format used to dial domestic numbers. The recognized field descriptors are the same as for *tel_int_fmt*.

int_select
followed by the prefix used to call international phone numbers.

int_prefix
followed by the prefix used from other countries to dial this country.

The **LC_TELEPHONE** definition ends with the string *END LC_TELEPHONE*.

LC_TIME

The definition starts with the string *LC_TIME* in the first column.

The following keywords are allowed:

abday followed by a list of abbreviated names of the days of the week. The list starts with the first day of the week as specified by *week* (Sunday by default). See NOTES.

day followed by a list of names of the days of the week. The list starts with the first day of the week as specified by *week* (Sunday by default). See NOTES.

abmon followed by a list of abbreviated month names.

mon followed by a list of month names.

d_t_fmt
followed by the appropriate date and time format (for syntax, see *strftime*(3)).

d_fmt followed by the appropriate date format (for syntax, see *strftime*(3)).

t_fmt followed by the appropriate time format (for syntax, see *strftime*(3)).

am_pm followed by the appropriate representation of the **am** and **pm** strings. This should be left empty for locales not using AM/PM convention.

t_fmt_ampm
followed by the appropriate time format (for syntax, see *strftime*(3)) when using 12h clock format. This should be left empty for locales not using AM/PM convention.

era followed by semicolon-separated strings that define how years are counted and displayed for each era in the locale. Each string has the following format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

The fields are to be defined as follows:

direction
: Either + or -. + means the years closer to *start_date* have lower numbers than years closer to *end_date*. − means the opposite.

offset
: The number of the year closest to *start_date* in the era, corresponding to the *%Ey* descriptor (see *strptime*(3)).

start_date
: The start of the era in the form of *yyyy/mm/dd*. Years prior AD 1 are represented as negative numbers.

end_date
: The end of the era in the form of *yyyy/mm/dd*, or one of the two special values of **−*** or **+***. **−*** means the ending date is the beginning of time. **+*** means the ending date is the end of time.

era_name
: The name of the era corresponding to the *%EC* descriptor (see *strptime*(3)).

era_format
: The format of the year in the era corresponding to the *%EY* descriptor (see *strptime*(3)).

era_d_fmt
: followed by the format of the date in alternative era notation, corresponding to the *%Ex* descriptor (see *strptime*(3)).

era_t_fmt
: followed by the format of the time in alternative era notation, corresponding to the *%EX* descriptor (see *strptime*(3)).

era_d_t_fmt
: followed by the format of the date and time in alternative era notation, corresponding to the *%Ec* descriptor (see *strptime*(3)).

alt_digits
: followed by the alternative digits used for date and time in the locale.

week
: followed by a list of three values separated by semicolons: The number of days in a week (by default 7), a date of beginning of the week (by default corresponds to Sunday), and the minimal length of the first week in year (by default 4). Regarding the start of the week, **19971130** shall be used for Sunday and **19971201** shall be used for Monday. See NOTES.

first_weekday (since glibc 2.2)
: followed by the number of the day from the *day* list to be shown as the first day of the week in calendar applications. The default value of **1** corresponds to either Sunday or Monday depending on the value of the second *week* list item. See NOTES.

first_workday (since glibc 2.2)
: followed by the number of the first working day from the *day* list. The default value is **2**. See NOTES.

cal_direction
: followed by a number value that indicates the direction for the display of calendar dates, as follows:

  1 Left-right from top.

  2 Top-down from left.

  3 Right-left from top.

date_fmt
: followed by the appropriate date representation for *date*(1) (for syntax, see *strftime*(3)).

The **LC_TIME** definition ends with the string *END LC_TIME*.

FILES

/usr/lib/locale/locale−archive
: Usual default locale archive location.

/usr/share/i18n/locales
: Usual default path for locale definition files.

STANDARDS

POSIX.2.

NOTES

The collective GNU C library community wisdom regarding *abday*, *day*, *week*, *first_weekday*, and *first_workday* states at https://sourceware.org/glibc/wiki/Locales the following:

- The value of the second *week* list item specifies the base of the *abday* and *day* lists.
- *first_weekday* specifies the offset of the first day-of-week in the *abday* and *day* lists.
- For compatibility reasons, all glibc locales should set the value of the second *week* list item to **19971130** (Sunday) and base the *abday* and *day* lists appropriately, and set *first_weekday* and *first_workday* to **1** or **2**, depending on whether the week and work week actually starts on Sunday or Monday for the locale.

SEE ALSO

iconv(1), *locale*(1), *localedef*(1), *localeconv*(3), *newlocale*(3), *setlocale*(3), *strftime*(3), *strptime*(3), *uselocale*(3), *charmap*(5), *charsets*(7), *locale*(7), *unicode*(7), *utf-8*(7)

NAME

motd − message of the day

DESCRIPTION

The contents of */etc/motd* are displayed by *login*(1) after a successful login but just before it executes the login shell.

The abbreviation "motd" stands for "message of the day", and this file has been traditionally used for exactly that (it requires much less disk space than mail to all users).

FILES

/etc/motd

SEE ALSO

login(1), *issue*(5)

NAME

networks − network name information

DESCRIPTION

The file */etc/networks* is a plain ASCII file that describes known DARPA networks and symbolic names for these networks. Each line represents a network and has the following structure:

name number aliases ...

where the fields are delimited by spaces or tabs. Empty lines are ignored. The hash character (#) indicates the start of a comment: this character, and the remaining characters up to the end of the current line, are ignored by library functions that process the file.

The field descriptions are:

name The symbolic name for the network. Network names can contain any printable characters except white-space characters or the comment character.

number
The official number for this network in numbers-and-dots notation (see *inet*(3)). The trailing ".0" (for the host component of the network address) may be omitted.

aliases Optional aliases for the network.

This file is read by the *route*(8) and *netstat*(8) utilities. Only Class A, B, or C networks are supported, partitioned networks (i.e., network/26 or network/28) are not supported by this file.

FILES

/etc/networks
The networks definition file.

SEE ALSO

getnetbyaddr(3), *getnetbyname*(3), *getnetent*(3), *netstat*(8), *route*(8)

NAME

nologin − prevent unprivileged users from logging into the system

DESCRIPTION

If the file */etc/nologin* exists and is readable, *login*(1) will allow access only to root. Other users will be shown the contents of this file and their logins will be refused. This provides a simple way of temporarily disabling all unprivileged logins.

FILES

/etc/nologin

SEE ALSO

login(1), *shutdown*(8)

NAME

nscd.conf − name service cache daemon configuration file

DESCRIPTION

The file */etc/nscd.conf* is read from *nscd*(8) at startup. Each line specifies either an attribute and a value, or an attribute, service, and a value. Fields are separated either by SPACE or TAB characters. A '#' (number sign) indicates the beginning of a comment; following characters, up to the end of the line, are not interpreted by nscd.

Valid services are *passwd*, *group*, *hosts*, *services*, or *netgroup*.

logfile *debug-file-name*
Specifies name of the file to which debug info should be written.

debug−level *value*
Sets the desired debug level. 0 hides debug info. 1 shows general debug info. 2 additionally shows data in cache dumps. 3 (and above) shows all debug info. The default is 0.

threads *number*
This is the initial number of threads that are started to wait for requests. At least five threads will always be created. The number of threads may increase dynamically up to **max−threads** in response to demand from clients, but never decreases.

max−threads *number*
Specifies the maximum number of threads. The default is 32.

server−user *user*
If this option is set, nscd will run as this user and not as root. If a separate cache for every user is used (−S parameter), this option is ignored.

stat−user *user*
Specifies the user who is allowed to request statistics.

reload−count unlimited | *number*
Sets a limit on the number of times a cached entry gets reloaded without being used before it gets removed. The limit can take values ranging from 0 to 254; values 255 or higher behave the same as **unlimited**. Limit values can be specified in either decimal or hexadecimal with a "0x" prefix. The special value **unlimited** is case-insensitive. The default limit is 5. A limit of 0 turns off the reloading feature. See NOTES below for further discussion of reloading.

paranoia <*yes*|*no*>
Enabling paranoia mode causes nscd to restart itself periodically. The default is no.

restart−interval *time*
Sets the restart interval to *time* seconds if periodic restart is enabled by enabling **paranoia** mode. The default is 3600.

enable−cache *service* <*yes*|*no*>
Enables or disables the specified *service* cache. The default is no.

positive−time−to−live *service value*
Sets the TTL (time-to-live) for positive entries (successful queries) in the specified cache for *service*. *Value* is in seconds. Larger values increase cache hit rates and reduce mean response times, but increase problems with cache coherence. Note that for some name services (including specifically DNS) the TTL returned from the name service is used and this attribute is ignored.

negative−time−to−live *service value*
Sets the TTL (time-to-live) for negative entries (unsuccessful queries) in the specified cache for *service*. *Value* is in seconds. Can result in significant performance improvements if there are several files owned by UIDs (user IDs) not in system databases (for example untarring the Linux kernel sources as root); should be kept small to reduce cache coherency problems.

suggested−size *service value*
This is the internal hash table size, *value* should remain a prime number for optimum efficiency. The default is 211.

check−files *service* <*yes*|*no*>

Enables or disables checking the file belonging to the specified *service* for changes. The files are */etc/passwd*, */etc/group*, */etc/hosts*, */etc/resolv.conf*, */etc/services*, and */etc/netgroup*. The default is yes.

persistent *service* <*yes*|*no*>
Keep the content of the cache for *service* over server restarts; useful when **paranoia** mode is set. The default is no.

shared *service* <*yes*|*no*>
The memory mapping of the nscd databases for *service* is shared with the clients so that they can directly search in them instead of having to ask the daemon over the socket each time a lookup is performed. The default is no. Note that a cache miss will still result in asking the daemon over the socket.

max-db-size *service bytes*
The maximum allowable size, in bytes, of the database files for the *service*. The default is 33554432.

auto-propagate *service* <*yes*|*no*>
When set to *no* for *passwd* or *group* service, then the *.byname* requests are not added to *passwd.byuid* or *group.bygid* cache. This can help with tables containing multiple records for the same ID. The default is yes. This option is valid only for services *passwd* and *group*.

NOTES

The default values stated in this manual page originate from the source code of *nscd*(8) and are used if not overridden in the configuration file. The default values used in the configuration file of your distribution might differ.

Reloading

nscd(8) has a feature called reloading, whose behavior can be surprising.

Reloading is enabled when the **reload-count** attribute has a non-zero value. The default value in the source code enables reloading, although your distribution may differ.

When reloading is enabled, positive cached entries (the results of successful queries) do not simply expire when their TTL is up. Instead, at the expiry time, **nscd** will "reload", i.e., re-issue to the name service the same query that created the cached entry, to get a new value to cache. Depending on */etc/nsswitch.conf* this may mean that a DNS, LDAP, or NIS request is made. If the new query is successful, reloading will repeat when the new value would expire, until **reload-count** reloads have happened for the entry, and only then will it actually be removed from the cache. A request from a client which hits the entry will reset the reload counter on the entry. Purging the cache using *nscd -i* overrides the reload logic and removes the entry.

Reloading has the effect of extending cache entry TTLs without compromising on cache coherency, at the cost of additional load on the backing name service. Whether this is a good idea on your system depends on details of your applications' behavior, your name service, and the effective TTL values of your cache entries. Note that for some name services (for example, DNS), the effective TTL is the value returned from the name service and *not* the value of the **positive-time-to-live** attribute.

Please consider the following advice carefully:

- If your application will make a second request for the same name, after more than 1 TTL but before **reload-count** TTLs, and is sensitive to the latency of a cache miss, then reloading may be a good idea for you.
- If your name service is configured to return very short TTLs, and your applications only make requests rarely under normal circumstances, then reloading may result in additional load on your backing name service without any benefit to applications, which is probably a bad idea for you.
- If your name service capacity is limited, reloading may have the surprising effect of increasing load on your name service instead of reducing it, and may be a bad idea for you.
- Setting **reload-count** to **unlimited** is almost never a good idea, as it will result in a cache that never expires entries and puts never-ending additional load on the backing name service.

Some distributions have an init script for *nscd*(8) with a *reload* command which uses *nscd -i* to purge the cache. That use of the word "reload" is entirely different from the "reloading" described here.

SEE ALSO

nscd(8)

NAME

nss − Name Service Switch configuration file

DESCRIPTION

Each call to a function which retrieves data from a system database like the password or group database is handled by the Name Service Switch implementation in the GNU C library. The various services provided are implemented by independent modules, each of which naturally varies widely from the other.

The default implementations coming with the GNU C library are by default conservative and do not use unsafe data. This might be very costly in some situations, especially when the databases are large. Some modules allow the system administrator to request taking shortcuts if these are known to be safe. It is then the system administrator's responsibility to ensure the assumption is correct.

There are other modules where the implementation changed over time. If an implementation used to sacrifice speed for memory consumption, it might create problems if the preference is switched.

The */etc/default/nss* file contains a number of variable assignments. Each variable controls the behavior of one or more NSS modules. White spaces are ignored. Lines beginning with '#' are treated as comments.

The variables currently recognized are:

NETID_AUTHORITATIVE = *TRUE*|*FALSE*

If set to TRUE, the NIS backend for the *initgroups*(3) function will accept the information from the *netid.byname* NIS map as authoritative. This can speed up the function significantly if the *group.byname* map is large. The content of the *netid.byname* map is used **as is**. The system administrator has to make sure it is correctly generated.

SERVICES_AUTHORITATIVE = *TRUE*|*FALSE*

If set to TRUE, the NIS backend for the *getservbyname*(3) and *getservbyname_r*(3) functions will assume that the *services.byservicename* NIS map exists and is authoritative, particularly that it contains both keys with /proto and without /proto for both primary service names and service aliases. The system administrator has to make sure it is correctly generated.

SETENT_BATCH_READ = *TRUE*|*FALSE*

If set to TRUE, the NIS backend for the *setpwent*(3) and *setgrent*(3) functions will read the entire database at once and then hand out the requests one by one from memory with every corresponding *getpwent*(3) or *getgrent*(3) call respectively. Otherwise, each *getpwent*(3) or *getgrent*(3) call might result in a network communication with the server to get the next entry.

FILES

/etc/default/nss

EXAMPLES

The default configuration corresponds to the following configuration file:

```
NETID_AUTHORITATIVE=FALSE
SERVICES_AUTHORITATIVE=FALSE
SETENT_BATCH_READ=FALSE
```

SEE ALSO

nsswitch.conf

NAME

nsswitch.conf − Name Service Switch configuration file

DESCRIPTION

The Name Service Switch (NSS) configuration file, */etc/nsswitch.conf*, is used by the GNU C Library and certain other applications to determine the sources from which to obtain name-service information in a range of categories, and in what order. Each category of information is identified by a database name.

The file is plain ASCII text, with columns separated by spaces or tab characters. The first column specifies the database name. The remaining columns describe the order of sources to query and a limited set of actions that can be performed by lookup result.

The following databases are understood by the GNU C Library:

aliases Mail aliases, used by *getaliasent*(3) and related functions.

ethers Ethernet numbers.

group Groups of users, used by *getgrent*(3) and related functions.

hosts Host names and numbers, used by *gethostbyname*(3) and related functions.

initgroups Supplementary group access list, used by *getgrouplist*(3) function.

netgroup Network-wide list of hosts and users, used for access rules. C libraries before glibc 2.1 supported netgroups only over NIS.

networks Network names and numbers, used by *getnetent*(3) and related functions.

passwd User passwords, used by *getpwent*(3) and related functions.

protocols Network protocols, used by *getprotoent*(3) and related functions.

publickey Public and secret keys for Secure_RPC used by NFS and NIS+.

rpc Remote procedure call names and numbers, used by *getrpcbyname*(3) and related functions.

services Network services, used by *getservent*(3) and related functions.

shadow Shadow user passwords, used by *getspnam*(3) and related functions.

The GNU C Library ignores databases with unknown names. Some applications use this to implement special handling for their own databases. For example, *sudo*(8) consults the **sudoers** database. Delegation of subordinate user/group IDs can be configured using the **subid** database. Refer to *subuid*(5) and *subgid*(5) for more details.

Here is an example */etc/nsswitch.conf* file:

```
passwd:         compat
group:          compat
shadow:         compat

hosts:          dns [!UNAVAIL=return] files
networks:       nis [NOTFOUND=return] files
ethers:         nis [NOTFOUND=return] files
protocols:      nis [NOTFOUND=return] files
rpc:            nis [NOTFOUND=return] files
services:       nis [NOTFOUND=return] files
```

The first column is the database name. The remaining columns specify:

- One or more service specifications, for example, "files", "db", or "nis". The order of the services on the line determines the order in which those services will be queried, in turn, until a result is found.
- Optional actions to perform if a particular result is obtained from the preceding service, for example, "[NOTFOUND=return]".

The service specifications supported on your system depend on the presence of shared libraries, and are therefore extensible. Libraries called */lib/libnss_SERVICE.so.***X** will provide the named *SERVICE*. On a standard installation, you can use "files", "db", "nis", and "nisplus". For the **hosts** database, you can

additionally specify "dns". For the **passwd**, **group**, and **shadow** databases, you can additionally specify "compat" (see **Compatibility mode** below). The version number **X** may be 1 for glibc 2.0, or 2 for glibc 2.1 and later. On systems with additional libraries installed, you may have access to further services such as "hesiod", "ldap", "winbind", and "wins".

An action may also be specified following a service specification. The action modifies the behavior following a result obtained from the preceding data source. Action items take the general form:

[*STATUS=ACTION*]
[!*STATUS=ACTION*]

where

STATUS => **success** | **notfound** | **unavail** | **tryagain**
ACTION => **return** | **continue** | **merge**

The ! negates the test, matching all possible results except the one specified. The case of the keywords is not significant.

The *STATUS* value is matched against the result of the lookup function called by the preceding service specification, and can be one of:

success No error occurred and the requested entry is returned. The default action for this condition is "return".

notfound The lookup succeeded, but the requested entry was not found. The default action for this condition is "continue".

unavail The service is permanently unavailable. This can mean either that the required file cannot be read, or, for network services, that the server is not available or does not allow queries. The default action for this condition is "continue".

tryagain The service is temporarily unavailable. This could mean a file is locked or a server currently cannot accept more connections. The default action for this condition is "continue".

The *ACTION* value can be one of:

return Return a result now. Do not call any further lookup functions. However, for compatibility reasons, if this is the selected action for the **group** database and the **notfound** status, and the configuration file does not contain the **initgroups** line, the next lookup function is always called, without affecting the search result.

continue Call the next lookup function.

merge *[SUCCESS=merge]* is used between two database entries. When a group is located in the first of the two group entries, processing will continue on to the next one. If the group is also found in the next entry (and the group name and GID are an exact match), the member list of the second entry will be added to the group object to be returned. Available since glibc 2.24. Note that merging will not be done for *getgrent*(3) nor will duplicate members be pruned when they occur in both entries being merged.

Compatibility mode (compat)

The NSS "compat" service is similar to "files" except that it additionally permits special entries in corresponding files for granting users or members of netgroups access to the system. The following entries are valid in this mode:

For **passwd** and **shadow** databases:

+user Include the specified *user* from the NIS passwd/shadow map.

+@netgroup Include all users in the given *netgroup*.

−user Exclude the specified *user* from the NIS passwd/shadow map.

−@netgroup Exclude all users in the given *netgroup*.

+ Include every user, except previously excluded ones, from the NIS passwd/shadow map.

For **group** database:

+*group* Include the specified *group* from the NIS group map.

−*group* Exclude the specified *group* from the NIS group map.

\+ Include every group, except previously excluded ones, from the NIS group map.

By default, the source is "nis", but this may be overridden by specifying any NSS service except "compat" itself as the source for the pseudo-databases **passwd_compat**, **group_compat**, and **shadow_compat**.

FILES

A service named *SERVICE* is implemented by a shared object library named *libnss_SERVICE.so.***X** that resides in */lib*.

/etc/nsswitch.conf	NSS configuration file.
*/lib/libnss_compat.so.***X**	implements "compat" source.
*/lib/libnss_db.so.***X**	implements "db" source.
*/lib/libnss_dns.so.***X**	implements "dns" source.
*/lib/libnss_files.so.***X**	implements "files" source.
*/lib/libnss_hesiod.so.***X**	implements "hesiod" source.
*/lib/libnss_nis.so.***X**	implements "nis" source.
*/lib/libnss_nisplus.so.***X**	implements "nisplus" source.

The following files are read when "files" source is specified for respective databases:

aliases	*/etc/aliases*
ethers	*/etc/ethers*
group	*/etc/group*
hosts	*/etc/hosts*
initgroups	*/etc/group*
netgroup	*/etc/netgroup*
networks	*/etc/networks*
passwd	*/etc/passwd*
protocols	*/etc/protocols*
publickey	*/etc/publickey*
rpc	*/etc/rpc*
services	*/etc/services*
shadow	*/etc/shadow*

NOTES

Starting with glibc 2.33, **nsswitch.conf** is automatically reloaded if the file is changed. In earlier versions, the entire file was read only once within each process. If the file was later changed, the process would continue using the old configuration.

Traditionally, there was only a single source for service information, often in the form of a single configuration file (e.g., */etc/passwd*). However, as other name services, such as the Network Information Service (NIS) and the Domain Name Service (DNS), became popular, a method was needed that would be more flexible than fixed search orders coded into the C library. The Name Service Switch mechanism, which was based on the mechanism used by Sun Microsystems in the Solaris 2 C library, introduced a cleaner solution to the problem.

SEE ALSO

getent(1), *nss*(5)

NAME

passwd − password file

DESCRIPTION

The */etc/passwd* file is a text file that describes user login accounts for the system. It should have read permission allowed for all users (many utilities, like *ls*(1) use it to map user IDs to usernames), but write access only for the superuser.

In the good old days there was no great problem with this general read permission. Everybody could read the encrypted passwords, but the hardware was too slow to crack a well-chosen password, and moreover the basic assumption used to be that of a friendly user-community. These days many people run some version of the shadow password suite, where */etc/passwd* has an 'x' character in the password field, and the encrypted passwords are in */etc/shadow*, which is readable by the superuser only.

If the encrypted password, whether in */etc/passwd* or in */etc/shadow*, is an empty string, login is allowed without even asking for a password. Note that this functionality may be intentionally disabled in applications, or configurable (for example using the "**nullok**" or "**nonull**" arguments to *pam_unix*(8)).

If the encrypted password in */etc/passwd* is "*NP*" (without the quotes), the shadow record should be obtained from an NIS+ server.

Regardless of whether shadow passwords are used, many system administrators use an asterisk (*) in the encrypted password field to make sure that this user can not authenticate themself using a password. (But see NOTES below.)

If you create a new login, first put an asterisk (*) in the password field, then use *passwd*(1) to set it.

Each line of the file describes a single user, and contains seven colon-separated fields:

```
name:password:UID:GID:GECOS:directory:shell
```

The field are as follows:

name This is the user's login name. It should not contain capital letters.

password This is either the encrypted user password, an asterisk (*), or the letter 'x'. (See *pwconv*(8) for an explanation of 'x'.)

UID The privileged *root* login account (superuser) has the user ID 0.

GID This is the numeric primary group ID for this user. (Additional groups for the user are defined in the system group file; see *group*(5)).

GECOS This field (sometimes called the "comment field") is optional and used only for informational purposes. Usually, it contains the full username. Some programs (for example, *finger*(1)) display information from this field.

GECOS stands for "General Electric Comprehensive Operating System", which was renamed to GCOS when GE's large systems division was sold to Honeywell. Dennis Ritchie has reported: "Sometimes we sent printer output or batch jobs to the GCOS machine. The gcos field in the password file was a place to stash the information for the $IDENTcard. Not elegant."

directory This is the user's home directory: the initial directory where the user is placed after logging in. The value in this field is used to set the **HOME** environment variable.

shell This is the program to run at login (if empty, use */bin/sh*). If set to a nonexistent executable, the user will be unable to login through *login*(1)The value in this field is used to set the **SHELL** environment variable.

FILES

/etc/passwd

NOTES

If you want to create user groups, there must be an entry in */etc/group*, or no group will exist.

If the encrypted password is set to an asterisk (*), the user will be unable to login using *login*(1), but may still login using *rlogin*(1), run existing processes and initiate new ones through *rsh*(1), *cron*(8), *at*(1), or mail filters, etc. Trying to lock an account by simply changing the shell field yields the same result and additionally allows the use of *su*(1)

SEE ALSO

chfn(1), *chsh*(1), *login*(1), *passwd*(1), *su*(1), *crypt*(3), *getpwent*(3), *getpwnam*(3), *group*(5), *shadow*(5), *vipw*(8)

NAME

proc − process information, system information, and sysctl pseudo-filesystem

DESCRIPTION

The **proc** filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at */proc*. Typically, it is mounted automatically by the system, but it can also be mounted manually using a command such as:

```
mount -t proc proc /proc
```

Most of the files in the **proc** filesystem are read-only, but some files are writable, allowing kernel variables to be changed.

Mount options

The **proc** filesystem supports the following mount options:

hidepid=*n* (since Linux 3.3)
: This option controls who can access the information in */proc/*pid directories. The argument, *n*, is one of the following values:

  Everybody may access all */proc/*pid directories. This is the traditional behavior, and the default if this mount option is not specified.

  1 Users may not access files and subdirectories inside any */proc/*pid directories but their own (the */proc/*pid directories themselves remain visible). Sensitive files such as */proc/*pid*/cmdline* and */proc/*pid*/status* are now protected against other users. This makes it impossible to learn whether any user is running a specific program (so long as the program doesn't otherwise reveal itself by its behavior).

  2 As for mode 1, but in addition the */proc/*pid directories belonging to other users become invisible. This means that */proc/*pid entries can no longer be used to discover the PIDs on the system. This doesn't hide the fact that a process with a specific PID value exists (it can be learned by other means, for example, by "kill −0 $PID"), but it hides a process's UID and GID, which could otherwise be learned by employing *stat*(2) on a */proc/*pid directory. This greatly complicates an attacker's task of gathering information about running processes (e.g., discovering whether some daemon is running with elevated privileges, whether another user is running some sensitive program, whether other users are running any program at all, and so on).

gid=*gid* (since Linux 3.3)
: Specifies the ID of a group whose members are authorized to learn process information otherwise prohibited by **hidepid** (i.e., users in this group behave as though */proc* was mounted with *hidepid=0*). This group should be used instead of approaches such as putting nonroot users into the *sudoers*(5) file.

Overview

Underneath */proc*, there are the following general groups of files and subdirectories:

*/proc/*pid subdirectories
: Each one of these subdirectories contains files and subdirectories exposing information about the process with the corresponding process ID.

  Underneath each of the */proc/*pid directories, a *task* subdirectory contains subdirectories of the form *task/*tid, which contain corresponding information about each of the threads in the process, where *tid* is the kernel thread ID of the thread.

  The */proc/*pid subdirectories are visible when iterating through */proc* with *getdents*(2) (and thus are visible when one uses *ls*(1) to view the contents of */proc*).

*/proc/*tid subdirectories
: Each one of these subdirectories contains files and subdirectories exposing information about the thread with the corresponding thread ID. The contents of these directories are the same as the corresponding */proc/*pid*/task/*tid directories.

  The */proc/*tid subdirectories are *not* visible when iterating through */proc* with *getdents*(2) (and thus are *not* visible when one uses *ls*(1) to view the contents of */proc*).

/proc/self
: When a process accesses this magic symbolic link, it resolves to the process's own */proc/*pid directory.

/proc/thread-self
: When a thread accesses this magic symbolic link, it resolves to the process's own */proc/self/task/*tid directory.

*/proc/[a-z]**
: Various other files and subdirectories under */proc* expose system-wide information.

All of the above are described in more detail below.

NOTES

Many files contain strings (e.g., the environment and command line) that are in the internal format, with subfields terminated by null bytes ('\0'). When inspecting such files, you may find that the results are more readable if you use a command of the following form to display them:

```
$ cat file | tr '\000' '\n'
```

SEE ALSO

cat(1), *dmesg*(1), *find*(1), *free*(1), *htop*(1), *init*(1), *ps*(1), *pstree*(1), *tr*(1), *uptime*(1), *chroot*(2), *mmap*(2), *readlink*(2), *syslog*(2), *slabinfo*(5), *sysfs*(5), *hier*(7), *namespaces*(7), *time*(7), *arp*(8), *hdparm*(8), *ifconfig*(8), *lsmod*(8), *lspci*(8), *mount*(8), *netstat*(8), *procinfo*(8), *route*(8), *sysctl*(8)

The Linux kernel source files: *Documentation/filesystems/proc.rst*, *Documentation/admin-guide/sysctl/fs.rst*, *Documentation/admin-guide/sysctl/kernel.rst*, *Documentation/admin-guide/sysctl/net.rst*, and *Documentation/admin-guide/sysctl/vm.rst*.

NAME

/proc/apm − advanced power management

DESCRIPTION

/proc/apm

Advanced power management version and battery information when **CONFIG_APM** is defined at kernel compilation time.

SEE ALSO

proc(5)

NAME

/proc/buddyinfo − memory fragmentation

DESCRIPTION

/proc/buddyinfo

This file contains information which is used for diagnosing memory fragmentation issues. Each line starts with the identification of the node and the name of the zone which together identify a memory region. This is then followed by the count of available chunks of a certain order in which these zones are split. The size in bytes of a certain order is given by the formula:

```
(2^order) * PAGE_SIZE
```

The binary buddy allocator algorithm inside the kernel will split one chunk into two chunks of a smaller order (thus with half the size) or combine two contiguous chunks into one larger chunk of a higher order (thus with double the size) to satisfy allocation requests and to counter memory fragmentation. The order matches the column number, when starting to count at zero.

For example on an x86-64 system:

```
Node 0, zone      DMA      1    1    1    0    2    1    1    0    1    1    3
Node 0, zone    DMA32     65   47    4   81   52   28   13   10    5    1  404
Node 0, zone   Normal    216   55  189  101   84   38   37   27    5    3  587
```

In this example, there is one node containing three zones and there are 11 different chunk sizes. If the page size is 4 kilobytes, then the first zone called *DMA* (on x86 the first 16 megabyte of memory) has 1 chunk of 4 kilobytes (order 0) available and has 3 chunks of 4 megabytes (order 10) available.

If the memory is heavily fragmented, the counters for higher order chunks will be zero and allocation of large contiguous areas will fail.

Further information about the zones can be found in */proc/zoneinfo*.

SEE ALSO

proc(5)

NAME

/proc/bus/ − installed buses

DESCRIPTION

/proc/bus/
: Contains subdirectories for installed buses.

/proc/bus/pccard/
: Subdirectory for PCMCIA devices when **CONFIG_PCMCIA** is set at kernel compilation time.

/proc/bus/pccard/drivers

/proc/bus/pci/
: Contains various bus subdirectories and pseudo-files containing information about PCI buses, installed devices, and device drivers. Some of these files are not ASCII.

/proc/bus/pci/devices
: Information about PCI devices. They may be accessed through *lspci*(8) and *setpci*(8)

SEE ALSO

proc(5)

NAME

/proc/cgroups − control groups

DESCRIPTION

/proc/cgroups (since Linux 2.6.24)

See *cgroups*(7).

SEE ALSO

proc(5)

NAME

/proc/cmdline − kernel boot arguments

DESCRIPTION

/proc/cmdline

Arguments passed to the Linux kernel at boot time. Often done via a boot manager such as *lilo*(8) or *grub*(8)Any arguments embedded in the kernel image or initramfs via **CONFIG_BOOT_CONFIG** will also be displayed.

SEE ALSO

proc(5)

NAME

/proc/config.gz − kernel build configuration

DESCRIPTION

/proc/config.gz (since Linux 2.6)

This file exposes the configuration options that were used to build the currently running kernel, in the same format as they would be shown in the *.config* file that resulted when configuring the kernel (using *make xconfig*, *make config*, or similar). The file contents are compressed; view or search them using *zcat*(1) and *zgrep*(1)As long as no changes have been made to the following file, the contents of */proc/config.gz* are the same as those provided by:

```
cat /lib/modules/$(uname -r)/build/.config
```

/proc/config.gz is provided only if the kernel is configured with **CONFIG_IKCONFIG_PROC**.

SEE ALSO

proc(5)

NAME

/proc/cpuinfo − CPU and system architecture information

DESCRIPTION

/proc/cpuinfo

This is a collection of CPU and system architecture dependent items, for each supported architecture a different list. Two common entries are *processor* which gives CPU number and *bogomips*; a system constant that is calculated during kernel initialization. SMP machines have information for each CPU. The *lscpu*(1) command gathers its information from this file.

SEE ALSO

proc(5)

NAME

/proc/crypto − ciphers provided by kernel crypto API

DESCRIPTION

/proc/crypto

A list of the ciphers provided by the kernel crypto API. For details, see the kernel *Linux Kernel Crypto API* documentation available under the kernel source directory *Documentation/crypto/* (or *Documentation/DocBook* before Linux 4.10; the documentation can be built using a command such as *make htmldocs* in the root directory of the kernel source tree).

SEE ALSO

proc(5)

NAME

/proc/devices − major numbers and device groups

DESCRIPTION

/proc/devices

Text listing of major numbers and device groups. This can be used by MAKEDEV scripts for consistency with the kernel.

SEE ALSO

proc(5)

NAME

/proc/diskstats − disk I/O statistics

DESCRIPTION

/proc/diskstats (since Linux 2.5.69)

This file contains disk I/O statistics for each disk device. See the Linux kernel source file *Documentation/admin−guide/iostats.rst* (or *Documentation/iostats.txt* before Linux 5.3) for further information.

SEE ALSO

proc(5)

NAME

/proc/dma − ISA DMA channels

DESCRIPTION

/proc/dma

This is a list of the registered *ISA* DMA (direct memory access) channels in use.

SEE ALSO

proc(5)

NAME

/proc/driver/ − empty dir

DESCRIPTION

/proc/driver/

Empty subdirectory.

SEE ALSO

proc(5)

NAME

/proc/execdomains − ABI personalities (obsolete)

DESCRIPTION

/proc/execdomains

Used to list ABI personalities before Linux 4.1; now contains a constant string for userspace compatibility.

SEE ALSO

proc(5)

NAME

/proc/fb − frame buffer

DESCRIPTION

/proc/fb

Frame buffer information when **CONFIG_FB** is defined during kernel compilation.

SEE ALSO

proc(5)

NAME

/proc/filesystems − supported filesystems

DESCRIPTION

/proc/filesystems

A text listing of the filesystems which are supported by the kernel, namely filesystems which were compiled into the kernel or whose kernel modules are currently loaded. (See also *filesystems*(5).) If a filesystem is marked with "nodev", this means that it does not require a block device to be mounted (e.g., virtual filesystem, network filesystem).

Incidentally, this file may be used by *mount*(8) when no filesystem is specified and it didn't manage to determine the filesystem type. Then filesystems contained in this file are tried (excepted those that are marked with "nodev").

SEE ALSO

proc(5)

NAME

/proc/fs/ − mounted filesystems

DESCRIPTION

/proc/fs/

Contains subdirectories that in turn contain files with information about (certain) mounted filesystems.

SEE ALSO

proc(5)

NAME

/proc/ide/ − IDE channels and attached devices

DESCRIPTION

/proc/ide

This directory exists on systems with the IDE bus. There are directories for each IDE channel and attached device. Files include:

```
cache              buffer size in KB
capacity           number of sectors
driver             driver version
geometry           physical and logical geometry
identify           in hexadecimal
media              media type
model              manufacturer's model number
settings           drive settings
smart_thresholds   IDE disk management thresholds (in hex)
smart_values       IDE disk management values (in hex)
```

The *hdparm*(8) utility provides access to this information in a friendly format.

SEE ALSO

proc(5)

NAME

/proc/interrupts − number of interrupts

DESCRIPTION

/proc/interrupts

This is used to record the number of interrupts per CPU per IO device. Since Linux 2.6.24, for the i386 and x86-64 architectures, at least, this also includes interrupts internal to the system (that is, not associated with a device as such), such as NMI (nonmaskable interrupt), LOC (local timer interrupt), and for SMP systems, TLB (TLB flush interrupt), RES (rescheduling interrupt), CAL (remote function call interrupt), and possibly others. Very easy to read formatting, done in ASCII.

SEE ALSO

proc(5)

NAME

/proc/iomem − I/O memory map

DESCRIPTION

/proc/iomem

I/O memory map in Linux 2.4.

SEE ALSO

proc(5)

NAME

/proc/ioports − I/O port regions

DESCRIPTION

/proc/ioports

This is a list of currently registered Input-Output port regions that are in use.

SEE ALSO

proc(5)

NAME

/proc/kallsyms − kernel exported symbols

DESCRIPTION

/proc/kallsyms (since Linux 2.5.71)

This holds the kernel exported symbol definitions used by the *modules*(X) tools to dynamically link and bind loadable modules. In Linux 2.5.47 and earlier, a similar file with slightly different syntax was named *ksyms*.

HISTORY

/proc/ksyms (Linux 1.1.23–2.5.47)

See */proc/kallsyms*.

SEE ALSO

proc(5)

NAME

/proc/kcore − physical memory

DESCRIPTION

/proc/kcore

This file represents the physical memory of the system and is stored in the ELF core file format. With this pseudo-file, and an unstripped kernel (*/usr/src/linux/vmlinux*) binary, GDB can be used to examine the current state of any kernel data structures.

The total length of the file is the size of physical memory (RAM) plus 4 KiB.

SEE ALSO

proc(5)

NAME

/proc/keys, /proc/key−users − in-kernel key management

DESCRIPTION

/proc/keys (since Linux 2.6.10)
See *keyrings*(7).

/proc/key−users (since Linux 2.6.10)
See *keyrings*(7).

SEE ALSO

proc(5)

NAME

/proc/kmsg − kernel messages

DESCRIPTION

/proc/kmsg

This file can be used instead of the *syslog*(2) system call to read kernel messages. A process must have superuser privileges to read this file, and only one process should read this file. This file should not be read if a syslog process is running which uses the *syslog*(2) system call facility to log kernel messages.

Information in this file is retrieved with the *dmesg*(1) program.

SEE ALSO

proc(5)

NAME

/proc/kpagecgroup − memory cgroups

DESCRIPTION

/proc/kpagecgroup (since Linux 4.3)

This file contains a 64-bit inode number of the memory cgroup each page is charged to, indexed by page frame number (see the discussion of */proc/*pid*/pagemap*).

The */proc/kpagecgroup* file is present only if the **CONFIG_MEMCG** kernel configuration option is enabled.

SEE ALSO

proc(5)

NAME

/proc/kpagecount − count of mappings of physical pages

DESCRIPTION

/proc/kpagecount (since Linux 2.6.25)

This file contains a 64-bit count of the number of times each physical page frame is mapped, indexed by page frame number (see the discussion of */proc/*pid*/pagemap*).

The */proc/kpagecount* file is present only if the **CONFIG_PROC_PAGE_MONITOR** kernel configuration option is enabled.

SEE ALSO

proc(5)

NAME

/proc/kpageflags − physical pages frame masks

DESCRIPTION

/proc/kpageflags (since Linux 2.6.25)

This file contains 64-bit masks corresponding to each physical page frame; it is indexed by page frame number (see the discussion of */proc/*pid*/pagemap*). The bits are as follows:

0	-	KPF_LOCKED	
1	-	KPF_ERROR	
2	-	KPF_REFERENCED	
3	-	KPF_UPTODATE	
4	-	KPF_DIRTY	
5	-	KPF_LRU	
6	-	KPF_ACTIVE	
7	-	KPF_SLAB	
8	-	KPF_WRITEBACK	
9	-	KPF_RECLAIM	
10	-	KPF_BUDDY	
11	-	KPF_MMAP	(since Linux 2.6.31)
12	-	KPF_ANON	(since Linux 2.6.31)
13	-	KPF_SWAPCACHE	(since Linux 2.6.31)
14	-	KPF_SWAPBACKED	(since Linux 2.6.31)
15	-	KPF_COMPOUND_HEAD	(since Linux 2.6.31)
16	-	KPF_COMPOUND_TAIL	(since Linux 2.6.31)
17	-	KPF_HUGE	(since Linux 2.6.31)
18	-	KPF_UNEVICTABLE	(since Linux 2.6.31)
19	-	KPF_HWPOISON	(since Linux 2.6.31)
20	-	KPF_NOPAGE	(since Linux 2.6.31)
21	-	KPF_KSM	(since Linux 2.6.32)
22	-	KPF_THP	(since Linux 3.4)
23	-	KPF_BALLOON	(since Linux 3.18)
24	-	KPF_ZERO_PAGE	(since Linux 4.0)
25	-	KPF_IDLE	(since Linux 4.3)
26	-	KPF_PGTABLE	(since Linux 4.18)

For further details on the meanings of these bits, see the kernel source file *Documentation/admin−guide/mm/pagemap.rst*. Before Linux 2.6.29, **KPF_WRITEBACK**, **KPF_RECLAIM**, **KPF_BUDDY**, and **KPF_LOCKED** did not report correctly.

The */proc/kpageflags* file is present only if the **CONFIG_PROC_PAGE_MONITOR** kernel configuration option is enabled.

SEE ALSO

proc(5)

NAME

/proc/loadavg − load average

DESCRIPTION

/proc/loadavg

The first three fields in this file are load average figures giving the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1, 5, and 15 minutes. They are the same as the load average numbers given by *uptime*(1) and other programs. The fourth field consists of two numbers separated by a slash (/). The first of these is the number of currently runnable kernel scheduling entities (processes, threads). The value after the slash is the number of kernel scheduling entities that currently exist on the system. The fifth field is the PID of the process that was most recently created on the system.

SEE ALSO

proc(5)

NAME

/proc/locks − current file locks and leases

DESCRIPTION

/proc/locks

This file shows current file locks (*flock*(2) and *fcntl*(2)) and leases (*fcntl*(2)).

An example of the content shown in this file is the following:

```
1: POSIX  ADVISORY  READ  5433 08:01:7864448 128 128
2: FLOCK  ADVISORY  WRITE 2001 08:01:7864554 0 EOF
3: FLOCK  ADVISORY  WRITE 1568 00:2f:32388 0 EOF
4: POSIX  ADVISORY  WRITE 699 00:16:28457 0 EOF
5: POSIX  ADVISORY  WRITE 764 00:16:21448 0 0
6: POSIX  ADVISORY  READ  3548 08:01:7867240 1 1
7: POSIX  ADVISORY  READ  3548 08:01:7865567 1826 2335
8: OFDLCK ADVISORY  WRITE -1 08:01:8713209 128 191
```

The fields shown in each line are as follows:

[1] The ordinal position of the lock in the list.

[2] The lock type. Values that may appear here include:

FLOCK
This is a BSD file lock created using *flock*(2).

OFDLCK
This is an open file description (OFD) lock created using *fcntl*(2).

POSIX This is a POSIX byte-range lock created using *fcntl*(2).

[3] Among the strings that can appear here are the following:

ADVISORY
This is an advisory lock.

MANDATORY
This is a mandatory lock.

[4] The type of lock. Values that can appear here are:

READ This is a POSIX or OFD read lock, or a BSD shared lock.

WRITE
This is a POSIX or OFD write lock, or a BSD exclusive lock.

[5] The PID of the process that owns the lock.

Because OFD locks are not owned by a single process (since multiple processes may have file descriptors that refer to the same open file description), the value −1 is displayed in this field for OFD locks. (Before Linux 4.14, a bug meant that the PID of the process that initially acquired the lock was displayed instead of the value −1.)

[6] Three colon-separated subfields that identify the major and minor device ID of the device containing the filesystem where the locked file resides, followed by the inode number of the locked file.

[7] The byte offset of the first byte of the lock. For BSD locks, this value is always 0.

[8] The byte offset of the last byte of the lock. **EOF** in this field means that the lock extends to the end of the file. For BSD locks, the value shown is always *EOF*.

Since Linux 4.9, the list of locks shown in */proc/locks* is filtered to show just the locks for the processes in the PID namespace (see *pid_namespaces*(7)) for which the */proc* filesystem was mounted. (In the initial PID namespace, there is no filtering of the records shown in this file.)

The *lslocks*(8) command provides a bit more information about each lock.

SEE ALSO

proc(5)

NAME

/proc/malloc − debug malloc (obsolete)

DESCRIPTION

/proc/malloc (only up to and including Linux 2.2)

This file is present only if **CONFIG_DEBUG_MALLOC** was defined during compilation.

SEE ALSO

proc(5)

NAME

/proc/meminfo − memory usage

DESCRIPTION

/proc/meminfo

This file reports statistics about memory usage on the system. It is used by *free*(1) to report the amount of free and used memory (both physical and swap) on the system as well as the shared memory and buffers used by the kernel. Each line of the file consists of a parameter name, followed by a colon, the value of the parameter, and an option unit of measurement (e.g., "kB"). The list below describes the parameter names and the format specifier required to read the field value. Except as noted below, all of the fields have been present since at least Linux 2.6.0. Some fields are displayed only if the kernel was configured with various options; those dependencies are noted in the list.

MemTotal %lu
: Total usable RAM (i.e., physical RAM minus a few reserved bits and the kernel binary code).

MemFree %lu
: The sum of *LowFree+HighFree*.

MemAvailable %lu (since Linux 3.14)
: An estimate of how much memory is available for starting new applications, without swapping.

Buffers %lu
: Relatively temporary storage for raw disk blocks that shouldn't get tremendously large (20 MB or so).

Cached %lu
: In-memory cache for files read from the disk (the page cache). Doesn't include *SwapCached*.

SwapCached %lu
: Memory that once was swapped out, is swapped back in but still also is in the swap file. (If memory pressure is high, these pages don't need to be swapped out again because they are already in the swap file. This saves I/O.)

Active %lu
: Memory that has been used more recently and usually not reclaimed unless absolutely necessary.

Inactive %lu
: Memory which has been less recently used. It is more eligible to be reclaimed for other purposes.

Active(anon) %lu (since Linux 2.6.28)
: [To be documented.]

Inactive(anon) %lu (since Linux 2.6.28)
: [To be documented.]

Active(file) %lu (since Linux 2.6.28)
: [To be documented.]

Inactive(file) %lu (since Linux 2.6.28)
: [To be documented.]

Unevictable %lu (since Linux 2.6.28)
: (From Linux 2.6.28 to Linux 2.6.30, **CONFIG_UNEVICTABLE_LRU** was required.) [To be documented.]

Mlocked %lu (since Linux 2.6.28)
: (From Linux 2.6.28 to Linux 2.6.30, **CONFIG_UNEVICTABLE_LRU** was required.) [To be documented.]

HighTotal %lu
(Starting with Linux 2.6.19, **CONFIG_HIGHMEM** is required.) Total amount of highmem. Highmem is all memory above ~860 MB of physical memory. Highmem areas are for use by user-space programs, or for the page cache. The kernel must use tricks to access this memory, making it slower to access than lowmem.

HighFree %lu
(Starting with Linux 2.6.19, **CONFIG_HIGHMEM** is required.) Amount of free highmem.

LowTotal %lu
(Starting with Linux 2.6.19, **CONFIG_HIGHMEM** is required.) Total amount of lowmem. Lowmem is memory which can be used for everything that highmem can be used for, but it is also available for the kernel's use for its own data structures. Among many other things, it is where everything from *Slab* is allocated. Bad things happen when you're out of lowmem.

LowFree %lu
(Starting with Linux 2.6.19, **CONFIG_HIGHMEM** is required.) Amount of free lowmem.

MmapCopy %lu (since Linux 2.6.29)
(**CONFIG_MMU** is required.) [To be documented.]

SwapTotal %lu
Total amount of swap space available.

SwapFree %lu
Amount of swap space that is currently unused.

Dirty %lu
Memory which is waiting to get written back to the disk.

Writeback %lu
Memory which is actively being written back to the disk.

AnonPages %lu (since Linux 2.6.18)
Non-file backed pages mapped into user-space page tables.

Mapped %lu
Files which have been mapped into memory (with *mmap*(2)), such as libraries.

Shmem %lu (since Linux 2.6.32)
Amount of memory consumed in *tmpfs*(5) filesystems.

KReclaimable %lu (since Linux 4.20)
Kernel allocations that the kernel will attempt to reclaim under memory pressure. Includes *SReclaimable* (below), and other direct allocations with a shrinker.

Slab %lu
In-kernel data structures cache. (See *slabinfo*(5).)

SReclaimable %lu (since Linux 2.6.19)
Part of *Slab*, that might be reclaimed, such as caches.

SUnreclaim %lu (since Linux 2.6.19)
Part of *Slab*, that cannot be reclaimed on memory pressure.

KernelStack %lu (since Linux 2.6.32)
Amount of memory allocated to kernel stacks.

PageTables %lu (since Linux 2.6.18)
Amount of memory dedicated to the lowest level of page tables.

Quicklists %lu (since Linux 2.6.27)
(**CONFIG_QUICKLIST** is required.) [To be documented.]

NFS_Unstable %lu (since Linux 2.6.18)
NFS pages sent to the server, but not yet committed to stable storage.

Bounce %lu (since Linux 2.6.18)
Memory used for block device "bounce buffers".

WritebackTmp %lu (since Linux 2.6.26)
Memory used by FUSE for temporary writeback buffers.

CommitLimit %lu (since Linux 2.6.10)
This is the total amount of memory currently available to be allocated on the system, expressed in kilobytes. This limit is adhered to only if strict overcommit accounting is enabled (mode 2 in */proc/sys/vm/overcommit_memory*). The limit is calculated according to the formula described under */proc/sys/vm/overcommit_memory*. For further details, see the kernel source file *Documentation/vm/overcommit−accounting.rst*.

Committed_AS %lu
The amount of memory presently allocated on the system. The committed memory is a sum of all of the memory which has been allocated by processes, even if it has not been "used" by them as of yet. A process which allocates 1 GB of memory (using *malloc*(3) or similar), but touches only 300 MB of that memory will show up as using only 300 MB of memory even if it has the address space allocated for the entire 1 GB.

This 1 GB is memory which has been "committed" to by the VM and can be used at any time by the allocating application. With strict overcommit enabled on the system (mode 2 in */proc/sys/vm/overcommit_memory*), allocations which would exceed the *CommitLimit* will not be permitted. This is useful if one needs to guarantee that processes will not fail due to lack of memory once that memory has been successfully allocated.

VmallocTotal %lu
Total size of vmalloc memory area.

VmallocUsed %lu
Amount of vmalloc area which is used. Since Linux 4.4, this field is no longer calculated, and is hard coded as 0. See */proc/vmallocinfo*.

VmallocChunk %lu
Largest contiguous block of vmalloc area which is free. Since Linux 4.4, this field is no longer calculated and is hard coded as 0. See */proc/vmallocinfo*.

HardwareCorrupted %lu (since Linux 2.6.32)
(**CONFIG_MEMORY_FAILURE** is required.) [To be documented.]

LazyFree %lu (since Linux 4.12)
Shows the amount of memory marked by *madvise*(2) **MADV_FREE**.

AnonHugePages %lu (since Linux 2.6.38)
(**CONFIG_TRANSPARENT_HUGEPAGE** is required.) Non-file backed huge pages mapped into user-space page tables.

ShmemHugePages %lu (since Linux 4.8)
(**CONFIG_TRANSPARENT_HUGEPAGE** is required.) Memory used by shared memory (shmem) and *tmpfs*(5) allocated with huge pages.

ShmemPmdMapped %lu (since Linux 4.8)
(**CONFIG_TRANSPARENT_HUGEPAGE** is required.) Shared memory mapped into user space with huge pages.

CmaTotal %lu (since Linux 3.1)
Total CMA (Contiguous Memory Allocator) pages. (**CONFIG_CMA** is required.)

CmaFree %lu (since Linux 3.1)
Free CMA (Contiguous Memory Allocator) pages. (**CONFIG_CMA** is required.)

HugePages_Total %lu
(**CONFIG_HUGETLB_PAGE** is required.) The size of the pool of huge pages.

HugePages_Free %lu
(**CONFIG_HUGETLB_PAGE** is required.) The number of huge pages in the pool that are not yet allocated.

HugePages_Rsvd %lu (since Linux 2.6.17)
(**CONFIG_HUGETLB_PAGE** is required.) This is the number of huge pages for which a commitment to allocate from the pool has been made, but no allocation has yet been made. These reserved huge pages guarantee that an application will be able to allocate a huge page from the pool of huge pages at fault time.

HugePages_Surp %lu (since Linux 2.6.24)
(**CONFIG_HUGETLB_PAGE** is required.) This is the number of huge pages in the pool above the value in */proc/sys/vm/nr_hugepages*. The maximum number of surplus huge pages is controlled by */proc/sys/vm/nr_overcommit_hugepages*.

Hugepagesize %lu
(**CONFIG_HUGETLB_PAGE** is required.) The size of huge pages.

DirectMap4k %lu (since Linux 2.6.27)
Number of bytes of RAM linearly mapped by kernel in 4 kB pages. (x86.)

DirectMap4M %lu (since Linux 2.6.27)
Number of bytes of RAM linearly mapped by kernel in 4 MB pages. (x86 with **CONFIG_X86_64** or **CONFIG_X86_PAE** enabled.)

DirectMap2M %lu (since Linux 2.6.27)
Number of bytes of RAM linearly mapped by kernel in 2 MB pages. (x86 with neither **CONFIG_X86_64** nor **CONFIG_X86_PAE** enabled.)

DirectMap1G %lu (since Linux 2.6.27)
(x86 with **CONFIG_X86_64** and **CONFIG_X86_DIRECT_GBPAGES** enabled.)

SEE ALSO

proc(5)

NAME

/proc/modules − loaded modules

DESCRIPTION

/proc/modules

A text list of the modules that have been loaded by the system. See also *lsmod*(8)

SEE ALSO

proc(5)

NAME

/proc/mtrr − memory type range registers

DESCRIPTION

/proc/mtrr

Memory Type Range Registers. See the Linux kernel source file *Documentation/x86/mtrr.rst* (or *Documentation/x86/mtrr.txt* before Linux 5.2, or *Documentation/mtrr.txt* before Linux 2.6.28) for details.

SEE ALSO

proc(5)

NAME

/proc/partitions − major and minor numbers of partitions

DESCRIPTION

/proc/partitions

Contains the major and minor numbers of each partition as well as the number of 1024-byte blocks and the partition name.

SEE ALSO

proc(5)

NAME

/proc/pci − PCI devices

DESCRIPTION

/proc/pci

This is a listing of all PCI devices found during kernel initialization and their configuration.

This file has been deprecated in favor of a new */proc* interface for PCI (*/proc/bus/pci*). It became optional in Linux 2.2 (available with **CONFIG_PCI_OLD_PROC** set at kernel compilation). It became once more nonoptionally enabled in Linux 2.4. Next, it was deprecated in Linux 2.6 (still available with **CONFIG_PCI_LEGACY_PROC** set), and finally removed altogether since Linux 2.6.17.

SEE ALSO

proc(5)

NAME

/proc/pid/, /proc/self/ − process information

DESCRIPTION

*/proc/*pid/

There is a numerical subdirectory for each running process; the subdirectory is named by the process ID. Each */proc/*pid subdirectory contains the pseudo-files and directories described below.

The files inside each */proc/*pid directory are normally owned by the effective user and effective group ID of the process. However, as a security measure, the ownership is made *root:root* if the process's "dumpable" attribute is set to a value other than 1.

Before Linux 4.11, *root:root* meant the "global" root user ID and group ID (i.e., UID 0 and GID 0 in the initial user namespace). Since Linux 4.11, if the process is in a noninitial user namespace that has a valid mapping for user (group) ID 0 inside the namespace, then the user (group) ownership of the files under */proc/*pid is instead made the same as the root user (group) ID of the namespace. This means that inside a container, things work as expected for the container "root" user.

The process's "dumpable" attribute may change for the following reasons:

- The attribute was explicitly set via the *prctl*(2) **PR_SET_DUMPABLE** operation.
- The attribute was reset to the value in the file */proc/sys/fs/suid_dumpable* (described below), for the reasons described in *prctl*(2).

Resetting the "dumpable" attribute to 1 reverts the ownership of the */proc/*pid/* files to the process's effective UID and GID. Note, however, that if the effective UID or GID is subsequently modified, then the "dumpable" attribute may be reset, as described in *prctl*(2). Therefore, it may be desirable to reset the "dumpable" attribute *after* making any desired changes to the process's effective UID or GID.

/proc/self/

This directory refers to the process accessing the */proc* filesystem, and is identical to the */proc* directory named by the process ID of the same process.

SEE ALSO

proc(5)

NAME

/proc/pid/attr/ − security-related attributes

DESCRIPTION

*/proc/*pid*/attr/*

The files in this directory provide an API for security modules. The contents of this directory are files that can be read and written in order to set security-related attributes. This directory was added to support SELinux, but the intention was that the API be general enough to support other security modules. For the purpose of explanation, examples of how SELinux uses these files are provided below.

This directory is present only if the kernel was configured with **CONFIG_SECURITY**.

*/proc/*pid*/attr/current* (since Linux 2.6.0)

The contents of this file represent the current security attributes of the process.

In SELinux, this file is used to get the security context of a process. Prior to Linux 2.6.11, this file could not be used to set the security context (a write was always denied), since SELinux limited process security transitions to *execve*(2) (see the description of */proc/*pid*/attr/exec*, below). Since Linux 2.6.11, SELinux lifted this restriction and began supporting "set" operations via writes to this node if authorized by policy, although use of this operation is only suitable for applications that are trusted to maintain any desired separation between the old and new security contexts.

Prior to Linux 2.6.28, SELinux did not allow threads within a multithreaded process to set their security context via this node as it would yield an inconsistency among the security contexts of the threads sharing the same memory space. Since Linux 2.6.28, SELinux lifted this restriction and began supporting "set" operations for threads within a multithreaded process if the new security context is bounded by the old security context, where the bounded relation is defined in policy and guarantees that the new security context has a subset of the permissions of the old security context.

Other security modules may choose to support "set" operations via writes to this node.

*/proc/*pid*/attr/exec* (since Linux 2.6.0)

This file represents the attributes to assign to the process upon a subsequent *execve*(2).

In SELinux, this is needed to support role/domain transitions, and *execve*(2) is the preferred point to make such transitions because it offers better control over the initialization of the process in the new security label and the inheritance of state. In SELinux, this attribute is reset on *execve*(2) so that the new program reverts to the default behavior for any *execve*(2) calls that it may make. In SELinux, a process can set only its own */proc/*pid*/attr/exec* attribute.

*/proc/*pid*/attr/fscreate* (since Linux 2.6.0)

This file represents the attributes to assign to files created by subsequent calls to *open*(2), *mkdir*(2), *symlink*(2), and *mknod*(2)

SELinux employs this file to support creation of a file (using the aforementioned system calls) in a secure state, so that there is no risk of inappropriate access being obtained between the time of creation and the time that attributes are set. In SELinux, this attribute is reset on *execve*(2), so that the new program reverts to the default behavior for any file creation calls it may make, but the attribute will persist across multiple file creation calls within a program unless it is explicitly reset. In SELinux, a process can set only its own */proc/*pid*/attr/fscreate* attribute.

*/proc/*pid*/attr/keycreate* (since Linux 2.6.18)

If a process writes a security context into this file, all subsequently created keys (**add_key**(2)) will be labeled with this context. For further information, see the kernel source file *Documentation/security/keys/core.rst* (or file *Documentation/security/keys.txt* between Linux 3.0 and Linux 4.13, or *Documentation/keys.txt* before Linux 3.0).

*/proc/*pid*/attr/prev* (since Linux 2.6.0)

This file contains the security context of the process before the last *execve*(2); that is, the previous value of */proc/*pid*/attr/current*.

*/proc/*pid*/attr/socketcreate* (since Linux 2.6.18)
If a process writes a security context into this file, all subsequently created sockets will be labeled with this context.

SEE ALSO

proc(5)

NAME

proc_pid_autogroup − group tasks for the scheduler

DESCRIPTION

*/proc/*pid*/autogroup* (since Linux 2.6.38)

See *sched*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/auxv − exec(3) information

DESCRIPTION

*/proc/*pid*/auxv* (since Linux 2.6.0)

This contains the contents of the ELF interpreter information passed to the process at exec time. The format is one *unsigned long* ID plus one *unsigned long* value for each entry. The last entry contains two zeros. See also *getauxval*(3).

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/cgroup − control group

DESCRIPTION

*/proc/*pid*/cgroup* (since Linux 2.6.24)
See *cgroups*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/clear_refs − reset the PG_Referenced and ACCESSED/YOUNG bits

DESCRIPTION

*/proc/*pid*/clear_refs* (since Linux 2.6.22)

This is a write-only file, writable only by owner of the process.

The following values may be written to the file:

1 (since Linux 2.6.22)
: Reset the PG_Referenced and ACCESSED/YOUNG bits for all the pages associated with the process. (Before Linux 2.6.32, writing any nonzero value to this file had this effect.)

2 (since Linux 2.6.32)
: Reset the PG_Referenced and ACCESSED/YOUNG bits for all anonymous pages associated with the process.

3 (since Linux 2.6.32)
: Reset the PG_Referenced and ACCESSED/YOUNG bits for all file-mapped pages associated with the process.

Clearing the PG_Referenced and ACCESSED/YOUNG bits provides a method to measure approximately how much memory a process is using. One first inspects the values in the "Referenced" fields for the VMAs shown in */proc/*pid*/smaps* to get an idea of the memory footprint of the process. One then clears the PG_Referenced and ACCESSED/YOUNG bits and, after some measured time interval, once again inspects the values in the "Referenced" fields to get an idea of the change in memory footprint of the process during the measured interval. If one is interested only in inspecting the selected mapping types, then the value 2 or 3 can be used instead of 1.

Further values can be written to affect different properties:

4 (since Linux 3.11)
: Clear the soft-dirty bit for all the pages associated with the process. This is used (in conjunction with */proc/*pid*/pagemap*) by the check-point restore system to discover which pages of a process have been dirtied since the file */proc/*pid*/clear_refs* was written to.

5 (since Linux 4.0)
: Reset the peak resident set size ("high water mark") to the process's current resident set size value.

Writing any value to */proc/*pid*/clear_refs* other than those listed above has no effect.

The */proc/*pid*/clear_refs* file is present only if the **CONFIG_PROC_PAGE_MONITOR** kernel configuration option is enabled.

SEE ALSO

proc(5)

NAME

/proc/pid/cmdline − command line

DESCRIPTION

*/proc/*pid*/cmdline*

This read-only file holds the complete command line for the process, unless the process is a zombie. In the latter case, there is nothing in this file: that is, a read on this file will return 0 characters.

For processes which are still running, the command-line arguments appear in this file in the same layout as they do in process memory: If the process is well-behaved, it is a set of strings separated by null bytes ('\0'), with a further null byte after the last string.

This is the common case, but processes have the freedom to override the memory region and break assumptions about the contents or format of the */proc/*pid*/cmdline* file.

If, after an *execve*(2), the process modifies its *argv* strings, those changes will show up here. This is not the same thing as modifying the *argv* array.

Furthermore, a process may change the memory location that this file refers via *prctl*(2) operations such as **PR_SET_MM_ARG_START**.

Think of this file as the command line that the process wants you to see.

SEE ALSO

proc(5)

NAME

/proc/pid/comm − command name

DESCRIPTION

*/proc/*pid*/comm* (since Linux 2.6.33)

This file exposes the process's *comm* value—that is, the command name associated with the process. Different threads in the same process may have different *comm* values, accessible via */proc/*pid*/task/*tid*/comm*. A thread may modify its *comm* value, or that of any of other thread in the same thread group (see the discussion of **CLONE_THREAD** in *clone*(2)), by writing to the file */proc/self/task/*tid*/comm*. Strings longer than **TASK_COMM_LEN** (16) characters (including the terminating null byte) are silently truncated.

This file provides a superset of the *prctl*(2) **PR_SET_NAME** and **PR_GET_NAME** operations, and is employed by *pthread_setname_np*(3) when used to rename threads other than the caller. The value in this file is used for the *%e* specifier in */proc/sys/kernel/core_pattern*; see *core*(5).

SEE ALSO

proc(5)

NAME

/proc/pid/coredump_filter − core dump filter

DESCRIPTION

*/proc/*pid*/coredump_filter* (since Linux 2.6.23)

See *core*(5).

SEE ALSO

proc(5)

NAME

/proc/pid/cpuset − CPU affinity sets

DESCRIPTION

*/proc/*pid*/cpuset* (since Linux 2.6.12)

See *cpuset*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/cwd − symbolic link to current working directory

DESCRIPTION

*/proc/*pid*/cwd*

This is a symbolic link to the current working directory of the process. To find out the current working directory of process 20, for instance, you can do this:

```
$ cd /proc/20/cwd; pwd -P
```

In a multithreaded process, the contents of this symbolic link are not available if the main thread has already terminated (typically by calling *pthread_exit*(3)).

Permission to dereference or read (**readlink**(2)) this symbolic link is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/environ − initial environment

DESCRIPTION

*/proc/*pid*/environ*

This file contains the initial environment that was set when the currently executing program was started via *execve*(2). The entries are separated by null bytes ('\0'), and there may be a null byte at the end. Thus, to print out the environment of process 1, you would do:

```
$ cat /proc/1/environ | tr '\000' '\n'
```

If, after an *execve*(2), the process modifies its environment (e.g., by calling functions such as *putenv*(3) or modifying the *environ*(7) variable directly), this file will *not* reflect those changes.

Furthermore, a process may change the memory location that this file refers via *prctl*(2) operations such as **PR_SET_MM_ENV_START**.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/exe − symbolic link to program pathname

DESCRIPTION

*/proc/*pid*/exe*

Under Linux 2.2 and later, this file is a symbolic link containing the actual pathname of the executed command. This symbolic link can be dereferenced normally; attempting to open it will open the executable. You can even type */proc/*pid*/exe* to run another copy of the same executable that is being run by process *pid*. If the pathname has been unlinked, the symbolic link will contain the string ' (deleted)' appended to the original pathname. In a multithreaded process, the contents of this symbolic link are not available if the main thread has already terminated (typically by calling *pthread_exit*(3)).

Permission to dereference or read (**readlink**(2)) this symbolic link is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

Under Linux 2.0 and earlier, */proc/*pid*/exe* is a pointer to the binary which was executed, and appears as a symbolic link. A *readlink*(2) call on this file under Linux 2.0 returns a string in the format:

```
[device]:inode
```

For example, [0301]:1502 would be inode 1502 on device major 03 (IDE, MFM, etc. drives) minor 01 (first partition on the first drive).

find(1) with the *−inum* option can be used to locate the file.

SEE ALSO

proc(5)

NAME

/proc/pid/fd/ − file descriptors

DESCRIPTION

*/proc/*pid*/fd/*

This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file. Thus, 0 is standard input, 1 standard output, 2 standard error, and so on.

For file descriptors for pipes and sockets, the entries will be symbolic links whose content is the file type with the inode. A *readlink*(2) call on this file returns a string in the format:

```
type:[inode]
```

For example, *socket:[2248868]* will be a socket and its inode is 2248868. For sockets, that inode can be used to find more information in one of the files under */proc/net/*.

For file descriptors that have no corresponding inode (e.g., file descriptors produced by *bpf*(2), *epoll_create*(2), *eventfd*(2), *inotify_init*(2), *perf_event_open*(2), *signalfd*(2), *timerfd_create*(2), and *userfaultfd*(2)), the entry will be a symbolic link with contents of the form

```
anon_inode:file-type
```

In many cases (but not all), the *file-type* is surrounded by square brackets.

For example, an epoll file descriptor will have a symbolic link whose content is the string *anon_inode:[eventpoll]*.

In a multithreaded process, the contents of this directory are not available if the main thread has already terminated (typically by calling *pthread_exit*(3)).

Programs that take a filename as a command-line argument, but don't take input from standard input if no argument is supplied, and programs that write to a file named as a command-line argument, but don't send their output to standard output if no argument is supplied, can nevertheless be made to use standard input or standard output by using */proc/*pid*/fd* files as command-line arguments. For example, assuming that *−i* is the flag designating an input file and *−o* is the flag designating an output file:

```
$ foobar −i /proc/self/fd/0 −o /proc/self/fd/1 ...
```

and you have a working filter.

/proc/self/fd/N is approximately the same as */dev/fd/N* in some UNIX and UNIX-like systems. Most Linux MAKEDEV scripts symbolically link */dev/fd* to */proc/self/fd*, in fact.

Most systems provide symbolic links */dev/stdin*, */dev/stdout*, and */dev/stderr*, which respectively link to the files *0*, *1*, and *2* in */proc/self/fd*. Thus the example command above could be written as:

```
$ foobar −i /dev/stdin −o /dev/stdout ...
```

Permission to dereference or read (**readlink**(2)) the symbolic links in this directory is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

Note that for file descriptors referring to inodes (pipes and sockets, see above), those inodes still have permission bits and ownership information distinct from those of the */proc/*pid*/fd* entry, and that the owner may differ from the user and group IDs of the process. An unprivileged process may lack permissions to open them, as in this example:

```
$ echo test | sudo −u nobody cat
test
$ echo test | sudo −u nobody cat /proc/self/fd/0
cat: /proc/self/fd/0: Permission denied
```

File descriptor 0 refers to the pipe created by the shell and owned by that shell's user, which is not *nobody*, so **cat** does not have permission to create a new file descriptor to read from that inode, even though it can still read from its existing file descriptor 0.

SEE ALSO

proc(5)

NAME

/proc/pid/fdinfo/ − information about file descriptors

DESCRIPTION

*/proc/*pid*/fdinfo/* (since Linux 2.6.22)

This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor. The files in this directory are readable only by the owner of the process. The contents of each file can be read to obtain information about the corresponding file descriptor. The content depends on the type of file referred to by the corresponding file descriptor.

For regular files and directories, we see something like:

```
$ cat /proc/12015/fdinfo/4
pos:    1000
flags:  01002002
mnt_id: 21
```

The fields are as follows:

pos This is a decimal number showing the file offset.

flags This is an octal number that displays the file access mode and file status flags (see *open*(2)). If the close-on-exec file descriptor flag is set, then *flags* will also include the value **O_CLOEXEC**.

Before Linux 3.1, this field incorrectly displayed the setting of **O_CLOEXEC** at the time the file was opened, rather than the current setting of the close-on-exec flag.

mnt_id This field, present since Linux 3.15, is the ID of the mount containing this file. See the description of */proc/*pid*/mountinfo*.

For eventfd file descriptors (see *eventfd*(2)), we see (since Linux 3.8) the following fields:

```
pos:  0
flags:      02
mnt_id:     10
eventfd-count:               40
```

eventfd-count is the current value of the eventfd counter, in hexadecimal.

For epoll file descriptors (see *epoll*(7)), we see (since Linux 3.8) the following fields:

```
pos:  0
flags:      02
mnt_id:     10
tfd:        9 events:       19 data: 74253d2500000009
tfd:        7 events:       19 data: 74253d2500000007
```

Each of the lines beginning *tfd* describes one of the file descriptors being monitored via the epoll file descriptor (see *epoll_ctl*(2) for some details). The *tfd* field is the number of the file descriptor. The *events* field is a hexadecimal mask of the events being monitored for this file descriptor. The *data* field is the data value associated with this file descriptor.

For signalfd file descriptors (see *signalfd*(2)), we see (since Linux 3.8) the following fields:

```
pos:  0
flags:      02
mnt_id:     10
sigmask:    0000000000000006
```

sigmask is the hexadecimal mask of signals that are accepted via this signalfd file descriptor. (In this example, bits 2 and 3 are set, corresponding to the signals **SIGINT** and **SIGQUIT**; see *signal*(7).)

For inotify file descriptors (see *inotify*(7)), we see (since Linux 3.8) the following fields:

```
pos:  0
flags:      00
mnt_id:     11
```

```
inotify wd:2 ino:7ef82a sdev:800001 mask:800afff ignored_mask:0 fhandle-byt
inotify wd:1 ino:192627 sdev:800001 mask:800afff ignored_mask:0 fhandle-byt
```

Each of the lines beginning with "inotify" displays information about one file or directory that is being monitored. The fields in this line are as follows:

wd A watch descriptor number (in decimal).

ino The inode number of the target file (in hexadecimal).

sdev The ID of the device where the target file resides (in hexadecimal).

mask The mask of events being monitored for the target file (in hexadecimal).

If the kernel was built with exportfs support, the path to the target file is exposed as a file handle, via three hexadecimal fields: *fhandle-bytes*, *fhandle-type*, and *f_handle*.

For fanotify file descriptors (see *fanotify*(7)), we see (since Linux 3.8) the following fields:

```
pos:  0
flags:      02
mnt_id:     11
fanotify flags:0 event-flags:88002
fanotify ino:19264f sdev:800001 mflags:0 mask:1 ignored_mask:0 fhandle-byte
```

The fourth line displays information defined when the fanotify group was created via *fanotify_init*(2):

flags The *flags* argument given to *fanotify_init*(2) (expressed in hexadecimal).

event-flags
The *event_f_flags* argument given to *fanotify_init*(2) (expressed in hexadecimal).

Each additional line shown in the file contains information about one of the marks in the fanotify group. Most of these fields are as for inotify, except:

mflags The flags associated with the mark (expressed in hexadecimal).

mask The events mask for this mark (expressed in hexadecimal).

ignored_mask
The mask of events that are ignored for this mark (expressed in hexadecimal).

For details on these fields, see *fanotify_mark*(2).

For timerfd file descriptors (see *timerfd*(2)), we see (since Linux 3.17) the following fields:

```
pos:    0
flags:  02004002
mnt_id: 13
clockid: 0
ticks: 0
settime flags: 03
it_value: (7695568592, 640020877)
it_interval: (0, 0)
```

clockid This is the numeric value of the clock ID (corresponding to one of the **CLOCK_*** constants defined via <*time.h*>) that is used to mark the progress of the timer (in this example, 0 is **CLOCK_REALTIME**).

ticks This is the number of timer expirations that have occurred, (i.e., the value that *read*(2) on it would return).

settime flags
This field lists the flags with which the timerfd was last armed (see *timerfd_settime*(2)), in octal (in this example, both **TFD_TIMER_ABSTIME** and **TFD_TIMER_CANCEL_ON_SET** are set).

it_value
This field contains the amount of time until the timer will next expire, expressed in seconds and nanoseconds. This is always expressed as a relative value, regardless of whether the timer was created using the **TFD_TIMER_ABSTIME** flag.

it_interval
This field contains the interval of the timer, in seconds and nanoseconds. (The *it_value* and *it_interval* fields contain the values that *timerfd_gettime*(2) on this file descriptor would return.)

SEE ALSO

proc(5)

NAME

/proc/pid/io − I/O statistics

DESCRIPTION

*/proc/*pid*/io* (since Linux 2.6.20)

This file contains I/O statistics for the process and its waited-for children, for example:

```
# cat /proc/3828/io
rchar: 323934931
wchar: 323929600
syscr: 632687
syscw: 632675
read_bytes: 0
write_bytes: 323932160
cancelled_write_bytes: 0
```

The fields are as follows:

rchar: characters read

The number of bytes returned by successful *read*(2) and similar system calls.

wchar: characters written

The number of bytes returned by successful *write*(2) and similar system calls.

syscr: read syscalls

The number of "file read" system calls—those from the *read*(2) family, *sendfile*(2), *copy_file_range*(2), and *ioctl*(2) **BTRFS_IOC_ENCODED_READ**[**_32**] (including when invoked by the kernel as part of other syscalls).

syscw: write syscalls

The number of "file write" system calls—those from the *write*(2) family, *sendfile*(2), *copy_file_range*(2), and *ioctl*(2) **BTRFS_IOC_ENCODED_WRITE**[**_32**] (including when invoked by the kernel as part of other syscalls).

read_bytes: bytes read

The number of bytes really fetched from the storage layer. This is accurate for block-backed filesystems.

write_bytes: bytes written

The number of bytes really sent to the storage layer.

cancelled_write_bytes:

The above statistics fail to account for truncation: if a process writes 1 MB to a regular file and then removes it, said 1 MB will not be written, but *will* have nevertheless been accounted as a 1 MB write. This field represents the number of bytes "saved" from I/O writeback. This can yield to having done negative I/O if caches dirtied by another process are truncated. *cancelled_write_bytes* applies to I/O already accounted-for in *write_bytes*.

Permission to access this file is governed by *ptrace*(2) access mode **PTRACE_MODE_READ_FSCREDS**.

CAVEATS

These counters are not atomic: on systems where 64-bit integer operations may tear, a counter could be updated simultaneously with a read, yielding an incorrect intermediate value.

SEE ALSO

getrusage(2), *proc*(5)

NAME

/proc/pid/limits − resource limits

DESCRIPTION

*/proc/*pid*/limits* (since Linux 2.6.24)

This file displays the soft limit, hard limit, and units of measurement for each of the process's resource limits (see *getrlimit*(2)). Up to and including Linux 2.6.35, this file is protected to allow reading only by the real UID of the process. Since Linux 2.6.36, this file is readable by all users on the system.

SEE ALSO

proc(5)

NAME

/proc/pid/map_files/ − memory-mapped files

DESCRIPTION

*/proc/*pid*/map_files/* (since Linux 3.3)

This subdirectory contains entries corresponding to memory-mapped files (see *mmap*(2)). Entries are named by memory region start and end address pair (expressed as hexadecimal numbers), and are symbolic links to the mapped files themselves. Here is an example, with the output wrapped and reformatted to fit on an 80-column display:

```
# ls -l /proc/self/map_files/
lr--------. 1 root root 64 Apr 16 21:31
            3252e00000-3252e20000 -> /usr/lib64/ld-2.15.so
...
```

Although these entries are present for memory regions that were mapped with the **MAP_FILE** flag, the way anonymous shared memory (regions created with the **MAP_ANON | MAP_SHARED** flags) is implemented in Linux means that such regions also appear on this directory. Here is an example where the target file is the deleted */dev/zero* one:

```
lrw-------. 1 root root 64 Apr 16 21:33
            7fc075d2f000-7fc075e6f000 -> /dev/zero (deleted)
```

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

Until Linux 4.3, this directory appeared only if the **CONFIG_CHECKPOINT_RESTORE** kernel configuration option was enabled.

Capabilities are required to read the contents of the symbolic links in this directory: before Linux 5.9, the reading process requires **CAP_SYS_ADMIN** in the initial user namespace; since Linux 5.9, the reading process must have either **CAP_SYS_ADMIN** or **CAP_CHECKPOINT_RESTORE** in the initial (i.e. root) user namespace.

SEE ALSO

proc(5)

NAME

/proc/pid/maps − mapped memory regions

DESCRIPTION

*/proc/*pid*/maps*

A file containing the currently mapped memory regions and their access permissions. See *mmap*(2) for some further information about memory mappings.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

The format of the file is:

```
address           perms offset  dev   inode       pathname
00400000-00452000 r-xp 00000000 08:02 173521      /usr/bin/dbus-daemon
00651000-00652000 r--p 00051000 08:02 173521      /usr/bin/dbus-daemon
00652000-00655000 rw-p 00052000 08:02 173521      /usr/bin/dbus-daemon
00e03000-00e24000 rw-p 00000000 00:00 0           [heap]
00e24000-011f7000 rw-p 00000000 00:00 0           [heap]
...
35b1800000-35b1820000 r-xp 00000000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000 r--p 0001f000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a20000-35b1a21000 rw-p 00020000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a21000-35b1a22000 rw-p 00000000 00:00 0
35b1c00000-35b1dac000 r-xp 00000000 08:02 135870  /usr/lib64/libc-2.15.so
35b1dac000-35b1fac000 ---p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fac000-35b1fb0000 r--p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fb0000-35b1fb2000 rw-p 001b0000 08:02 135870  /usr/lib64/libc-2.15.so
...
f2c6ff8c000-7f2c7078c000 rw-p 00000000 00:00 0    [stack:986]
...
7fffb2c0d000-7fffb2c2e000 rw-p 00000000 00:00 0   [stack]
7fffb2d48000-7fffb2d49000 r-xp 00000000 00:00 0   [vdso]
```

The *address* field is the address space in the process that the mapping occupies. The *perms* field is a set of permissions:

```
r = read
w = write
x = execute
s = shared
p = private (copy on write)
```

The *offset* field is the offset into the file/whatever; *dev* is the device (major:minor); *inode* is the inode on that device. 0 indicates that no inode is associated with the memory region, as would be the case with BSS (uninitialized data).

The *pathname* field will usually be the file that is backing the mapping. For ELF files, you can easily coordinate with the *offset* field by looking at the Offset field in the ELF program headers (*readelf −l*).

There are additional helpful pseudo-paths:

[stack] The initial process's (also known as the main thread's) stack.

*[stack:*tid*]* (from Linux 3.4 to Linux 4.4)
A thread's stack (where the *tid* is a thread ID). It corresponds to the */proc/*pid*/task/*tid*/* path. This field was removed in Linux 4.5, since providing this information for a process with large numbers of threads is expensive.

[vdso] The virtual dynamically linked shared object. See *vdso*(7).

[heap] The process's heap.

*[anon:*name*]* (since Linux 5.17)
A named private anonymous mapping. Set with *prctl*(2) **PR_SET_VMA_ANON_NAME**.

*[anon_shmem:*name*]* (since Linux 6.2)
A named shared anonymous mapping. Set with *prctl*(2) **PR_SET_VMA_ANON_NAME**.

If the *pathname* field is blank, this is an anonymous mapping as obtained via *mmap*(2). There is no easy way to coordinate this back to a process's source, short of running it through *gdb*(1), *strace*(1), or similar.

pathname is shown unescaped except for newline characters, which are replaced with an octal escape sequence. As a result, it is not possible to determine whether the original pathname contained a newline character or the literal *\012* character sequence.

If the mapping is file-backed and the file has been deleted, the string " (deleted)" is appended to the pathname. Note that this is ambiguous too.

Under Linux 2.0, there is no field giving pathname.

SEE ALSO

proc(5)

NAME

/proc/pid/mem − memory

DESCRIPTION

*/proc/*pid*/mem*

This file can be used to access the pages of a process's memory through *open*(2), *read*(2), and *lseek*(2).

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/mountinfo − mount information

DESCRIPTION

*/proc/*pid*/mountinfo* (since Linux 2.6.26)

This file contains information about mounts in the process's mount namespace (see *mount_namespaces*(7)). It supplies various information (e.g., propagation state, root of mount for bind mounts, identifier for each mount and its parent) that is missing from the (older) */proc/*pid*/mounts* file, and fixes various other problems with that file (e.g., nonextensibility, failure to distinguish per-mount versus per-superblock options).

The file contains lines of the form:

```
36 35 98:0 /mnt1 /mnt2 rw,noatime master:1 - ext3 /dev/root rw,errors=continue
(1)(2)(3)   (4)   (5)      (6)      (7)   (8) (9)   (10)         (11)
```

The numbers in parentheses are labels for the descriptions below:

(1) mount ID: a unique ID for the mount (may be reused after *umount*(2)).

(2) parent ID: the ID of the parent mount (or of self for the root of this mount namespace's mount tree).

If a new mount is stacked on top of a previous existing mount (so that it hides the existing mount) at pathname P, then the parent of the new mount is the previous mount at that location. Thus, when looking at all the mounts stacked at a particular location, the top-most mount is the one that is not the parent of any other mount at the same location. (Note, however, that this top-most mount will be accessible only if the longest path subprefix of P that is a mount point is not itself hidden by a stacked mount.)

If the parent mount lies outside the process's root directory (see *chroot*(2)), the ID shown here won't have a corresponding record in *mountinfo* whose mount ID (field 1) matches this parent mount ID (because mounts that lie outside the process's root directory are not shown in *mountinfo*). As a special case of this point, the process's root mount may have a parent mount (for the initramfs filesystem) that lies outside the process's root directory, and an entry for that mount will not appear in *mountinfo*.

(3) major:minor: the value of *st_dev* for files on this filesystem (see *stat*(2)).

(4) root: the pathname of the directory in the filesystem which forms the root of this mount.

(5) mount point: the pathname of the mount point relative to the process's root directory.

(6) mount options: per-mount options (see *mount*(2)).

(7) optional fields: zero or more fields of the form "tag[:value]"; see below.

(8) separator: the end of the optional fields is marked by a single hyphen.

(9) filesystem type: the filesystem type in the form "type[.subtype]".

(10) mount source: filesystem-specific information or "none".

(11) super options: per-superblock options (see *mount*(2)).

Currently, the possible optional fields are *shared*, *master*, *propagate_from*, and *unbindable*. See *mount_namespaces*(7) for a description of these fields. Parsers should ignore all unrecognized optional fields.

For more information on mount propagation see *Documentation/filesystems/sharedsubtree.rst* (or *Documentation/filesystems/sharedsubtree.txt* before Linux 5.8) in the Linux kernel source tree.

SEE ALSO

proc(5)

NAME

/proc/pid/mounts − mounted filesystems

DESCRIPTION

*/proc/*pid*/mounts* (since Linux 2.4.19)

This file lists all the filesystems currently mounted in the process's mount namespace (see *mount_namespaces*(7)). The format of this file is documented in *fstab*(5)

Since Linux 2.6.15, this file is pollable: after opening the file for reading, a change in this file (i.e., a filesystem mount or unmount) causes *select*(2) to mark the file descriptor as having an exceptional condition, and *poll*(2) and *epoll_wait*(2) mark the file as having a priority event (**POLLPRI**). (Before Linux 2.6.30, a change in this file was indicated by the file descriptor being marked as readable for *select*(2), and being marked as having an error condition for *poll*(2) and *epoll_wait*(2).)

/proc/mounts

Before Linux 2.4.19, this file was a list of all the filesystems currently mounted on the system. With the introduction of per-process mount namespaces in Linux 2.4.19 (see *mount_namespaces*(7)), this file became a link to */proc/self/mounts*, which lists the mounts of the process's own mount namespace. The format of this file is documented in *fstab*(5)

SEE ALSO

proc(5)

NAME

/proc/pid/mountstats − mount statistics

DESCRIPTION

*/proc/*pid*/mountstats* (since Linux 2.6.17)

This file exports information (statistics, configuration information) about the mounts in the process's mount namespace (see *mount_namespaces*(7)). Lines in this file have the form:

```
device /dev/sda7 mounted on /home with fstype ext3 [stats]
(       1      )            ( 2 )             (3 ) (  4  )
```

The fields in each line are:

(1) The name of the mounted device (or "nodevice" if there is no corresponding device).

(2) The mount point within the filesystem tree.

(3) The filesystem type.

(4) Optional statistics and configuration information. Currently (as at Linux 2.6.26), only NFS filesystems export information via this field.

This file is readable only by the owner of the process.

SEE ALSO

proc(5)

NAME

/proc/pid/net/, /proc/net/ − network layer information

DESCRIPTION

*/proc/*pid*/net/* (since Linux 2.6.25)
See the description of */proc/net*.

/proc/net/
This directory contains various files and subdirectories containing information about the networking layer. The files contain ASCII structures and are, therefore, readable with *cat*(1)However, the standard *netstat*(8) suite provides much cleaner access to these files.

With the advent of network namespaces, various information relating to the network stack is virtualized (see *network_namespaces*(7)). Thus, since Linux 2.6.25, */proc/net* is a symbolic link to the directory */proc/self/net*, which contains the same files and directories as listed below. However, these files and directories now expose information for the network namespace of which the process is a member.

/proc/net/arp
This holds an ASCII readable dump of the kernel ARP table used for address resolutions. It will show both dynamically learned and preprogrammed ARP entries. The format is:

```
IP address      HW type   Flags      HW address           Mask   Device
192.168.0.50    0x1       0x2        00:50:BF:25:68:F3    *      eth0
192.168.0.250   0x1       0xc        00:00:00:00:00:00    *      eth0
```

Here "IP address" is the IPv4 address of the machine and the "HW type" is the hardware type of the address from RFC 826. The flags are the internal flags of the ARP structure (as defined in */usr/include/linux/if_arp.h*) and the "HW address" is the data link layer mapping for that IP address if it is known.

/proc/net/dev
The dev pseudo-file contains network device status information. This gives the number of received and sent packets, the number of errors and collisions and other basic statistics. These are used by the *ifconfig*(8) program to report device status. The format is:

```
Inter−|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    pac
    lo: 2776770   11307    0    0    0     0          0         0  2776770   1
  eth0: 1215645    2751    0    0    0     0          0         0  1782404
  ppp0: 1622270    5552    1    0    0     0          0         0   354130
  tap0:    7714      81    0    0    0     0          0         0     7714
```

/proc/net/dev_mcast
Defined in */usr/src/linux/net/core/dev_mcast.c*:

```
indx interface_name  dmi_u dmi_g dmi_address
2    eth0            1     0     01005e000001
3    eth1            1     0     01005e000001
4    eth2            1     0     01005e000001
```

/proc/net/igmp
Internet Group Management Protocol. Defined in */usr/src/linux/net/core/igmp.c*.

/proc/net/rarp
This file uses the same format as the *arp* file and contains the current reverse mapping database used to provide *rarp*(8) reverse address lookup services. If RARP is not configured into the kernel, this file will not be present.

/proc/net/raw
Holds a dump of the RAW socket table. Much of the information is not of use apart from debugging. The "sl" value is the kernel hash slot for the socket, the "local_address" is the local address and protocol number pair. "St" is the internal status of the socket. The "tx_queue" and "rx_queue" are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm−>when", and "rexmits" fields are not used by RAW. The "uid" field holds the effective UID of the creator of the socket.

/proc/net/snmp
This file holds the ASCII data needed for the IP, ICMP, TCP, and UDP management information bases for an SNMP agent.

/proc/net/tcp
Holds a dump of the TCP socket table. Much of the information is not of use apart from debugging. The "sl" value is the kernel hash slot for the socket, the "local_address" is the local address and port number pair. The "rem_address" is the remote address and port number pair (if connected). "St" is the internal status of the socket. The "tx_queue" and "rx_queue" are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm−>when", and "rexmits" fields hold internal information of the kernel socket state and are useful only for debugging. The "uid" field holds the effective UID of the creator of the socket.

/proc/net/udp
Holds a dump of the UDP socket table. Much of the information is not of use apart from debugging. The "sl" value is the kernel hash slot for the socket, the "local_address" is the local address and port number pair. The "rem_address" is the remote address and port number pair (if connected). "St" is the internal status of the socket. The "tx_queue" and "rx_queue" are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm−>when", and "rexmits" fields are not used by UDP. The "uid" field holds the effective UID of the creator of the socket. The format is:

```
sl  local_address rem_address   st tx_queue rx_queue tr rexmits  tm->when uid
 1: 01642C89:0201 0C642C89:03FF 01 00000000:00000001 01:000071BA 00000000 0
 1: 00000000:0801 00000000:0000 0A 00000000:00000000 00:00000000 6F000100 0
 1: 00000000:0201 00000000:0000 0A 00000000:00000000 00:00000000 00000000 0
```

/proc/net/unix
Lists the UNIX domain sockets present within the system and their status. The format is:

```
Num RefCount Protocol Flags    Type St Inode Path
 0: 00000002 00000000 00000000 0001 03    42
 1: 00000001 00000000 00010000 0001 01  1948 /dev/printer
```

The fields are as follows:

Num: the kernel table slot number.

RefCount: the number of users of the socket.

Protocol: currently always 0.

Flags: the internal kernel flags holding the status of the socket.

Type: the socket type. For **SOCK_STREAM** sockets, this is 0001; for **SOCK_DGRAM** sockets, it is 0002; and for **SOCK_SEQPACKET** sockets, it is 0005.

St: the internal state of the socket.

Inode: the inode number of the socket.

Path: the bound pathname (if any) of the socket. Sockets in the abstract namespace are included in the list, and are shown with a *Path* that commences with the character '@'.

/proc/net/netfilter/nfnetlink_queue
This file contains information about netfilter user-space queueing, if used. Each line represents a queue. Queues that have not been subscribed to by user space are not shown.

```
  1   4207     0  2 65535     0     0        0  1
(1)   (2)    (3)(4)  (5)    (6)   (7)      (8)
```

The fields in each line are:

(1) The ID of the queue. This matches what is specified in the **−−queue−num** or **−−queue−balance** options to the *iptables*(8) NFQUEUE target. See *iptables-extensions*(8) for more information.

(2) The netlink port ID subscribed to the queue.

(3) The number of packets currently queued and waiting to be processed by the application.

(4) The copy mode of the queue. It is either 1 (metadata only) or 2 (also copy payload data to user space).

(5) Copy range; that is, how many bytes of packet payload should be copied to user space at most.

(6) queue dropped. Number of packets that had to be dropped by the kernel because too many packets are already waiting for user space to send back the mandatory accept/drop verdicts.

(7) queue user dropped. Number of packets that were dropped within the netlink subsystem. Such drops usually happen when the corresponding socket buffer is full; that is, user space is not able to read messages fast enough.

(8) sequence number. Every queued packet is associated with a (32-bit) monotonically increasing sequence number. This shows the ID of the most recent packet queued.

The last number exists only for compatibility reasons and is always 1.

SEE ALSO

proc(5)

NAME

/proc/pid/ns/ − namespaces

DESCRIPTION

*/proc/*pid*/ns/* (since Linux 3.0)

This is a subdirectory containing one entry for each namespace that supports being manipulated by *setns*(2). For more information, see *namespaces*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/numa_maps − NUMA memory policy and allocation

DESCRIPTION

*/proc/*pid*/numa_maps* (since Linux 2.6.14)

See *numa*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/oom_score − OOM-killer score

DESCRIPTION

*/proc/*pid*/oom_score* (since Linux 2.6.11)

This file displays the current score that the kernel gives to this process for the purpose of selecting a process for the OOM-killer. A higher score means that the process is more likely to be selected by the OOM-killer. The basis for this score is the amount of memory used by the process, with increases (+) or decreases (−) for factors including:

- whether the process is privileged (−).

Before Linux 2.6.36 the following factors were also used in the calculation of oom_score:

- whether the process creates a lot of children using *fork*(2) (+);
- whether the process has been running a long time, or has used a lot of CPU time (−);
- whether the process has a low nice value (i.e., > 0) (+); and
- whether the process is making direct hardware access (−).

The *oom_score* also reflects the adjustment specified by the *oom_score_adj* or *oom_adj* setting for the process.

SEE ALSO

proc(5), *proc_pid_oom_score_adj*(5)

NAME

/proc/pid/oom_score_adj − OOM-killer score adjustment

DESCRIPTION

*/proc/*pid*/oom_score_adj* (since Linux 2.6.36)

This file can be used to adjust the badness heuristic used to select which process gets killed in out-of-memory conditions.

The badness heuristic assigns a value to each candidate task ranging from 0 (never kill) to 1000 (always kill) to determine which process is targeted. The units are roughly a proportion along that range of allowed memory the process may allocate from, based on an estimation of its current memory and swap use. For example, if a task is using all allowed memory, its badness score will be 1000. If it is using half of its allowed memory, its score will be 500.

There is an additional factor included in the badness score: root processes are given 3% extra memory over other tasks.

The amount of "allowed" memory depends on the context in which the OOM-killer was called. If it is due to the memory assigned to the allocating task's cpuset being exhausted, the allowed memory represents the set of mems assigned to that cpuset (see *cpuset*(7)). If it is due to a mempolicy's node(s) being exhausted, the allowed memory represents the set of mempolicy nodes. If it is due to a memory limit (or swap limit) being reached, the allowed memory is that configured limit. Finally, if it is due to the entire system being out of memory, the allowed memory represents all allocatable resources.

The value of *oom_score_adj* is added to the badness score before it is used to determine which task to kill. Acceptable values range from −1000 (OOM_SCORE_ADJ_MIN) to +1000 (OOM_SCORE_ADJ_MAX). This allows user space to control the preference for OOM-killing, ranging from always preferring a certain task or completely disabling it from OOM-killing. The lowest possible value, −1000, is equivalent to disabling OOM-killing entirely for that task, since it will always report a badness score of 0.

Consequently, it is very simple for user space to define the amount of memory to consider for each task. Setting an *oom_score_adj* value of +500, for example, is roughly equivalent to allowing the remainder of tasks sharing the same system, cpuset, mempolicy, or memory controller resources to use at least 50% more memory. A value of −500, on the other hand, would be roughly equivalent to discounting 50% of the task's allowed memory from being considered as scoring against the task.

For backward compatibility with previous kernels, */proc/*pid*/oom_adj* can still be used to tune the badness score. Its value is scaled linearly with *oom_score_adj*.

Writing to */proc/*pid*/oom_score_adj* or */proc/*pid*/oom_adj* will change the other with its scaled value.

The *choom*(1) program provides a command-line interface for adjusting the *oom_score_adj* value of a running process or a newly executed command.

HISTORY

*/proc/*pid*/oom_adj* (since Linux 2.6.11)

This file can be used to adjust the score used to select which process should be killed in an out-of-memory (OOM) situation. The kernel uses this value for a bit-shift operation of the process's *oom_score* value: valid values are in the range −16 to +15, plus the special value −17, which disables OOM-killing altogether for this process. A positive score increases the likelihood of this process being killed by the OOM-killer; a negative score decreases the likelihood.

The default value for this file is 0; a new process inherits its parent's *oom_adj* setting. A process must be privileged (**CAP_SYS_RESOURCE**) to update this file, although a process can always increase its own *oom_adj* setting (since Linux 2.6.20).

Since Linux 2.6.36, use of this file is deprecated in favor of */proc/*pid*/oom_score_adj*, and finally removed in Linux 3.7.

SEE ALSO

proc(5), *proc_pid_oom_score*(5)

NAME

/proc/pid/pagemap − mapping of virtual pages

DESCRIPTION

*/proc/*pid*/pagemap* (since Linux 2.6.25)

This file shows the mapping of each of the process's virtual pages into physical page frames or swap area. It contains one 64-bit value for each virtual page, with the bits set as follows:

63 If set, the page is present in RAM.

62 If set, the page is in swap space

61 (since Linux 3.5)
The page is a file-mapped page or a shared anonymous page.

60–58 (since Linux 3.11)
Zero

57 (since Linux 5.14)
If set, the page is write-protected through *userfaultfd*(2).

56 (since Linux 4.2)
The page is exclusively mapped.

55 (since Linux 3.11)
PTE is soft-dirty (see the kernel source file *Documentation/admin−guide/mm/soft−dirty.rst*).

54–0 If the page is present in RAM (bit 63), then these bits provide the page frame number, which can be used to index */proc/kpageflags* and */proc/kpagecount*. If the page is present in swap (bit 62), then bits 4–0 give the swap type, and bits 54–5 encode the swap offset.

Before Linux 3.11, bits 60–55 were used to encode the base-2 log of the page size.

To employ */proc/*pid*/pagemap* efficiently, use */proc/*pid*/maps* to determine which areas of memory are actually mapped and seek to skip over unmapped regions.

The */proc/*pid*/pagemap* file is present only if the **CONFIG_PROC_PAGE_MONITOR** kernel configuration option is enabled.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/personality − execution domain

DESCRIPTION

*/proc/*pid*/personality* (since Linux 2.6.28)

This read-only file exposes the process's execution domain, as set by *personality*(2). The value is displayed in hexadecimal notation.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/projid_map − project ID mappings

DESCRIPTION

*/proc/*pid*/projid_map* (since Linux 3.7)

See *user_namespaces*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/root/ − symbolic link to root directory

DESCRIPTION

*/proc/*pid*/root/*

UNIX and Linux support the idea of a per-process root of the filesystem, set by the *chroot*(2) system call. This file is a symbolic link that points to the process's root directory, and behaves in the same way as *exe*, and *fd/**.

Note however that this file is not merely a symbolic link. It provides the same view of the filesystem (including namespaces and the set of per-process mounts) as the process itself. An example illustrates this point. In one terminal, we start a shell in new user and mount namespaces, and in that shell we create some new mounts:

```
$ PS1='sh1# ' unshare -Urnm
sh1# mount -t tmpfs tmpfs /etc  # Mount empty tmpfs at /etc
sh1# mount --bind /usr /dev     # Mount /usr at /dev
sh1# echo $$
27123
```

In a second terminal window, in the initial mount namespace, we look at the contents of the corresponding mounts in the initial and new namespaces:

```
$ PS1='sh2# ' sudo sh
sh2# ls /etc | wc -l                  # In initial NS
309
sh2# ls /proc/27123/root/etc | wc -l  # /etc in other NS
0                                     # The empty tmpfs dir
sh2# ls /dev | wc -l                  # In initial NS
205
sh2# ls /proc/27123/root/dev | wc -l  # /dev in other NS
11                                    # Actually bind
                                      # mounted to /usr
sh2# ls /usr | wc -l                  # /usr in initial NS
11
```

In a multithreaded process, the contents of the */proc/*pid*/root* symbolic link are not available if the main thread has already terminated (typically by calling *pthread_exit*(3)).

Permission to dereference or read (**readlink**(2)) this symbolic link is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/seccomp − secure computing mode

DESCRIPTION

*/proc/*pid*/seccomp* (Linux 2.6.12 to Linux 2.6.22)

This file can be used to read and change the process's secure computing (seccomp) mode setting. It contains the value 0 if the process is not in seccomp mode, and 1 if the process is in strict seccomp mode (see *seccomp*(2)). Writing 1 to this file places the process irreversibly in strict seccomp mode. (Further attempts to write to the file fail with the **EPERM** error.)

In Linux 2.6.23, this file went away, to be replaced by the *prctl*(2) **PR_GET_SECCOMP** and **PR_SET_SECCOMP** operations (and later by *seccomp*(2) and the *Seccomp* field in */proc/*pid*/status*).

SEE ALSO

proc(5)

NAME

/proc/pid/setgroups − allow or deny setting groups

DESCRIPTION

*/proc/*pid*/setgroups* (since Linux 3.19)

See *user_namespaces*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/smaps − XXX: What does ’s’ in "smaps" stand for?

DESCRIPTION

*/proc/*pid*/smaps* (since Linux 2.6.14)

This file shows memory consumption for each of the process’s mappings. (The *pmap*(1) command displays similar information, in a form that may be easier for parsing.) For each mapping there is a series of lines such as the following:

```
00400000-0048a000 r-xp 00000000 fd:03 960637       /bin/bash
Size:                552 kB
Rss:                 460 kB
Pss:                 100 kB
Shared_Clean:        452 kB
Shared_Dirty:          0 kB
Private_Clean:         8 kB
Private_Dirty:         0 kB
Referenced:          460 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
ProtectionKey:         0
VmFlags: rd ex mr mw me dw
```

The first of these lines shows the same information as is displayed for the mapping in */proc/*pid*/maps*. The following lines show the size of the mapping, the amount of the mapping that is currently resident in RAM ("Rss"), the process’s proportional share of this mapping ("Pss"), the number of clean and dirty shared pages in the mapping, and the number of clean and dirty private pages in the mapping. "Referenced" indicates the amount of memory currently marked as referenced or accessed. "Anonymous" shows the amount of memory that does not belong to any file. "Swap" shows how much would-be-anonymous memory is also used, but out on swap.

The "KernelPageSize" line (available since Linux 2.6.29) is the page size used by the kernel to back the virtual memory area. This matches the size used by the MMU in the majority of cases. However, one counter-example occurs on PPC64 kernels whereby a kernel using 64 kB as a base page size may still use 4 kB pages for the MMU on older processors. To distinguish the two attributes, the "MMUPageSize" line (also available since Linux 2.6.29) reports the page size used by the MMU.

The "Locked" indicates whether the mapping is locked in memory or not.

The "ProtectionKey" line (available since Linux 4.9, on x86 only) contains the memory protection key (see *pkeys*(7)) associated with the virtual memory area. This entry is present only if the kernel was built with the **CONFIG_X86_INTEL_MEMORY_PROTECTION_KEYS** configuration option (since Linux 4.6).

The "VmFlags" line (available since Linux 3.8) represents the kernel flags associated with the virtual memory area, encoded using the following two-letter codes:

rd	-	readable
wr	-	writable
ex	-	executable
sh	-	shared
mr	-	may read
mw	-	may write
me	-	may execute

ms	-	may share
gd	-	stack segment grows down
pf	-	pure PFN range
dw	-	disabled write to the mapped file
lo	-	pages are locked in memory
io	-	memory mapped I/O area
sr	-	sequential read advise provided
rr	-	random read advise provided
dc	-	do not copy area on fork
de	-	do not expand area on remapping
ac	-	area is accountable
nr	-	swap space is not reserved for the area
ht	-	area uses huge tlb pages
sf	-	perform synchronous page faults (since Linux 4.15)
nl	-	non-linear mapping (removed in Linux 4.0)
ar	-	architecture specific flag
wf	-	wipe on fork (since Linux 4.14)
dd	-	do not include area into core dump
sd	-	soft-dirty flag (since Linux 3.13)
mm	-	mixed map area
hg	-	huge page advise flag
nh	-	no-huge page advise flag
mg	-	mergeable advise flag
um	-	userfaultfd missing pages tracking (since Linux 4.3)
uw	-	userfaultfd wprotect pages tracking (since Linux 4.3)

The */proc/*pid*/smaps* file is present only if the **CONFIG_PROC_PAGE_MONITOR** kernel configuration option is enabled.

SEE ALSO

proc(5)

NAME

/proc/pid/stack − kernel stack

DESCRIPTION

*/proc/*pid*/stack* (since Linux 2.6.29)

This file provides a symbolic trace of the function calls in this process's kernel stack. This file is provided only if the kernel was built with the **CONFIG_STACKTRACE** configuration option.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/stat − status information

DESCRIPTION

*/proc/*pid*/stat*

Status information about the process. This is used by *ps*(1)It is defined in the kernel source file *fs/proc/array.c*.

The fields, in order, with their proper *scanf*(3) format specifiers, are listed below. Whether or not certain of these fields display valid information is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** | **PTRACE_MODE_NOAUDIT** check (refer to *ptrace*(2)). If the check denies access, then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

(1) *pid* %d
The process ID.

(2) *comm* %s
The filename of the executable, in parentheses. Strings longer than **TASK_COMM_LEN** (16) characters (including the terminating null byte) are silently truncated. This is visible whether or not the executable is swapped out.

(3) *state* %c
One of the following characters, indicating process state:

R	Running
S	Sleeping in an interruptible wait
D	Waiting in uninterruptible disk sleep
Z	Zombie
T	Stopped (on a signal) or (before Linux 2.6.33) trace stopped
t	Tracing stop (Linux 2.6.33 onward)
W	Paging (only before Linux 2.6.0)
X	Dead (from Linux 2.6.0 onward)
x	Dead (Linux 2.6.33 to 3.13 only)
K	Wakekill (Linux 2.6.33 to 3.13 only)
W	Waking (Linux 2.6.33 to 3.13 only)
P	Parked (Linux 3.9 to 3.13 only)
I	Idle (Linux 4.14 onward)

(4) *ppid* %d
The PID of the parent of this process.

(5) *pgrp* %d
The process group ID of the process.

(6) *session* %d
The session ID of the process.

(7) *tty_nr* %d
The controlling terminal of the process. (The minor device number is contained in the combination of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)

(8) *tpgid* %d
The ID of the foreground process group of the controlling terminal of the process.

(9) *flags* %u
The kernel flags word of the process. For bit meanings, see the PF_* defines in the Linux kernel source file *include/linux/sched.h*. Details depend on the kernel version.

The format for this field was %lu before Linux 2.6.

(10) *minflt* %lu
The number of minor faults the process has made which have not required loading a memory page from disk.

(11) *cminflt* %lu
The number of minor faults that the process's waited-for children have made.

(12) *majflt* %lu
The number of major faults the process has made which have required loading a memory page from disk.

(13) *cmajflt* %lu
The number of major faults that the process's waited-for children have made.

(14) *utime* %lu
Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*). This includes guest time, *guest_time* (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.

(15) *stime* %lu
Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*).

(16) *cutime* %ld
Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*). (See also *times*(2).) This includes guest time, *cguest_time* (time spent running a virtual CPU, see below).

(17) *cstime* %ld
Amount of time that this process's waited-for children have been scheduled in kernel mode, measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*).

(18) *priority* %ld
(Explanation for Linux 2.6) For processes running a real-time scheduling policy (*policy* below; see *sched_setscheduler*(2)), this is the negated scheduling priority, minus one; that is, a number in the range −2 to −100, corresponding to real-time priorities 1 to 99. For processes running under a non-real-time scheduling policy, this is the raw nice value (**setpriority**(2)) as represented in the kernel. The kernel stores nice values as numbers in the range 0 (high) to 39 (low), corresponding to the user-visible nice range of −20 to 19.

Before Linux 2.6, this was a scaled value based on the scheduler weighting given to this process.

(19) *nice* %ld
The nice value (see *setpriority*(2)), a value in the range 19 (low priority) to −20 (high priority).

(20) *num_threads* %ld
Number of threads in this process (since Linux 2.6). Before Linux 2.6, this field was hard coded to 0 as a placeholder for an earlier removed field.

(21) *itrealvalue* %ld
The time in jiffies before the next **SIGALRM** is sent to the process due to an interval timer. Since Linux 2.6.17, this field is no longer maintained, and is hard coded as 0.

(22) *starttime* %llu
The time the process started after system boot. Before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks (divide by *sysconf(_SC_CLK_TCK)*).

The format for this field was %lu before Linux 2.6.

(23) *vsize* %lu
Virtual memory size in bytes.

(24) *rss* %ld
Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out. This value is inaccurate; see */proc/*pid*/statm* below.

(25) *rsslim* %lu
Current soft limit in bytes on the rss of the process; see the description of **RLIMIT_RSS** in *getrlimit*(2).

(26) *startcode* %lu [PT]
The address above which program text can run.

(27) *endcode* %lu [PT]
The address below which program text can run.

(28) *startstack* %lu [PT]
The address of the start (i.e., bottom) of the stack.

(29) *kstkesp* %lu [PT]
The current value of ESP (stack pointer), as found in the kernel stack page for the process.

(30) *kstkeip* %lu [PT]
The current EIP (instruction pointer).

(31) *signal* %lu
The bitmap of pending signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use */proc/*pid*/status* instead.

(32) *blocked* %lu
The bitmap of blocked signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use */proc/*pid*/status* instead.

(33) *sigignore* %lu
The bitmap of ignored signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use */proc/*pid*/status* instead.

(34) *sigcatch* %lu
The bitmap of caught signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use */proc/*pid*/status* instead.

(35) *wchan* %lu [PT]
This is the "channel" in which the process is waiting. It is the address of a location in the kernel where the process is sleeping. The corresponding symbolic name can be found in */proc/*pid*/wchan*.

(36) *nswap* %lu
Number of pages swapped (not maintained).

(37) *cnswap* %lu
Cumulative *nswap* for child processes (not maintained).

(38) *exit_signal* %d (since Linux 2.1.22)
Signal to be sent to parent when we die.

(39) *processor* %d (since Linux 2.2.8)
CPU number last executed on.

(40) *rt_priority* %u (since Linux 2.5.19)
Real-time scheduling priority, a number in the range 1 to 99 for processes scheduled under a real-time policy, or 0, for non-real-time processes (see *sched_setscheduler*(2)).

(41) *policy* %u (since Linux 2.5.19)
Scheduling policy (see *sched_setscheduler*(2)). Decode using the SCHED_* constants in *linux/sched.h*.

The format for this field was %lu before Linux 2.6.22.

(42) *delayacct_blkio_ticks* %llu (since Linux 2.6.18)
Aggregated block I/O delays, measured in clock ticks (centiseconds).

(43) *guest_time* %lu (since Linux 2.6.24)
Guest time of the process (time spent running a virtual CPU for a guest operating system), measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*).

(44) *cguest_time* %ld (since Linux 2.6.24)
Guest time of the process's children, measured in clock ticks (divide by *sysconf(_SC_CLK_TCK)*).

(45) *start_data* %lu (since Linux 3.3) [PT]
Address above which program initialized and uninitialized (BSS) data are placed.

(46) *end_data* %lu (since Linux 3.3) [PT]
Address below which program initialized and uninitialized (BSS) data are placed.

(47) *start_brk* %lu (since Linux 3.3) [PT]
Address above which program heap can be expanded with *brk*(2).

(48) *arg_start* %lu (since Linux 3.5) [PT]
Address above which program command-line arguments (*argv*) are placed.

(49) *arg_end* %lu (since Linux 3.5) [PT]
Address below program command-line arguments (*argv*) are placed.

(50) *env_start* %lu (since Linux 3.5) [PT]
Address above which program environment is placed.

(51) *env_end* %lu (since Linux 3.5) [PT]
Address below which program environment is placed.

(52) *exit_code* %d (since Linux 3.5) [PT]
The thread's exit status in the form reported by *waitpid*(2).

SEE ALSO

proc(5), *proc_pid_status*(5)

NAME

/proc/pid/statm − memory usage information

DESCRIPTION

*/proc/*pid*/statm*

Provides information about memory usage, measured in pages. The columns are:

```
size       (1) total program size
           (same as VmSize in /proc/pid/status)
resident   (2) resident set size
           (inaccurate; same as VmRSS in /proc/pid/status)
shared     (3) number of resident shared pages
           (i.e., backed by a file)
           (inaccurate; same as RssFile+RssShmem in
           /proc/pid/status)
text       (4) text (code)
lib        (5) library (unused since Linux 2.6; always 0)
data       (6) data + stack
dt         (7) dirty pages (unused since Linux 2.6; always 0)
```

Some of these values are inaccurate because of a kernel-internal scalability optimization. If accurate values are required, use */proc/*pid*/smaps* or */proc/*pid*/smaps_rollup* instead, which are much slower but provide accurate, detailed information.

SEE ALSO

proc(5), *proc_pid_status*(5)

NAME

/proc/pid/status − memory usage and status information

DESCRIPTION

*/proc/*pid*/status*

Provides much of the information in */proc/*pid*/stat* and */proc/*pid*/statm* in a format that's easier for humans to parse. Here's an example:

```
$ cat /proc/$$/status
Name:   bash
Umask:  0022
State:  S (sleeping)
Tgid:   17248
Ngid:   0
Pid:    17248
PPid:   17200
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    100     100     100     100
FDSize: 256
Groups: 16 33 100
NStgid: 17248
NSpid:  17248
NSpgid: 17248
NSsid:  17200
VmPeak:       131168 kB
VmSize:       131168 kB
VmLck:             0 kB
VmPin:             0 kB
VmHWM:         13484 kB
VmRSS:         13484 kB
RssAnon:       10264 kB
RssFile:        3220 kB
RssShmem:          0 kB
VmData:        10332 kB
VmStk:           136 kB
VmExe:           992 kB
VmLib:          2104 kB
VmPTE:            76 kB
VmPMD:            12 kB
VmSwap:            0 kB
HugetlbPages:          0 kB         # 4.4
CoreDumping:      0                       # 4.15
Threads:        1
SigQ:   0/3067
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000010000
SigIgn: 0000000000384004
SigCgt: 000000004b813efb
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
CapAmb:     0000000000000000
NoNewPrivs:     0
Seccomp:        0
Seccomp_filters:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   00000001
```

```
Cpus_allowed_list:      0
Mems_allowed:   1
Mems_allowed_list:      0
voluntary_ctxt_switches:        150
nonvoluntary_ctxt_switches:     545
```

The fields are as follows:

Name Command run by this process. Strings longer than **TASK_COMM_LEN** (16) characters (including the terminating null byte) are silently truncated.

Umask Process umask, expressed in octal with a leading zero; see *umask*(2). (Since Linux 4.7.)

State Current state of the process. One of "R (running)", "S (sleeping)", "D (disk sleep)", "T (stopped)", "t (tracing stop)", "Z (zombie)", or "X (dead)".

Tgid Thread group ID (i.e., Process ID).

Ngid NUMA group ID (0 if none; since Linux 3.13).

Pid Thread ID (see *gettid*(2)).

PPid PID of parent process.

TracerPid
PID of process tracing this process (0 if not being traced).

Uid
Gid Real, effective, saved set, and filesystem UIDs (GIDs).

FDSize Number of file descriptor slots currently allocated.

Groups Supplementary group list.

NStgid Thread group ID (i.e., PID) in each of the PID namespaces of which *pid* is a member. The leftmost entry shows the value with respect to the PID namespace of the process that mounted this procfs (or the root namespace if mounted by the kernel), followed by the value in successively nested inner namespaces. (Since Linux 4.1.)

NSpid Thread ID in each of the PID namespaces of which *pid* is a member. The fields are ordered as for *NStgid*. (Since Linux 4.1.)

NSpgid Process group ID in each of the PID namespaces of which *pid* is a member. The fields are ordered as for *NStgid*. (Since Linux 4.1.)

NSsid descendant namespace session ID hierarchy Session ID in each of the PID namespaces of which *pid* is a member. The fields are ordered as for *NStgid*. (Since Linux 4.1.)

VmPeak
Peak virtual memory size.

VmSize Virtual memory size.

VmLck Locked memory size (see *mlock*(2)).

VmPin Pinned memory size (since Linux 3.2). These are pages that can't be moved because something needs to directly access physical memory.

VmHWM
Peak resident set size ("high water mark"). This value is inaccurate; see */proc/*pid*/statm* above.

VmRSS Resident set size. Note that the value here is the sum of *RssAnon*, *RssFile*, and *RssShmem*. This value is inaccurate; see */proc/*pid*/statm* above.

RssAnon
Size of resident anonymous memory. (since Linux 4.5). This value is inaccurate; see */proc/*pid*/statm* above.

RssFile Size of resident file mappings. (since Linux 4.5). This value is inaccurate; see */proc/*pid*/statm* above.

RssShmem
Size of resident shared memory (includes System V shared memory, mappings from *tmpfs*(5), and shared anonymous mappings). (since Linux 4.5).

VmData
VmStk
VmExe Size of data, stack, and text segments. This value is inaccurate; see */proc/*pid*/statm* above.

VmLib Shared library code size.

VmPTE
Page table entries size (since Linux 2.6.10).

VmPMD
Size of second-level page tables (added in Linux 4.0; removed in Linux 4.15).

VmSwap
Swapped-out virtual memory size by anonymous private pages; shmem swap usage is not included (since Linux 2.6.34). This value is inaccurate; see */proc/*pid*/statm* above.

HugetlbPages
Size of hugetlb memory portions (since Linux 4.4).

CoreDumping
Contains the value 1 if the process is currently dumping core, and 0 if it is not (since Linux 4.15). This information can be used by a monitoring process to avoid killing a process that is currently dumping core, which could result in a corrupted core dump file.

Threads
Number of threads in process containing this thread.

SigQ This field contains two slash-separated numbers that relate to queued signals for the real user ID of this process. The first of these is the number of currently queued signals for this real user ID, and the second is the resource limit on the number of queued signals for this process (see the description of **RLIMIT_SIGPENDING** in *getrlimit*(2)).

SigPnd
ShdPnd
Mask (expressed in hexadecimal) of signals pending for thread and for process as a whole (see *pthreads*(7) and *signal*(7)).

SigBlk
SigIgn
SigCgt Masks (expressed in hexadecimal) indicating signals being blocked, ignored, and caught (see *signal*(7)).

CapInh
CapPrm
CapEff Masks (expressed in hexadecimal) of capabilities enabled in inheritable, permitted, and effective sets (see *capabilities*(7)).

CapBnd
Capability bounding set, expressed in hexadecimal (since Linux 2.6.26, see *capabilities*(7)).

CapAmb
Ambient capability set, expressed in hexadecimal (since Linux 4.3, see *capabilities*(7)).

NoNewPrivs
: Value of the *no_new_privs* bit (since Linux 4.10, see *prctl*(2)).

Seccomp
: Seccomp mode of the process (since Linux 3.8, see *seccomp*(2)). 0 means **SECCOMP_MODE_DISABLED**; 1 means **SECCOMP_MODE_STRICT**; 2 means **SECCOMP_MODE_FILTER**. This field is provided only if the kernel was built with the **CONFIG_SECCOMP** kernel configuration option enabled.

Seccomp_filters
: Number of seccomp filters attached to the process (since Linux 5.9, see *seccomp*(2)).

Speculation_Store_Bypass
: Speculation flaw mitigation state (since Linux 4.17, see *prctl*(2)).

Cpus_allowed
: Hexadecimal mask of CPUs on which this process may run (since Linux 2.6.24, see *cpuset*(7)).

Cpus_allowed_list
: Same as previous, but in "list format" (since Linux 2.6.26, see *cpuset*(7)).

Mems_allowed
: Mask of memory nodes allowed to this process (since Linux 2.6.24, see *cpuset*(7)).

Mems_allowed_list
: Same as previous, but in "list format" (since Linux 2.6.26, see *cpuset*(7)).

voluntary_ctxt_switches
nonvoluntary_ctxt_switches
: Number of voluntary and involuntary context switches (since Linux 2.6.23).

SEE ALSO

proc(5)

NAME

/proc/pid/syscall − currently executed system call

DESCRIPTION

*/proc/*pid*/syscall* (since Linux 2.6.27)

This file exposes the system call number and argument registers for the system call currently being executed by the process, followed by the values of the stack pointer and program counter registers. The values of all six argument registers are exposed, although most system calls use fewer registers.

If the process is blocked, but not in a system call, then the file displays −1 in place of the system call number, followed by just the values of the stack pointer and program counter. If process is not blocked, then the file contains just the string "running".

This file is present only if the kernel was configured with **CONFIG_HAVE_ARCH_TRACEHOOK**.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/pid/task/, /proc/tid/, /proc/thread−self/ − thread information

DESCRIPTION

*/proc/*pid*/task/* (since Linux 2.6.0)

This is a directory that contains one subdirectory for each thread in the process. The name of each subdirectory is the numerical thread ID (*tid*) of the thread (see *gettid*(2)).

Within each of these subdirectories, there is a set of files with the same names and contents as under the */proc/*pid directories. For attributes that are shared by all threads, the contents for each of the files under the *task/*tid subdirectories will be the same as in the corresponding file in the parent */proc/*pid directory (e.g., in a multithreaded process, all of the *task/*tid*/cwd* files will have the same value as the */proc/*pid*/cwd* file in the parent directory, since all of the threads in a process share a working directory). For attributes that are distinct for each thread, the corresponding files under *task/*tid may have different values (e.g., various fields in each of the *task/*tid*/status* files may be different for each thread), or they might not exist in */proc/*pid at all.

In a multithreaded process, the contents of the */proc/*pid*/task* directory are not available if the main thread has already terminated (typically by calling *pthread_exit*(3)).

*/proc/*tid/

There is a numerical subdirectory for each running thread that is not a thread group leader (i.e., a thread whose thread ID is not the same as its process ID); the subdirectory is named by the thread ID. Each one of these subdirectories contains files and subdirectories exposing information about the thread with the thread ID *tid*. The contents of these directories are the same as the corresponding */proc/*pid*/task/*tid directories.

The */proc/*tid subdirectories are *not* visible when iterating through */proc* with *getdents*(2) (and thus are *not* visible when one uses *ls*(1) to view the contents of */proc*). However, the pathnames of these directories are visible to (i.e., usable as arguments in) system calls that operate on pathnames.

/proc/thread−self/ (since Linux 3.17)

This directory refers to the thread accessing the */proc* filesystem, and is identical to the */proc/self/task/*tid directory named by the process thread ID (*tid*) of the same thread.

SEE ALSO

proc(5)

NAME

/proc/pid/timers − POSIX timers

DESCRIPTION

*/proc/*pid*/timers* (since Linux 3.10)

A list of the POSIX timers for this process. Each timer is listed with a line that starts with the string "ID:". For example:

```
ID: 1
signal: 60/00007fff86e452a8
notify: signal/pid.2634
ClockID: 0
ID: 0
signal: 60/00007fff86e452a8
notify: signal/pid.2634
ClockID: 1
```

The lines shown for each timer have the following meanings:

ID The ID for this timer. This is not the same as the timer ID returned by *timer_create*(2); rather, it is the same kernel-internal ID that is available via the *si_timerid* field of the *siginfo_t* structure (see *sigaction*(2)).

signal This is the signal number that this timer uses to deliver notifications followed by a slash, and then the *sigev_value* value supplied to the signal handler. Valid only for timers that notify via a signal.

notify The part before the slash specifies the mechanism that this timer uses to deliver notifications, and is one of "thread", "signal", or "none". Immediately following the slash is either the string "tid" for timers with **SIGEV_THREAD_ID** notification, or "pid" for timers that notify by other mechanisms. Following the "." is the PID of the process (or the kernel thread ID of the thread) that will be delivered a signal if the timer delivers notifications via a signal.

ClockID

This field identifies the clock that the timer uses for measuring time. For most clocks, this is a number that matches one of the user-space **CLOCK_*** constants exposed via <*time.h*>. **CLOCK_PROCESS_CPUTIME_ID** timers display with a value of −6 in this field. **CLOCK_THREAD_CPUTIME_ID** timers display with a value of −2 in this field.

This file is available only when the kernel was configured with **CONFIG_CHECKPOINT_RESTORE**.

SEE ALSO

proc(5)

NAME

/proc/pid/timerslack_ns − timer slack in nanoseconds

DESCRIPTION

*/proc/*pid*/timerslack_ns* (since Linux 4.6)

This file exposes the process's "current" timer slack value, expressed in nanoseconds. The file is writable, allowing the process's timer slack value to be changed. Writing 0 to this file resets the "current" timer slack to the "default" timer slack value. For further details, see the discussion of **PR_SET_TIMERSLACK** in *prctl*(2).

Initially, permission to access this file was governed by a ptrace access mode **PTRACE_MODE_ATTACH_FSCREDS** check (see *ptrace*(2)). However, this was subsequently deemed too strict a requirement (and had the side effect that requiring a process to have the **CAP_SYS_PTRACE** capability would also allow it to view and change any process's memory). Therefore, since Linux 4.9, only the (weaker) **CAP_SYS_NICE** capability is required to access this file.

SEE ALSO

proc(5)

NAME

/proc/pid/gid_map, /proc/pid/uid_map − user and group ID mappings

DESCRIPTION

*/proc/*pid*/gid_map* (since Linux 3.5)
See *user_namespaces*(7).

*/proc/*pid*/uid_map* (since Linux 3.5)
See *user_namespaces*(7).

SEE ALSO

proc(5)

NAME

/proc/pid/wchan − wait channel

DESCRIPTION

*/proc/*pid*/wchan* (since Linux 2.6.0)

The symbolic name corresponding to the location in the kernel where the process is sleeping.

Permission to access this file is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

SEE ALSO

proc(5)

NAME

/proc/profile − kernel profiling

DESCRIPTION

/proc/profile (since Linux 2.4)

This file is present only if the kernel was booted with the *profile=1* command-line option. It exposes kernel profiling information in a binary format for use by *readprofile*(1)Writing (e.g., an empty string) to this file resets the profiling counters; on some architectures, writing a binary integer "profiling multiplier" of size *sizeof(int)* sets the profiling interrupt frequency.

SEE ALSO

proc(5)

NAME

/proc/scsi/ − SCSI

DESCRIPTION

/proc/scsi/

A directory with the *scsi* mid-level pseudo-file and various SCSI low-level driver directories, which contain a file for each SCSI host in this system, all of which give the status of some part of the SCSI IO subsystem. These files contain ASCII structures and are, therefore, readable with *cat*(1)

You can also write to some of the files to reconfigure the subsystem or switch certain features on or off.

/proc/scsi/scsi

This is a listing of all SCSI devices known to the kernel. The listing is similar to the one seen during bootup. scsi currently supports only the *add−single−device* command which allows root to add a hotplugged device to the list of known devices.

The command

```
echo 'scsi add−single−device 1 0 5 0' > /proc/scsi/scsi
```

will cause host scsi1 to scan on SCSI channel 0 for a device on ID 5 LUN 0. If there is already a device known on this address or the address is invalid, an error will be returned.

*/proc/scsi/*drivername*/*

drivername can currently be NCR53c7xx, aha152x, aha1542, aha1740, aic7xxx, buslogic, eata_dma, eata_pio, fdomain, in2000, pas16, qlogic, scsi_debug, seagate, t128, u15−24f, ultrastore, or wd7000. These directories show up for all drivers that registered at least one SCSI HBA. Every directory contains one file per registered host. Every host-file is named after the number the host was assigned during initialization.

Reading these files will usually show driver and host configuration, statistics, and so on.

Writing to these files allows different things on different hosts. For example, with the *latency* and *nolatency* commands, root can switch on and off command latency measurement code in the eata_dma driver. With the *lockup* and *unlock* commands, root can control bus lockups simulated by the scsi_debug driver.

SEE ALSO

proc(5)

NAME

/proc/slabinfo − kernel caches

DESCRIPTION

/proc/slabinfo

Information about kernel caches. See *slabinfo*(5) for details.

SEE ALSO

proc(5)

NAME

/proc/stat − kernel system statistics

DESCRIPTION

/proc/stat

kernel/system statistics. Varies with architecture. Common entries include:

cpu 10132153 290696 3084719 46828483 16683 0 25195 0 175628 0
cpu0 1393280 32966 572056 13343292 6130 0 17875 0 23933 0

The amount of time, measured in units of USER_HZ (1/100ths of a second on most architectures, use *sysconf(_SC_CLK_TCK)* to obtain the right value), that the system ("cpu" line) or the specific CPU ("cpu*N*" line) spent in various states:

user (1) Time spent in user mode.

nice (2) Time spent in user mode with low priority (nice).

system (3) Time spent in system mode.

idle (4) Time spent in the idle task. This value should be USER_HZ times the second entry in the */proc/uptime* pseudo-file.

iowait (since Linux 2.5.41)
(5) Time waiting for I/O to complete. This value is not reliable, for the following reasons:

- The CPU will not wait for I/O to complete; iowait is the time that a task is waiting for I/O to complete. When a CPU goes into idle state for outstanding task I/O, another task will be scheduled on this CPU.
- On a multi-core CPU, the task waiting for I/O to complete is not running on any CPU, so the iowait of each CPU is difficult to calculate.
- The value in this field may *decrease* in certain conditions.

irq (since Linux 2.6.0)
(6) Time servicing interrupts.

softirq (since Linux 2.6.0)
(7) Time servicing softirqs.

steal (since Linux 2.6.11)
(8) Stolen time, which is the time spent in other operating systems when running in a virtualized environment

guest (since Linux 2.6.24)
(9) Time spent running a virtual CPU for guest operating systems under the control of the Linux kernel.

guest_nice (since Linux 2.6.33)
(10) Time spent running a niced guest (virtual CPU for guest operating systems under the control of the Linux kernel).

page 5741 1808
The number of pages the system paged in and the number that were paged out (from disk).

swap 1 0
The number of swap pages that have been brought in and out.

intr 1462898
This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

disk_io: (2,0):(31,30,5764,1,2) (3,0):...
(major,disk_idx):(noinfo, read_io_ops, blks_read, write_io_ops, blks_written)
(Linux 2.4 only)

ctxt 115315
The number of context switches that the system underwent.

btime 769041601
boot time, in seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

processes 86031
Number of forks since boot.

procs_running 6
Number of processes in runnable state. (Linux 2.5.45 onward.)

procs_blocked 2
Number of processes blocked waiting for I/O to complete. (Linux 2.5.45 onward.)

softirq 229245889 94 60001584 13619 5175704 2471304 28 51212741 59130143 0 51240672
This line shows the number of softirq for all CPUs. The first column is the total of all softirqs and each subsequent column is the total for particular softirq. (Linux 2.6.31 onward.)

SEE ALSO

proc(5)

NAME

/proc/swaps − swap areas

DESCRIPTION

/proc/swaps

Swap areas in use. See also *swapon*(8)

SEE ALSO

proc(5)

NAME

/proc/sys/ − system information, and sysctl pseudo-filesystem

DESCRIPTION

/proc/sys/

This directory (present since Linux 1.3.57) contains a number of files and subdirectories corresponding to kernel variables. These variables can be read and in some cases modified using the */proc* filesystem, and the (deprecated) *sysctl*(2) system call.

String values may be terminated by either '\0' or '\n'.

Integer and long values may be written either in decimal or in hexadecimal notation (e.g., 0x3FFF). When writing multiple integer or long values, these may be separated by any of the following whitespace characters: ' ', '\t', or '\n'. Using other separators leads to the error **EINVAL**.

SEE ALSO

proc(5)

NAME

/proc/sys/abi/ − application binary information

DESCRIPTION

/proc/sys/abi/ (since Linux 2.4.10)

This directory may contain files with application binary information. See the Linux kernel source file *Documentation/sysctl/abi.rst* (or *Documentation/sysctl/abi.txt* before Linux 5.3) for more information.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/debug/ − debug

DESCRIPTION

/proc/sys/debug/

This directory may be empty.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/dev/ − device-specific information

DESCRIPTION

/proc/sys/dev/

This directory contains device-specific information (e.g., *dev/cdrom/info*). On some systems, it may be empty.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/fs/ − kernel variables related to filesystems

DESCRIPTION

/proc/sys/fs/

This directory contains the files and subdirectories for kernel variables related to filesystems.

/proc/sys/fs/aio−max−nr and */proc/sys/fs/aio−nr* (since Linux 2.6.4)

aio−nr is the running total of the number of events specified by *io_setup*(2) calls for all currently active AIO contexts. If *aio−nr* reaches *aio−max−nr*, then *io_setup*(2) will fail with the error **EAGAIN**. Raising *aio−max−nr* does not result in the preallocation or resizing of any kernel data structures.

/proc/sys/fs/binfmt_misc

Documentation for files in this directory can be found in the Linux kernel source in the file *Documentation/admin−guide/binfmt−misc.rst* (or in *Documentation/binfmt_misc.txt* on older kernels).

/proc/sys/fs/dentry−state (since Linux 2.2)

This file contains information about the status of the directory cache (dcache). The file contains six numbers, *nr_dentry*, *nr_unused*, *age_limit* (age in seconds), *want_pages* (pages requested by system) and two dummy values.

- *nr_dentry* is the number of allocated dentries (dcache entries). This field is unused in Linux 2.2.
- *nr_unused* is the number of unused dentries.
- *age_limit* is the age in seconds after which dcache entries can be reclaimed when memory is short.
- *want_pages* is nonzero when the kernel has called shrink_dcache_pages() and the dcache isn't pruned yet.

/proc/sys/fs/dir−notify−enable

This file can be used to disable or enable the *dnotify* interface described in *fcntl*(2) on a system-wide basis. A value of 0 in this file disables the interface, and a value of 1 enables it.

/proc/sys/fs/dquot−max

This file shows the maximum number of cached disk quota entries. On some (2.4) systems, it is not present. If the number of free cached disk quota entries is very low and you have some awesome number of simultaneous system users, you might want to raise the limit.

/proc/sys/fs/dquot−nr

This file shows the number of allocated disk quota entries and the number of free disk quota entries.

/proc/sys/fs/epoll/ (since Linux 2.6.28)

This directory contains the file *max_user_watches*, which can be used to limit the amount of kernel memory consumed by the *epoll* interface. For further details, see *epoll*(7).

/proc/sys/fs/file−max

This file defines a system-wide limit on the number of open files for all processes. System calls that fail when encountering this limit fail with the error **ENFILE**. (See also *setrlimit*(2), which can be used by a process to set the per-process limit, **RLIMIT_NOFILE**, on the number of files it may open.) If you get lots of error messages in the kernel log about running out of file handles (open file descriptions) (look for "VFS: file−max limit <number> reached"), try increasing this value:

```
echo 100000 > /proc/sys/fs/file−max
```

Privileged processes (**CAP_SYS_ADMIN**) can override the *file−max* limit.

/proc/sys/fs/file−nr

This (read-only) file contains three numbers: the number of allocated file handles (i.e., the number of open file descriptions; see *open*(2)); the number of free file handles; and the maximum number of file handles (i.e., the same value as */proc/sys/fs/file−max*). If the number of allocated file handles is close to the maximum, you should consider increasing the maximum.

Before Linux 2.6, the kernel allocated file handles dynamically, but it didn't free them again. Instead the free file handles were kept in a list for reallocation; the "free file handles" value indicates the size of that list. A large number of free file handles indicates that there was a past peak in the usage of open file handles. Since Linux 2.6, the kernel does deallocate freed file handles, and the "free file handles" value is always zero.

/proc/sys/fs/inode−max (only present until Linux 2.2)
This file contains the maximum number of in-memory inodes. This value should be 3–4 times larger than the value in *file−max*, since *stdin*, *stdout* and network sockets also need an inode to handle them. When you regularly run out of inodes, you need to increase this value.

Starting with Linux 2.4, there is no longer a static limit on the number of inodes, and this file is removed.

/proc/sys/fs/inode−nr
This file contains the first two values from *inode−state*.

/proc/sys/fs/inode−state
This file contains seven numbers: *nr_inodes*, *nr_free_inodes*, *preshrink*, and four dummy values (always zero).

nr_inodes is the number of inodes the system has allocated. *nr_free_inodes* represents the number of free inodes.

preshrink is nonzero when the *nr_inodes* > *inode−max* and the system needs to prune the inode list instead of allocating more; since Linux 2.4, this field is a dummy value (always zero).

/proc/sys/fs/inotify/ (since Linux 2.6.13)
This directory contains files *max_queued_events*, *max_user_instances*, and *max_user_watches*, that can be used to limit the amount of kernel memory consumed by the *inotify* interface. For further details, see *inotify*(7).

/proc/sys/fs/lease−break−time
This file specifies the grace period that the kernel grants to a process holding a file lease (**fcntl**(2)) after it has sent a signal to that process notifying it that another process is waiting to open the file. If the lease holder does not remove or downgrade the lease within this grace period, the kernel forcibly breaks the lease.

/proc/sys/fs/leases−enable
This file can be used to enable or disable file leases (**fcntl**(2)) on a system-wide basis. If this file contains the value 0, leases are disabled. A nonzero value enables leases.

/proc/sys/fs/mount−max (since Linux 4.9)
The value in this file specifies the maximum number of mounts that may exist in a mount namespace. The default value in this file is 100,000.

/proc/sys/fs/mqueue/ (since Linux 2.6.6)
This directory contains files *msg_max*, *msgsize_max*, and *queues_max*, controlling the resources used by POSIX message queues. See *mq_overview*(7) for details.

/proc/sys/fs/nr_open (since Linux 2.6.25)
This file imposes a ceiling on the value to which the **RLIMIT_NOFILE** resource limit can be raised (see *getrlimit*(2)). This ceiling is enforced for both unprivileged and privileged process. The default value in this file is 1048576. (Before Linux 2.6.25, the ceiling for **RLIMIT_NOFILE** was hard-coded to the same value.)

/proc/sys/fs/overflowgid and */proc/sys/fs/overflowuid*
These files allow you to change the value of the fixed UID and GID. The default is 65534. Some filesystems support only 16-bit UIDs and GIDs, although in Linux UIDs and GIDs are 32 bits. When one of these filesystems is mounted with writes enabled, any UID or GID that would exceed 65535 is translated to the overflow value before being written to disk.

/proc/sys/fs/pipe−max−size (since Linux 2.6.35)
See *pipe*(7).

/proc/sys/fs/pipe−user−pages−hard (since Linux 4.5)
See *pipe*(7).

/proc/sys/fs/pipe-user-pages-soft (since Linux 4.5)
See *pipe*(7).

/proc/sys/fs/protected_fifos (since Linux 4.19)
The value in this file is/can be set to one of the following:

Writing to FIFOs is unrestricted.

1 Don't allow **O_CREAT** *open*(2) on FIFOs that the caller doesn't own in world-writable sticky directories, unless the FIFO is owned by the owner of the directory.

2 As for the value 1, but the restriction also applies to group-writable sticky directories.

The intent of the above protections is to avoid unintentional writes to an attacker-controlled FIFO when a program expected to create a regular file.

/proc/sys/fs/protected_hardlinks (since Linux 3.6)
When the value in this file is 0, no restrictions are placed on the creation of hard links (i.e., this is the historical behavior before Linux 3.6). When the value in this file is 1, a hard link can be created to a target file only if one of the following conditions is true:

- The calling process has the **CAP_FOWNER** capability in its user namespace and the file UID has a mapping in the namespace.
- The filesystem UID of the process creating the link matches the owner (UID) of the target file (as described in *credentials*(7), a process's filesystem UID is normally the same as its effective UID).
- All of the following conditions are true:
  - the target is a regular file;
  - the target file does not have its set-user-ID mode bit enabled;
  - the target file does not have both its set-group-ID and group-executable mode bits enabled; and
  - the caller has permission to read and write the target file (either via the file's permissions mask or because it has suitable capabilities).

The default value in this file is 0. Setting the value to 1 prevents a longstanding class of security issues caused by hard-link-based time-of-check, time-of-use races, most commonly seen in world-writable directories such as */tmp*. The common method of exploiting this flaw is to cross privilege boundaries when following a given hard link (i.e., a root process follows a hard link created by another user). Additionally, on systems without separated partitions, this stops unauthorized users from "pinning" vulnerable set-user-ID and set-group-ID files against being upgraded by the administrator, or linking to special files.

/proc/sys/fs/protected_regular (since Linux 4.19)
The value in this file is/can be set to one of the following:

Writing to regular files is unrestricted.

1 Don't allow **O_CREAT** *open*(2) on regular files that the caller doesn't own in world-writable sticky directories, unless the regular file is owned by the owner of the directory.

2 As for the value 1, but the restriction also applies to group-writable sticky directories.

The intent of the above protections is similar to *protected_fifos*, but allows an application to avoid writes to an attacker-controlled regular file, where the application expected to create one.

/proc/sys/fs/protected_symlinks (since Linux 3.6)
When the value in this file is 0, no restrictions are placed on following symbolic links (i.e., this is the historical behavior before Linux 3.6). When the value in this file is 1, symbolic links are followed only in the following circumstances:

- the filesystem UID of the process following the link matches the owner (UID) of the symbolic link (as described in *credentials*(7), a process's filesystem UID is normally the same as its effective UID);

- the link is not in a sticky world-writable directory; or
- the symbolic link and its parent directory have the same owner (UID)

A system call that fails to follow a symbolic link because of the above restrictions returns the error **EACCES** in *errno*.

The default value in this file is 0. Setting the value to 1 avoids a longstanding class of security issues based on time-of-check, time-of-use races when accessing symbolic links.

/proc/sys/fs/suid_dumpable (since Linux 2.6.13)
The value in this file is assigned to a process's "dumpable" flag in the circumstances described in *prctl*(2). In effect, the value in this file determines whether core dump files are produced for set-user-ID or otherwise protected/tainted binaries. The "dumpable" setting also affects the ownership of files in a process's */proc/*pid directory, as described above.

Three different integer values can be specified:

0 (default)
This provides the traditional (pre-Linux 2.6.13) behavior. A core dump will not be produced for a process which has changed credentials (by calling *seteuid*(2), *setgid*(2), or similar, or by executing a set-user-ID or set-group-ID program) or whose binary does not have read permission enabled.

1 ("debug")
All processes dump core when possible. (Reasons why a process might nevertheless not dump core are described in *core*(5).) The core dump is owned by the filesystem user ID of the dumping process and no security is applied. This is intended for system debugging situations only: this mode is insecure because it allows unprivileged users to examine the memory contents of privileged processes.

2 ("suidsafe")
Any binary which normally would not be dumped (see "0" above) is dumped readable by root only. This allows the user to remove the core dump file but not to read it. For security reasons core dumps in this mode will not overwrite one another or other files. This mode is appropriate when administrators are attempting to debug problems in a normal environment.

Additionally, since Linux 3.6, */proc/sys/kernel/core_pattern* must either be an absolute pathname or a pipe command, as detailed in *core*(5). Warnings will be written to the kernel log if *core_pattern* does not follow these rules, and no core dump will be produced.

For details of the effect of a process's "dumpable" setting on ptrace access mode checking, see *ptrace*(2).

/proc/sys/fs/super−max
This file controls the maximum number of superblocks, and thus the maximum number of mounted filesystems the kernel can have. You need increase only *super−max* if you need to mount more filesystems than the current value in *super−max* allows you to.

/proc/sys/fs/super−nr
This file contains the number of filesystems currently mounted.

SEE ALSO
proc(5), *proc_sys*(5)

NAME

/proc/sys/kernel/ − control a range of kernel parameters

DESCRIPTION

/proc/sys/kernel/

This directory contains files controlling a range of kernel parameters, as described below.

/proc/sys/kernel/acct

This file contains three numbers: *highwater*, *lowwater*, and *frequency*. If BSD-style process accounting is enabled, these values control its behavior. If free space on filesystem where the log lives goes below *lowwater* percent, accounting suspends. If free space gets above *highwater* percent, accounting resumes. *frequency* determines how often the kernel checks the amount of free space (value is in seconds). Default values are 4, 2, and 30. That is, suspend accounting if 2% or less space is free; resume it if 4% or more space is free; consider information about amount of free space valid for 30 seconds.

/proc/sys/kernel/auto_msgmni (Linux 2.6.27 to Linux 3.18)

From Linux 2.6.27 to Linux 3.18, this file was used to control recomputing of the value in */proc/sys/kernel/msgmni* upon the addition or removal of memory or upon IPC namespace creation/removal. Echoing "1" into this file enabled *msgmni* automatic recomputing (and triggered a recomputation of *msgmni* based on the current amount of available memory and number of IPC namespaces). Echoing "0" disabled automatic recomputing. (Automatic recomputing was also disabled if a value was explicitly assigned to */proc/sys/kernel/msgmni*.) The default value in *auto_msgmni* was 1.

Since Linux 3.19, the content of this file has no effect (because *msgmni* defaults to near the maximum value possible), and reads from this file always return the value "0".

/proc/sys/kernel/cap_last_cap (since Linux 3.2)

See *capabilities*(7).

/proc/sys/kernel/cap−bound (from Linux 2.2 to Linux 2.6.24)

This file holds the value of the kernel *capability bounding set* (expressed as a signed decimal number). This set is ANDed against the capabilities permitted to a process during *execve*(2). Starting with Linux 2.6.25, the system-wide capability bounding set disappeared, and was replaced by a per-thread bounding set; see *capabilities*(7).

/proc/sys/kernel/core_pattern

See *core*(5).

/proc/sys/kernel/core_pipe_limit

See *core*(5).

/proc/sys/kernel/core_uses_pid

See *core*(5).

/proc/sys/kernel/ctrl−alt−del

This file controls the handling of Ctrl-Alt-Del from the keyboard. When the value in this file is 0, Ctrl-Alt-Del is trapped and sent to the *init*(1) program to handle a graceful restart. When the value is greater than zero, Linux's reaction to a Vulcan Nerve Pinch (tm) will be an immediate reboot, without even syncing its dirty buffers. Note: when a program (like dosemu) has the keyboard in "raw" mode, the Ctrl-Alt-Del is intercepted by the program before it ever reaches the kernel tty layer, and it's up to the program to decide what to do with it.

/proc/sys/kernel/dmesg_restrict (since Linux 2.6.37)

The value in this file determines who can see kernel syslog contents. A value of 0 in this file imposes no restrictions. If the value is 1, only privileged users can read the kernel syslog. (See *syslog*(2) for more details.) Since Linux 3.4, only users with the **CAP_SYS_ADMIN** capability may change the value in this file.

/proc/sys/kernel/domainname and */proc/sys/kernel/hostname*

can be used to set the NIS/YP domainname and the hostname of your box in exactly the same way as the commands *domainname*(1) and *hostname*(1), that is:

```
# echo 'darkstar' > /proc/sys/kernel/hostname
# echo 'mydomain' > /proc/sys/kernel/domainname
```

has the same effect as

```
# hostname 'darkstar'
# domainname 'mydomain'
```

Note, however, that the classic darkstar.frop.org has the hostname "darkstar" and DNS (Internet Domain Name Server) domainname "frop.org", not to be confused with the NIS (Network Information Service) or YP (Yellow Pages) domainname. These two domain names are in general different. For a detailed discussion see the *hostname*(1) man page.

/proc/sys/kernel/hotplug
This file contains the pathname for the hotplug policy agent. The default value in this file is */sbin/hotplug*.

/proc/sys/kernel/htab−reclaim (before Linux 2.4.9.2)
(PowerPC only) If this file is set to a nonzero value, the PowerPC htab (see kernel file *Documentation/powerpc/ppc_htab.txt*) is pruned each time the system hits the idle loop.

/proc/sys/kernel/keys/
This directory contains various files that define parameters and limits for the key-management facility. These files are described in *keyrings*(7).

/proc/sys/kernel/kptr_restrict (since Linux 2.6.38)
The value in this file determines whether kernel addresses are exposed via */proc* files and other interfaces. A value of 0 in this file imposes no restrictions. If the value is 1, kernel pointers printed using the *%pK* format specifier will be replaced with zeros unless the user has the **CAP_SYSLOG** capability. If the value is 2, kernel pointers printed using the *%pK* format specifier will be replaced with zeros regardless of the user's capabilities. The initial default value for this file was 1, but the default was changed to 0 in Linux 2.6.39. Since Linux 3.4, only users with the **CAP_SYS_ADMIN** capability can change the value in this file.

/proc/sys/kernel/l2cr
(PowerPC only) This file contains a flag that controls the L2 cache of G3 processor boards. If 0, the cache is disabled. Enabled if nonzero.

/proc/sys/kernel/modprobe
This file contains the pathname for the kernel module loader. The default value is */sbin/modprobe*. The file is present only if the kernel is built with the **CONFIG_MODULES** (**CONFIG_KMOD** in Linux 2.6.26 and earlier) option enabled. It is described by the Linux kernel source file *Documentation/kmod.txt* (present only in Linux 2.4 and earlier).

/proc/sys/kernel/modules_disabled (since Linux 2.6.31)
A toggle value indicating if modules are allowed to be loaded in an otherwise modular kernel. This toggle defaults to off (0), but can be set true (1). Once true, modules can be neither loaded nor unloaded, and the toggle cannot be set back to false. The file is present only if the kernel is built with the **CONFIG_MODULES** option enabled.

/proc/sys/kernel/msgmax (since Linux 2.2)
This file defines a system-wide limit specifying the maximum number of bytes in a single message written on a System V message queue.

/proc/sys/kernel/msgmni (since Linux 2.4)
This file defines the system-wide limit on the number of message queue identifiers. See also */proc/sys/kernel/auto_msgmni*.

/proc/sys/kernel/msgmnb (since Linux 2.2)
This file defines a system-wide parameter used to initialize the *msg_qbytes* setting for subsequently created message queues. The *msg_qbytes* setting specifies the maximum number of bytes that may be written to the message queue.

/proc/sys/kernel/ngroups_max (since Linux 2.6.4)
This is a read-only file that displays the upper limit on the number of a process's group memberships.

/proc/sys/kernel/ns_last_pid (since Linux 3.3)
See *pid_namespaces*(7).

/proc/sys/kernel/ostype and */proc/sys/kernel/osrelease*
These files give substrings of */proc/version*.

/proc/sys/kernel/overflowgid and */proc/sys/kernel/overflowuid*
These files duplicate the files */proc/sys/fs/overflowgid* and */proc/sys/fs/overflowuid*.

/proc/sys/kernel/panic
This file gives read/write access to the kernel variable *panic_timeout*. If this is zero, the kernel will loop on a panic; if nonzero, it indicates that the kernel should autoreboot after this number of seconds. When you use the software watchdog device driver, the recommended setting is 60.

/proc/sys/kernel/panic_on_oops (since Linux 2.5.68)
This file controls the kernel's behavior when an oops or BUG is encountered. If this file contains 0, then the system tries to continue operation. If it contains 1, then the system delays a few seconds (to give klogd time to record the oops output) and then panics. If the */proc/sys/kernel/panic* file is also nonzero, then the machine will be rebooted.

/proc/sys/kernel/pid_max (since Linux 2.5.34)
This file specifies the value at which PIDs wrap around (i.e., the value in this file is one greater than the maximum PID). PIDs greater than this value are not allocated; thus, the value in this file also acts as a system-wide limit on the total number of processes and threads. The default value for this file, 32768, results in the same range of PIDs as on earlier kernels. On 32-bit platforms, 32768 is the maximum value for *pid_max*. On 64-bit systems, *pid_max* can be set to any value up to 2^22 (**PID_MAX_LIMIT**, approximately 4 million).

/proc/sys/kernel/powersave−nap (PowerPC only)
This file contains a flag. If set, Linux-PPC will use the "nap" mode of powersaving, otherwise the "doze" mode will be used.

/proc/sys/kernel/printk
See *syslog*(2).

/proc/sys/kernel/pty (since Linux 2.6.4)
This directory contains two files relating to the number of UNIX 98 pseudoterminals (see *pts*(4)) on the system.

/proc/sys/kernel/pty/max
This file defines the maximum number of pseudoterminals.

/proc/sys/kernel/pty/nr
This read-only file indicates how many pseudoterminals are currently in use.

/proc/sys/kernel/random/
This directory contains various parameters controlling the operation of the file */dev/random*. See *random*(4) for further information.

/proc/sys/kernel/random/uuid (since Linux 2.4)
Each read from this read-only file returns a randomly generated 128-bit UUID, as a string in the standard UUID format.

/proc/sys/kernel/randomize_va_space (since Linux 2.6.12)
Select the address space layout randomization (ASLR) policy for the system (on architectures that support ASLR). Three values are supported for this file:

0 Turn ASLR off. This is the default for architectures that don't support ASLR, and when the kernel is booted with the *norandmaps* parameter.

1 Make the addresses of *mmap*(2) allocations, the stack, and the VDSO page randomized. Among other things, this means that shared libraries will be loaded at randomized addresses. The text segment of PIE-linked binaries will also be loaded at a randomized address. This value is the default if the kernel was configured with **CONFIG_COMPAT_BRK**.

2 (Since Linux 2.6.25) Also support heap randomization. This value is the default if the kernel was not configured with **CONFIG_COMPAT_BRK**.

/proc/sys/kernel/real−root−dev
This file is documented in the Linux kernel source file *Documentation/admin−guide/initrd.rst* (or *Documentation/initrd.txt* before Linux 4.10).

/proc/sys/kernel/reboot−cmd (Sparc only)
This file seems to be a way to give an argument to the SPARC ROM/Flash boot loader. Maybe to tell it what to do after rebooting?

/proc/sys/kernel/rtsig−max
(Up to and including Linux 2.6.7; see *setrlimit*(2)) This file can be used to tune the maximum number of POSIX real-time (queued) signals that can be outstanding in the system.

/proc/sys/kernel/rtsig−nr
(Up to and including Linux 2.6.7.) This file shows the number of POSIX real-time signals currently queued.

*/proc/*pid*/sched_autogroup_enabled* (since Linux 2.6.38)
See *sched*(7).

/proc/sys/kernel/sched_child_runs_first (since Linux 2.6.23)
If this file contains the value zero, then, after a *fork*(2), the parent is first scheduled on the CPU. If the file contains a nonzero value, then the child is scheduled first on the CPU. (Of course, on a multiprocessor system, the parent and the child might both immediately be scheduled on a CPU.)

/proc/sys/kernel/sched_rr_timeslice_ms (since Linux 3.9)
See *sched_rr_get_interval*(2).

/proc/sys/kernel/sched_rt_period_us (since Linux 2.6.25)
See *sched*(7).

/proc/sys/kernel/sched_rt_runtime_us (since Linux 2.6.25)
See *sched*(7).

/proc/sys/kernel/seccomp/ (since Linux 4.14)
This directory provides additional seccomp information and configuration. See *seccomp*(2) for further details.

/proc/sys/kernel/sem (since Linux 2.4)
This file contains 4 numbers defining limits for System V IPC semaphores. These fields are, in order:

SEMMSL
The maximum semaphores per semaphore set.

SEMMNS
A system-wide limit on the number of semaphores in all semaphore sets.

SEMOPM
The maximum number of operations that may be specified in a *semop*(2) call.

SEMMNI
A system-wide limit on the maximum number of semaphore identifiers.

/proc/sys/kernel/sg−big−buff
This file shows the size of the generic SCSI device (sg) buffer. You can't tune it just yet, but you could change it at compile time by editing *include/scsi/sg.h* and changing the value of **SG_BIG_BUFF**. However, there shouldn't be any reason to change this value.

/proc/sys/kernel/shm_rmid_forced (since Linux 3.1)
If this file is set to 1, all System V shared memory segments will be marked for destruction as soon as the number of attached processes falls to zero; in other words, it is no longer possible to create shared memory segments that exist independently of any attached process.

The effect is as though a *shmctl*(2) **IPC_RMID** is performed on all existing segments as well as all segments created in the future (until this file is reset to 0). Note that existing segments that are attached to no process will be immediately destroyed when this file is set to 1. Setting this option will also destroy segments that were created, but never attached, upon termination of the process that created the segment with *shmget*(2).

Setting this file to 1 provides a way of ensuring that all System V shared memory segments are counted against the resource usage and resource limits (see the description of **RLIMIT_AS** in *getrlimit*(2)) of at least one process.

Because setting this file to 1 produces behavior that is nonstandard and could also break existing applications, the default value in this file is 0. Set this file to 1 only if you have a good understanding of the semantics of the applications using System V shared memory on your system.

/proc/sys/kernel/shmall (since Linux 2.2)
This file contains the system-wide limit on the total number of pages of System V shared memory.

/proc/sys/kernel/shmmax (since Linux 2.2)
This file can be used to query and set the run-time limit on the maximum (System V IPC) shared memory segment size that can be created. Shared memory segments up to 1 GB are now supported in the kernel. This value defaults to **SHMMAX**.

/proc/sys/kernel/shmmni (since Linux 2.4)
This file specifies the system-wide maximum number of System V shared memory segments that can be created.

/proc/sys/kernel/sysctl_writes_strict (since Linux 3.16)
The value in this file determines how the file offset affects the behavior of updating entries in files under */proc/sys*. The file has three possible values:

−1 This provides legacy handling, with no printk warnings. Each *write*(2) must fully contain the value to be written, and multiple writes on the same file descriptor will overwrite the entire value, regardless of the file position.

(default) This provides the same behavior as for −1,
but printk warnings are written for processes that perform writes when the file offset is not 0.

1 Respect the file offset when writing strings into */proc/sys* files. Multiple writes will *append* to the value buffer. Anything written beyond the maximum length of the value buffer will be ignored. Writes to numeric */proc/sys* entries must always be at file offset 0 and the value must be fully contained in the buffer provided to *write*(2).

/proc/sys/kernel/sysrq
This file controls the functions allowed to be invoked by the SysRq key. By default, the file contains 1 meaning that every possible SysRq request is allowed (in older kernel versions, SysRq was disabled by default, and you were required to specifically enable it at run-time, but this is not the case any more). Possible values in this file are:

Disable sysrq completely

1 Enable all functions of sysrq

> 1 Bit mask of allowed sysrq functions, as follows:
2 Enable control of console logging level
4 Enable control of keyboard (SAK, unraw)
8 Enable debugging dumps of processes etc.
16 Enable sync command
32 Enable remount read-only
64 Enable signaling of processes (term, kill, oom-kill)
128 Allow reboot/poweroff
256 Allow nicing of all real-time tasks

This file is present only if the **CONFIG_MAGIC_SYSRQ** kernel configuration option is enabled. For further details see the Linux kernel source file *Documentation/admin−guide/sysrq.rst* (or *Documentation/sysrq.txt* before Linux 4.10).

/proc/sys/kernel/version
This file contains a string such as:

```
#5 Wed Feb 25 21:49:24 MET 1998
```

The "#5" means that this is the fifth kernel built from this source base and the date following it indicates the time the kernel was built.

/proc/sys/kernel/threads-max (since Linux 2.3.11)
This file specifies the system-wide limit on the number of threads (tasks) that can be created on the system.

Since Linux 4.1, the value that can be written to *threads-max* is bounded. The minimum value that can be written is 20. The maximum value that can be written is given by the constant **FUTEX_TID_MASK** (0x3fffffff). If a value outside of this range is written to *threads-max*, the error **EINVAL** occurs.

The value written is checked against the available RAM pages. If the thread structures would occupy too much (more than 1/8th) of the available RAM pages, *threads-max* is reduced accordingly.

/proc/sys/kernel/yama/ptrace_scope (since Linux 3.5)
See *ptrace*(2).

/proc/sys/kernel/zero-paged (PowerPC only)
This file contains a flag. When enabled (nonzero), Linux-PPC will pre-zero pages in the idle loop, possibly speeding up get_free_pages.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/net/ − networking

DESCRIPTION

/proc/sys/net/

This directory contains networking stuff. Explanations for some of the files under this directory can be found in *tcp*(7) and *ip*(7).

/proc/sys/net/core/bpf_jit_enable

See *bpf*(2).

/proc/sys/net/core/somaxconn

This file defines a ceiling value for the *backlog* argument of *listen*(2); see the *listen*(2) manual page for details.

SEE ALSO

proc(5), *proc_net*(5)

NAME

/proc/sys/proc/ − ???

DESCRIPTION

/proc/sys/proc/

This directory may be empty.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/sunrpc/ − Sun remote procedure call for NFS

DESCRIPTION

/proc/sys/sunrpc/

This directory supports Sun remote procedure call for network filesystem (NFS). On some systems, it is not present.

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/user/ − limits on the number of namespaces of various types

DESCRIPTION

/proc/sys/user/ (since Linux 4.9)

See *namespaces*(7).

SEE ALSO

proc(5), *proc_sys*(5)

NAME

/proc/sys/vm/ − virtual memory subsystem

DESCRIPTION

/proc/sys/vm/

This directory contains files for memory management tuning, buffer, and cache management.

/proc/sys/vm/admin_reserve_kbytes (since Linux 3.10)

This file defines the amount of free memory (in KiB) on the system that should be reserved for users with the capability **CAP_SYS_ADMIN**.

The default value in this file is the minimum of [3% of free pages, 8MiB] expressed as KiB. The default is intended to provide enough for the superuser to log in and kill a process, if necessary, under the default overcommit 'guess' mode (i.e., 0 in */proc/sys/vm/overcommit_memory*).

Systems running in "overcommit never" mode (i.e., 2 in */proc/sys/vm/overcommit_memory*) should increase the value in this file to account for the full virtual memory size of the programs used to recover (e.g., *login*(1) *ssh*(1), and *top*(1)) Otherwise, the superuser may not be able to log in to recover the system. For example, on x86-64 a suitable value is 131072 (128MiB reserved).

Changing the value in this file takes effect whenever an application requests memory.

/proc/sys/vm/compact_memory (since Linux 2.6.35)

When 1 is written to this file, all zones are compacted such that free memory is available in contiguous blocks where possible. The effect of this action can be seen by examining */proc/buddyinfo*.

Present only if the kernel was configured with **CONFIG_COMPACTION**.

/proc/sys/vm/drop_caches (since Linux 2.6.16)

Writing to this file causes the kernel to drop clean caches, dentries, and inodes from memory, causing that memory to become free. This can be useful for memory management testing and performing reproducible filesystem benchmarks. Because writing to this file causes the benefits of caching to be lost, it can degrade overall system performance.

To free pagecache, use:

```
echo 1 > /proc/sys/vm/drop_caches
```

To free dentries and inodes, use:

```
echo 2 > /proc/sys/vm/drop_caches
```

To free pagecache, dentries, and inodes, use:

```
echo 3 > /proc/sys/vm/drop_caches
```

Because writing to this file is a nondestructive operation and dirty objects are not freeable, the user should run *sync*(1) first.

/proc/sys/vm/sysctl_hugetlb_shm_group (since Linux 2.6.7)

This writable file contains a group ID that is allowed to allocate memory using huge pages. If a process has a filesystem group ID or any supplementary group ID that matches this group ID, then it can make huge-page allocations without holding the **CAP_IPC_LOCK** capability; see *memfd_create*(2), *mmap*(2), and *shmget*(2).

/proc/sys/vm/legacy_va_layout (since Linux 2.6.9)

If nonzero, this disables the new 32-bit memory-mapping layout; the kernel will use the legacy (2.4) layout for all processes.

/proc/sys/vm/memory_failure_early_kill (since Linux 2.6.32)

Control how to kill processes when an uncorrected memory error (typically a 2-bit error in a memory module) that cannot be handled by the kernel is detected in the background by hardware. In some cases (like the page still having a valid copy on disk), the kernel will handle the failure transparently without affecting any applications. But if there is no other up-to-date copy of the data, it will kill processes to prevent any data corruptions from propagating.

The file has one of the following values:

1 Kill all processes that have the corrupted-and-not-reloadable page mapped as soon as the corruption is detected. Note that this is not supported for a few types of pages, such as kernel internally allocated data or the swap cache, but works for the majority of user pages.

0 Unmap the corrupted page from all processes and kill a process only if it tries to access the page.

The kill is performed using a **SIGBUS** signal with *si_code* set to **BUS_MCEERR_AO**. Processes can handle this if they want to; see *sigaction*(2) for more details.

This feature is active only on architectures/platforms with advanced machine check handling and depends on the hardware capabilities.

Applications can override the *memory_failure_early_kill* setting individually with the *prctl*(2) **PR_MCE_KILL** operation.

Present only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

/proc/sys/vm/memory_failure_recovery (since Linux 2.6.32)
Enable memory failure recovery (when supported by the platform).

1 Attempt recovery.

0 Always panic on a memory failure.

Present only if the kernel was configured with **CONFIG_MEMORY_FAILURE**.

/proc/sys/vm/oom_dump_tasks (since Linux 2.6.25)
Enables a system-wide task dump (excluding kernel threads) to be produced when the kernel performs an OOM-killing. The dump includes the following information for each task (thread, process): thread ID, real user ID, thread group ID (process ID), virtual memory size, resident set size, the CPU that the task is scheduled on, oom_adj score (see the description of */proc/*pid*/oom_adj*), and command name. This is helpful to determine why the OOM-killer was invoked and to identify the rogue task that caused it.

If this contains the value zero, this information is suppressed. On very large systems with thousands of tasks, it may not be feasible to dump the memory state information for each one. Such systems should not be forced to incur a performance penalty in OOM situations when the information may not be desired.

If this is set to nonzero, this information is shown whenever the OOM-killer actually kills a memory-hogging task.

The default value is 0.

/proc/sys/vm/oom_kill_allocating_task (since Linux 2.6.24)
This enables or disables killing the OOM-triggering task in out-of-memory situations.

If this is set to zero, the OOM-killer will scan through the entire tasklist and select a task based on heuristics to kill. This normally selects a rogue memory-hogging task that frees up a large amount of memory when killed.

If this is set to nonzero, the OOM-killer simply kills the task that triggered the out-of-memory condition. This avoids a possibly expensive tasklist scan.

If */proc/sys/vm/panic_on_oom* is nonzero, it takes precedence over whatever value is used in */proc/sys/vm/oom_kill_allocating_task*.

The default value is 0.

/proc/sys/vm/overcommit_kbytes (since Linux 3.14)
This writable file provides an alternative to */proc/sys/vm/overcommit_ratio* for controlling the *CommitLimit* when */proc/sys/vm/overcommit_memory* has the value 2. It allows the amount of memory overcommitting to be specified as an absolute value (in kB), rather than as a percentage, as is done with *overcommit_ratio*. This allows for finer-grained control of *CommitLimit* on systems with extremely large memory sizes.

Only one of *overcommit_kbytes* or *overcommit_ratio* can have an effect: if *overcommit_kbytes* has a nonzero value, then it is used to calculate *CommitLimit*, otherwise *overcommit_ratio* is used. Writing a value to either of these files causes the value in the other file to be set to zero.

/proc/sys/vm/overcommit_memory

This file contains the kernel virtual memory accounting mode. Values are:

0: heuristic overcommit (this is the default)
1: always overcommit, never check
2: always check, never overcommit

In mode 0, calls of *mmap*(2) with **MAP_NORESERVE** are not checked, and the default check is very weak, leading to the risk of getting a process "OOM-killed".

In mode 1, the kernel pretends there is always enough memory, until memory actually runs out. One use case for this mode is scientific computing applications that employ large sparse arrays. Before Linux 2.6.0, any nonzero value implies mode 1.

In mode 2 (available since Linux 2.6), the total virtual address space that can be allocated (*CommitLimit* in */proc/meminfo*) is calculated as

```
CommitLimit = (total_RAM − total_huge_TLB) *
              overcommit_ratio / 100 + total_swap
```

where:

- *total_RAM* is the total amount of RAM on the system;
- *total_huge_TLB* is the amount of memory set aside for huge pages;
- *overcommit_ratio* is the value in */proc/sys/vm/overcommit_ratio*; and
- *total_swap* is the amount of swap space.

For example, on a system with 16 GB of physical RAM, 16 GB of swap, no space dedicated to huge pages, and an *overcommit_ratio* of 50, this formula yields a *CommitLimit* of 24 GB.

Since Linux 3.14, if the value in */proc/sys/vm/overcommit_kbytes* is nonzero, then *CommitLimit* is instead calculated as:

```
CommitLimit = overcommit_kbytes + total_swap
```

See also the description of */proc/sys/vm/admin_reserve_kbytes* and */proc/sys/vm/user_reserve_kbytes*.

/proc/sys/vm/overcommit_ratio (since Linux 2.6.0)

This writable file defines a percentage by which memory can be overcommitted. The default value in the file is 50. See the description of */proc/sys/vm/overcommit_memory*.

/proc/sys/vm/panic_on_oom (since Linux 2.6.18)

This enables or disables a kernel panic in an out-of-memory situation.

If this file is set to the value 0, the kernel's OOM-killer will kill some rogue process. Usually, the OOM-killer is able to kill a rogue process and the system will survive.

If this file is set to the value 1, then the kernel normally panics when out-of-memory happens. However, if a process limits allocations to certain nodes using memory policies (**mbind**(2) **MPOL_BIND**) or cpusets (**cpuset**(7)) and those nodes reach memory exhaustion status, one process may be killed by the OOM-killer. No panic occurs in this case: because other nodes' memory may be free, this means the system as a whole may not have reached an out-of-memory situation yet.

If this file is set to the value 2, the kernel always panics when an out-of-memory condition occurs.

The default value is 0. 1 and 2 are for failover of clustering. Select either according to your policy of failover.

/proc/sys/vm/swappiness

The value in this file controls how aggressively the kernel will swap memory pages. Higher values increase aggressiveness, lower values decrease aggressiveness. The default value is 60.

/proc/sys/vm/user_reserve_kbytes (since Linux 3.10)
Specifies an amount of memory (in KiB) to reserve for user processes. This is intended to prevent a user from starting a single memory hogging process, such that they cannot recover (kill the hog). The value in this file has an effect only when */proc/sys/vm/overcommit_memory* is set to 2 ("overcommit never" mode). In this case, the system reserves an amount of memory that is the minimum of [3% of current process size, *user_reserve_kbytes*].

The default value in this file is the minimum of [3% of free pages, 128MiB] expressed as KiB.

If the value in this file is set to zero, then a user will be allowed to allocate all free memory with a single process (minus the amount reserved by */proc/sys/vm/admin_reserve_kbytes*). Any subsequent attempts to execute a command will result in "fork: Cannot allocate memory".

Changing the value in this file takes effect whenever an application requests memory.

/proc/sys/vm/unprivileged_userfaultfd (since Linux 5.2)
This (writable) file exposes a flag that controls whether unprivileged processes are allowed to employ *userfaultfd*(2). If this file has the value 1, then unprivileged processes may use *userfaultfd*(2). If this file has the value 0, then only processes that have the **CAP_SYS_PTRACE** capability may employ *userfaultfd*(2). The default value in this file is 1.

SEE ALSO
proc(5), *proc_sys*(5)

NAME

/proc/sysrq−trigger − SysRq function

DESCRIPTION

/proc/sysrq−trigger (since Linux 2.4.21)

Writing a character to this file triggers the same SysRq function as typing ALT-SysRq-<character> (see the description of */proc/sys/kernel/sysrq*). This file is normally writable only by *root*. For further details see the Linux kernel source file *Documentation/admin−guide/sysrq.rst* (or *Documentation/sysrq.txt* before Linux 4.10).

SEE ALSO

proc(5)

NAME

/proc/sysvipc/ − System V IPC

DESCRIPTION

/proc/sysvipc/

Subdirectory containing the pseudo-files *msg*, *sem* and *shm*. These files list the System V Interprocess Communication (IPC) objects (respectively: message queues, semaphores, and shared memory) that currently exist on the system, providing similar information to that available via *ipcs*(1)These files have headers and are formatted (one IPC object per line) for easy understanding. *sysvipc*(7) provides further background on the information shown by these files.

SEE ALSO

proc(5)

NAME

/proc/tid/children − child tasks

DESCRIPTION

*/proc/*tid*/children* (since Linux 3.5)

A space-separated list of child tasks of this task. Each child task is represented by its TID.

This option is intended for use by the checkpoint-restore (CRIU) system, and reliably provides a list of children only if all of the child processes are stopped or frozen. It does not work properly if children of the target task exit while the file is being read! Exiting children may cause non-exiting children to be omitted from the list. This makes this interface even more unreliable than classic PID-based approaches if the inspected task and its children aren't frozen, and most code should probably not use this interface.

Until Linux 4.2, the presence of this file was governed by the **CONFIG_CHECKPOINT_RESTORE** kernel configuration option. Since Linux 4.2, it is governed by the **CONFIG_PROC_CHILDREN** option.

SEE ALSO

proc(5)

NAME

/proc/timer_list − pending timers

DESCRIPTION

/proc/timer_list (since Linux 2.6.21)

This read-only file exposes a list of all currently pending (high-resolution) timers, all clock-event sources, and their parameters in a human-readable form.

SEE ALSO

proc(5)

NAME

/proc/timer_stats − timer statistics

DESCRIPTION

/proc/timer_stats (from Linux 2.6.21 until Linux 4.10)

This is a debugging facility to make timer (ab)use in a Linux system visible to kernel and user-space developers. It can be used by kernel and user-space developers to verify that their code does not make undue use of timers. The goal is to avoid unnecessary wakeups, thereby optimizing power consumption.

If enabled in the kernel (**CONFIG_TIMER_STATS**), but not used, it has almost zero run-time overhead and a relatively small data-structure overhead. Even if collection is enabled at run time, overhead is low: all the locking is per-CPU and lookup is hashed.

The */proc/timer_stats* file is used both to control sampling facility and to read out the sampled information.

The *timer_stats* functionality is inactive on bootup. A sampling period can be started using the following command:

```
# echo 1 > /proc/timer_stats
```

The following command stops a sampling period:

```
# echo 0 > /proc/timer_stats
```

The statistics can be retrieved by:

```
$ cat /proc/timer_stats
```

While sampling is enabled, each readout from */proc/timer_stats* will see newly updated statistics. Once sampling is disabled, the sampled information is kept until a new sample period is started. This allows multiple readouts.

Sample output from */proc/timer_stats*:

```
$ cat /proc/timer_stats
Timer Stats Version: v0.3
Sample period: 1.764 s
Collection: active
  255,     0 swapper/3        hrtimer_start_range_ns (tick_sched_timer)
   71,     0 swapper/1        hrtimer_start_range_ns (tick_sched_timer)
   58,     0 swapper/0        hrtimer_start_range_ns (tick_sched_timer)
    4,  1694 gnome-shell      mod_delayed_work_on (delayed_work_timer_fn)
   17,     7 rcu_sched        rcu_gp_kthread (process_timeout)
...
    1,  4911 kworker/u16:0    mod_delayed_work_on (delayed_work_timer_fn)
   1D,  2522 kworker/0:0      queue_delayed_work_on (delayed_work_timer_fn)
1029 total events, 583.333 events/sec
```

The output columns are:

[1] a count of the number of events, optionally (since Linux 2.6.23) followed by the letter 'D' if this is a deferrable timer;

[2] the PID of the process that initialized the timer;

[3] the name of the process that initialized the timer;

[4] the function where the timer was initialized; and (in parentheses) the callback function that is associated with the timer.

During the Linux 4.11 development cycle, this file was removed because of security concerns, as it exposes information across namespaces. Furthermore, it is possible to obtain the same information via in-kernel tracing facilities such as ftrace.

SEE ALSO

proc(5)

NAME

/proc/tty/ − tty

DESCRIPTION

/proc/tty/

Subdirectory containing the pseudo-files and subdirectories for tty drivers and line disciplines.

SEE ALSO

proc(5)

NAME

/proc/uptime − system uptime

DESCRIPTION

/proc/uptime

This file contains two numbers (values in seconds): the uptime of the system (including time spent in suspend) and the amount of time spent in the idle process.

SEE ALSO

proc(5)

NAME

/proc/version − kernel version

DESCRIPTION

/proc/version

This string identifies the kernel version that is currently running. It includes the contents of */proc/sys/kernel/ostype*, */proc/sys/kernel/osrelease*, and */proc/sys/kernel/version*. For example:

```
Linux version 1.0.9 (quinlan@phaze) #1 Sat May 14 01:51:54 EDT 1994
```

SEE ALSO

proc(5)

NAME

/proc/vmstat − virtual memory statistics

DESCRIPTION

/proc/vmstat (since Linux 2.6.0)

This file displays various virtual memory statistics. Each line of this file contains a single name-value pair, delimited by white space. Some lines are present only if the kernel was configured with suitable options. (In some cases, the options required for particular files have changed across kernel versions, so they are not listed here. Details can be found by consulting the kernel source code.) The following fields may be present:

nr_free_pages (since Linux 2.6.31)

nr_alloc_batch (since Linux 3.12)

nr_inactive_anon (since Linux 2.6.28)

nr_active_anon (since Linux 2.6.28)

nr_inactive_file (since Linux 2.6.28)

nr_active_file (since Linux 2.6.28)

nr_unevictable (since Linux 2.6.28)

nr_mlock (since Linux 2.6.28)

nr_anon_pages (since Linux 2.6.18)

nr_mapped (since Linux 2.6.0)

nr_file_pages (since Linux 2.6.18)

nr_dirty (since Linux 2.6.0)

nr_writeback (since Linux 2.6.0)

nr_slab_reclaimable (since Linux 2.6.19)

nr_slab_unreclaimable (since Linux 2.6.19)

nr_page_table_pages (since Linux 2.6.0)

nr_kernel_stack (since Linux 2.6.32)
Amount of memory allocated to kernel stacks.

nr_unstable (since Linux 2.6.0)

nr_bounce (since Linux 2.6.12)

nr_vmscan_write (since Linux 2.6.19)

nr_vmscan_immediate_reclaim (since Linux 3.2)

nr_writeback_temp (since Linux 2.6.26)

nr_isolated_anon (since Linux 2.6.32)

nr_isolated_file (since Linux 2.6.32)

nr_shmem (since Linux 2.6.32)
Pages used by shmem and *tmpfs*(5).

nr_dirtied (since Linux 2.6.37)

nr_written (since Linux 2.6.37)

nr_pages_scanned (since Linux 3.17)

numa_hit (since Linux 2.6.18)

numa_miss (since Linux 2.6.18)

numa_foreign (since Linux 2.6.18)

numa_interleave (since Linux 2.6.18)

numa_local (since Linux 2.6.18)

numa_other (since Linux 2.6.18)

workingset_refault (since Linux 3.15)

workingset_activate (since Linux 3.15)

workingset_nodereclaim (since Linux 3.15)

nr_anon_transparent_hugepages (since Linux 2.6.38)

nr_free_cma (since Linux 3.7)
Number of free CMA (Contiguous Memory Allocator) pages.

nr_dirty_threshold (since Linux 2.6.37)

nr_dirty_background_threshold (since Linux 2.6.37)

pgpgin (since Linux 2.6.0)

pgpgout (since Linux 2.6.0)

pswpin (since Linux 2.6.0)

pswpout (since Linux 2.6.0)

pgalloc_dma (since Linux 2.6.5)

pgalloc_dma32 (since Linux 2.6.16)

pgalloc_normal (since Linux 2.6.5)

pgalloc_high (since Linux 2.6.5)

pgalloc_movable (since Linux 2.6.23)

pgfree (since Linux 2.6.0)

pgactivate (since Linux 2.6.0)

pgdeactivate (since Linux 2.6.0)

pgfault (since Linux 2.6.0)

pgmajfault (since Linux 2.6.0)

pgrefill_dma (since Linux 2.6.5)

pgrefill_dma32 (since Linux 2.6.16)

pgrefill_normal (since Linux 2.6.5)

pgrefill_high (since Linux 2.6.5)

pgrefill_movable (since Linux 2.6.23)

pgsteal_kswapd_dma (since Linux 3.4)

pgsteal_kswapd_dma32 (since Linux 3.4)

pgsteal_kswapd_normal (since Linux 3.4)

pgsteal_kswapd_high (since Linux 3.4)

pgsteal_kswapd_movable (since Linux 3.4)

pgsteal_direct_dma

pgsteal_direct_dma32 (since Linux 3.4)

pgsteal_direct_normal (since Linux 3.4)

pgsteal_direct_high (since Linux 3.4)

pgsteal_direct_movable (since Linux 2.6.23)

pgscan_kswapd_dma

pgscan_kswapd_dma32 (since Linux 2.6.16)

pgscan_kswapd_normal (since Linux 2.6.5)

pgscan_kswapd_high

pgscan_kswapd_movable (since Linux 2.6.23)

pgscan_direct_dma

pgscan_direct_dma32 (since Linux 2.6.16)

pgscan_direct_normal

pgscan_direct_high

pgscan_direct_movable (since Linux 2.6.23)

pgscan_direct_throttle (since Linux 3.6)

zone_reclaim_failed (since linux 2.6.31)

pginodesteal (since linux 2.6.0)

slabs_scanned (since linux 2.6.5)

kswapd_inodesteal (since linux 2.6.0)

kswapd_low_wmark_hit_quickly (since Linux 2.6.33)

kswapd_high_wmark_hit_quickly (since Linux 2.6.33)

pageoutrun (since Linux 2.6.0)

allocstall (since Linux 2.6.0)

pgrotated (since Linux 2.6.0)

drop_pagecache (since Linux 3.15)

drop_slab (since Linux 3.15)

numa_pte_updates (since Linux 3.8)

numa_huge_pte_updates (since Linux 3.13)

numa_hint_faults (since Linux 3.8)

numa_hint_faults_local (since Linux 3.8)

numa_pages_migrated (since Linux 3.8)

pgmigrate_success (since Linux 3.8)

pgmigrate_fail (since Linux 3.8)

compact_migrate_scanned (since Linux 3.8)

compact_free_scanned (since Linux 3.8)

compact_isolated (since Linux 3.8)

compact_stall (since Linux 2.6.35)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

compact_fail (since Linux 2.6.35)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

compact_success (since Linux 2.6.35)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

htlb_buddy_alloc_success (since Linux 2.6.26)

htlb_buddy_alloc_fail (since Linux 2.6.26)

unevictable_pgs_culled (since Linux 2.6.28)

unevictable_pgs_scanned (since Linux 2.6.28)

unevictable_pgs_rescued (since Linux 2.6.28)

unevictable_pgs_mlocked (since Linux 2.6.28)

unevictable_pgs_munlocked (since Linux 2.6.28)

unevictable_pgs_cleared (since Linux 2.6.28)

unevictable_pgs_stranded (since Linux 2.6.28)

thp_fault_alloc (since Linux 2.6.39)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_fault_fallback (since Linux 2.6.39)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_collapse_alloc (since Linux 2.6.39)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_collapse_alloc_failed (since Linux 2.6.39)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_split (since Linux 2.6.39)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_zero_page_alloc (since Linux 3.8)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

thp_zero_page_alloc_failed (since Linux 3.8)
See the kernel source file *Documentation/admin−guide/mm/transhuge.rst*.

balloon_inflate (since Linux 3.18)

balloon_deflate (since Linux 3.18)

balloon_migrate (since Linux 3.18)

nr_tlb_remote_flush (since Linux 3.12)

nr_tlb_remote_flush_received (since Linux 3.12)

nr_tlb_local_flush_all (since Linux 3.12)

nr_tlb_local_flush_one (since Linux 3.12)

vmacache_find_calls (since Linux 3.16)

vmacache_find_hits (since Linux 3.16)

vmacache_full_flushes (since Linux 3.19)

SEE ALSO

proc(5)

NAME

/proc/zoneinfo − memory zones

DESCRIPTION

/proc/zoneinfo (since Linux 2.6.13)

This file displays information about memory zones. This is useful for analyzing virtual memory behavior.

SEE ALSO

proc(5)

NAME

protocols − protocols definition file

DESCRIPTION

This file is a plain ASCII file, describing the various DARPA internet protocols that are available from the TCP/IP subsystem. It should be consulted instead of using the numbers in the ARPA include files, or, even worse, just guessing them. These numbers will occur in the protocol field of any IP header.

Keep this file untouched since changes would result in incorrect IP packages. Protocol numbers and names are specified by the IANA (Internet Assigned Numbers Authority).

Each line is of the following format:

protocol number aliases ...

where the fields are delimited by spaces or tabs. Empty lines are ignored. If a line contains a hash mark (#), the hash mark and the part of the line following it are ignored.

The field descriptions are:

protocol
: the native name for the protocol. For example *ip*, *tcp*, or *udp*.

number
: the official number for this protocol as it will appear within the IP header.

aliases optional aliases for the protocol.

This file might be distributed over a network using a network-wide naming service like Yellow Pages/NIS or BIND/Hesiod.

FILES

/etc/protocols
: The protocols definition file.

SEE ALSO

getprotoent(3)

NAME

repertoiremap − map symbolic character names to Unicode code points

DESCRIPTION

A repertoire map defines mappings between symbolic character names (mnemonics) and Unicode code points when compiling a locale with *localedef*(1). Using a repertoire map is optional, it is needed only when symbolic names are used instead of now preferred Unicode code points.

Syntax

The repertoiremap file starts with a header that may consist of the following keywords:

comment_char
: is followed by a character that will be used as the comment character for the rest of the file. It defaults to the number sign (#).

escape_char
: is followed by a character that should be used as the escape character for the rest of the file to mark characters that should be interpreted in a special way. It defaults to the backslash (\).

The mapping section starts with the keyword *CHARIDS* in the first column.

The mapping lines have the following form:

<*symbolic-name*> <*code-point*> *comment*
: This defines exactly one mapping, *comment* being optional.

The mapping section ends with the string *END CHARIDS*.

FILES

/usr/share/i18n/repertoiremaps
: Usual default repertoire map path.

STANDARDS

POSIX.2.

NOTES

Repertoire maps are deprecated in favor of Unicode code points.

EXAMPLES

A mnemonic for the Euro sign can be defined as follows:

<Eu> <U20AC> EURO SIGN

SEE ALSO

locale(1), *localedef*(1), *charmap*(5), *locale*(5)

NAME

resolv.conf − resolver configuration file

SYNOPSIS

/etc/resolv.conf

DESCRIPTION

The *resolver* is a set of routines in the C library that provide access to the Internet Domain Name System (DNS). The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of keywords with values that provide various types of resolver information. The configuration file is considered a trusted source of DNS information; see the **trust-ad** option below for details.

If this file does not exist, only the name server on the local machine will be queried, and the search list contains the local domain name determined from the hostname.

The different configuration options are:

nameserver Name server IP address

Internet address of a name server that the resolver should query, either an IPv4 address (in dot notation), or an IPv6 address in colon (and possibly dot) notation as per RFC 2373. Up to **MAXNS** (currently 3, see <*resolv.h*>) name servers may be listed, one per keyword. If there are multiple servers, the resolver library queries them in the order listed. If no **nameserver** entries are present, the default is to use the name server on the local machine. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made.)

search Search list for host-name lookup.

By default, the search list contains one entry, the local domain name. It is determined from the local hostname returned by *gethostname*(2); the local domain name is taken to be everything after the first '.'. Finally, if the hostname does not contain a '.', the root domain is assumed as the local domain name.

This may be changed by listing the desired domain search path following the *search* keyword with spaces or tabs separating the names. Resolver queries having fewer than *ndots* dots (default is 1) in them will be attempted using each component of the search path in turn until a match is found. For environments with multiple subdomains please read **options ndots:***n* below to avoid man-in-the-middle attacks and unnecessary traffic for the root-dns-servers. Note that this process may be slow and will generate a lot of network traffic if the servers for the listed domains are not local, and that queries will time out if no server is available for one of the domains.

If there are multiple **search** directives, only the search list from the last instance is used.

In glibc 2.25 and earlier, the search list is limited to six domains with a total of 256 characters. Since glibc 2.26, the search list is unlimited.

The **domain** directive is an obsolete name for the **search** directive that handles one search list entry only.

sortlist This option allows addresses returned by *gethostbyname*(3) to be sorted. A sortlist is specified by IP-address-netmask pairs. The netmask is optional and defaults to the natural netmask of the net. The IP address and optional network pairs are separated by slashes. Up to 10 pairs may be specified. Here is an example:

```
sortlist 130.155.160.0/255.255.240.0 130.155.0.0
```

options

Options allows certain internal resolver variables to be modified. The syntax is

options *option* ...

where *option* is one of the following:

debug Sets **RES_DEBUG** in *_res.options* (effective only if glibc was built with debug support; see *resolver*(3)).

ndots:*n*
Sets a threshold for the number of dots which must appear in a name given to *res_query*(3) (see *resolver*(3)) before an *initial absolute query* will be made. The default for *n* is 1, meaning that if there are any dots in a name, the name will be tried first as an absolute name before any *search list* elements are appended to it. The value for this option is silently capped to 15.

timeout:*n*
Sets the amount of time the resolver will wait for a response from a remote name server before retrying the query via a different name server. This may **not** be the total time taken by any resolver API call and there is no guarantee that a single resolver API call maps to a single timeout. Measured in seconds, the default is **RES_TIMEOUT** (currently 5, see <*resolv.h*>). The value for this option is silently capped to 30.

attempts:*n*
Sets the number of times the resolver will send a query to its name servers before giving up and returning an error to the calling application. The default is **RES_DFLRETRY** (currently 2, see <*resolv.h*>). The value for this option is silently capped to 5.

rotate Sets **RES_ROTATE** in *_res.options*, which causes round-robin selection of name servers from among those listed. This has the effect of spreading the query load among all listed servers, rather than having all clients try the first listed server first every time.

no−aaaa (since glibc 2.36)
Sets **RES_NOAAAA** in *_res.options*, which suppresses AAAA queries made by the stub resolver, including AAAA lookups triggered by NSS-based interfaces such as *getaddrinfo*(3). Only DNS lookups are affected: IPv6 data in *hosts*(5) is still used, *getaddrinfo*(3) with **AI_PASSIVE** will still produce IPv6 addresses, and configured IPv6 name servers are still used. To produce correct Name Error (NXDOMAIN) results, AAAA queries are translated to A queries. This option is intended preliminary for diagnostic purposes, to rule out that AAAA DNS queries have adverse impact. It is incompatible with EDNS0 usage and DNSSEC validation by applications.

no−check−names
Sets **RES_NOCHECKNAME** in *_res.options*, which disables the modern BIND checking of incoming hostnames and mail names for invalid characters such as underscore (_), non-ASCII, or control characters.

inet6 Sets **RES_USE_INET6** in *_res.options*. This has the effect of trying an AAAA query before an A query inside the *gethostbyname*(3) function, and of mapping IPv4 responses in IPv6 "tunneled form" if no AAAA records are found but an A record set exists. Since glibc 2.25, this option is deprecated; applications should use *getaddrinfo*(3), rather than *gethostbyname*(3).

ip6-bytestring (since glibc 2.3.4 to glibc 2.24)
Sets **RES_USEBSTRING** in *_res.options*. This causes reverse IPv6 lookups to be made using the bit-label format described in RFC 2673; if this option is not set (which is the default), then nibble format is used. This option was removed in glibc 2.25, since it relied on a backward-incompatible DNS extension that was never deployed on the Internet.

ip6-dotint/**no-ip6-dotint** (glibc 2.3.4 to glibc 2.24)
Clear/set **RES_NOIP6DOTINT** in *_res.options*. When this option is clear (**ip6−dotint**), reverse IPv6 lookups are made in the (deprecated) *ip6.int* zone; when this option is set (**no−ip6−dotint**), reverse IPv6 lookups are made in the *ip6.arpa* zone by default. These options are available up to glibc 2.24, where **no−ip6−dotint** is the default. Since **ip6−dotint** support long ago ceased to be available on the Internet, these options were removed in glibc 2.25.

edns0 (since glibc 2.6)
Sets **RES_USE_EDNS0** in *_res.options*. This enables support for the DNS extensions described in RFC 2671.

single-request (since glibc 2.10)
Sets **RES_SNGLKUP** in *_res.options*. By default, glibc performs IPv4 and IPv6 lookups in parallel since glibc 2.9. Some appliance DNS servers cannot handle these queries properly and make the requests time out. This option disables the behavior and makes glibc perform the IPv6 and IPv4 requests sequentially (at the cost of some slowdown of the resolving process).

single-request-reopen (since glibc 2.9)
Sets **RES_SNGLKUPREOP** in *_res.options*. The resolver uses the same socket for the A and AAAA requests. Some hardware mistakenly sends back only one reply. When that happens the client system will sit and wait for the second reply. Turning this option on changes this behavior so that if two requests from the same port are not handled correctly it will close the socket and open a new one before sending the second request.

no-tld-query (since glibc 2.14)
Sets **RES_NOTLDQUERY** in *_res.options*. This option causes **res_nsearch**() to not attempt to resolve an unqualified name as if it were a top level domain (TLD). This option can cause problems if the site has ‘‘localhost’’ as a TLD rather than having localhost on one or more elements of the search list. This option has no effect if neither RES_DEFNAMES or RES_DNSRCH is set.

use-vc (since glibc 2.14)
Sets **RES_USEVC** in *_res.options*. This option forces the use of TCP for DNS resolutions.

no-reload (since glibc 2.26)
Sets **RES_NORELOAD** in *_res.options*. This option disables automatic reloading of a changed configuration file.

trust-ad (since glibc 2.31)
Sets **RES_TRUSTAD** in *_res.options*. This option controls the AD bit behavior of the stub resolver. If a validating resolver sets the AD bit in a response, it indicates that the data in the response was verified according to the DNSSEC protocol. In order to rely on the AD bit, the local system has to trust both the DNSSEC-validating resolver and the network path to it, which is why an explicit opt-in is required. If the **trust−ad** option is active, the stub resolver sets the AD bit in outgoing DNS queries (to enable AD bit support), and preserves the AD bit in responses. Without this option, the AD bit is not set in queries, and it is always removed from responses before they are returned to the application. This means that applications can trust the AD bit in responses if the **trust−ad** option has been set correctly.

In glibc 2.30 and earlier, the AD is not set automatically in queries, and is passed through unchanged to applications in responses.

The *search* keyword of a system’s *resolv.conf* file can be overridden on a per-process basis by setting the environment variable **LOCALDOMAIN** to a space-separated list of search domains.

The *options* keyword of a system’s *resolv.conf* file can be amended on a per-process basis by setting the environment variable **RES_OPTIONS** to a space-separated list of resolver options as explained above under **options**.

The keyword and value must appear on a single line, and the keyword (e.g., **nameserver**) must start the line. The value follows the keyword, separated by white space.

Lines that contain a semicolon (;) or hash character (#) in the first column are treated as comments.

FILES

/etc/resolv.conf , *<resolv.h>*

SEE ALSO

gethostbyname(3), *resolver*(3), *host.conf*(5), *hosts*(5), *nsswitch.conf*(5), *hostname*(7), *named*(8)

Name Server Operations Guide for BIND

NAME

rpc − RPC program number data base

SYNOPSIS

/etc/rpc

DESCRIPTION

The *rpc* file contains user readable names that can be used in place of RPC program numbers. Each line has the following information:

- name of server for the RPC program
- RPC program number
- aliases

Items are separated by any number of blanks and/or tab characters. A '#' indicates the beginning of a comment; characters from the '#' to the end of the line are not interpreted by routines which search the file.

Here is an example of the */etc/rpc* file from the Sun RPC Source distribution.

```
#
# rpc 88/08/01 4.0 RPCSRC; from 1.12   88/02/07 SMI
#
portmapper      100000  portmap sunrpc
rstatd          100001  rstat rstat_svc rup perfmeter
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
ypbind          100007
walld           100008  rwall shutdown
yppasswdd       100009  yppasswd
etherstatd      100010  etherstat
rquotad         100011  rquotaprog quota rquota
sprayd          100012  spray
3270_mapper     100013
rje_mapper      100014
selection_svc   100015  selnsvc
database_svc    100016
rexd            100017  rex
alis            100018
sched           100019
llockmgr        100020
nlockmgr        100021
x25.inr         100022
statmon         100023
status          100024
bootparam       100026
ypupdated       100028  ypupdate
keyserv         100029  keyserver
tfsd            100037
nsed            100038
nsemntd         100039
```

FILES

/etc/rpc

RPC program number data base

SEE ALSO

getrpcent(3)

NAME

securetty − list of terminals on which root is allowed to login

DESCRIPTION

The file */etc/securetty* contains the names of terminals (one per line, without leading */dev/*) which are considered secure for the transmission of certain authentication tokens.

It is used by (some versions of) *login*(1) to restrict the terminals on which root is allowed to login. See *login.defs*(5) if you use the shadow suite.

On PAM enabled systems, it is used for the same purpose by *pam_securetty*(8) to restrict the terminals on which empty passwords are accepted.

FILES

/etc/securetty

SEE ALSO

login(1), *login.defs*(5), *pam_securetty*(8)

NAME

services − Internet network services list

DESCRIPTION

services is a plain ASCII file providing a mapping between human-friendly textual names for internet services, and their underlying assigned port numbers and protocol types. Every networking program should look into this file to get the port number (and protocol) for its service. The C library routines *getservent*(3), *getservbyname*(3), *getservbyport*(3), *setservent*(3), and *endservent*(3) support querying this file from programs.

Port numbers are assigned by the IANA (Internet Assigned Numbers Authority), and their current policy is to assign both TCP and UDP protocols when assigning a port number. Therefore, most entries will have two entries, even for TCP-only services.

Port numbers below 1024 (so-called "low numbered" ports) can be bound to only by root (see *bind*(2), *tcp*(7), and *udp*(7)). This is so clients connecting to low numbered ports can trust that the service running on the port is the standard implementation, and not a rogue service run by a user of the machine. Well-known port numbers specified by the IANA are normally located in this root-only space.

The presence of an entry for a service in the **services** file does not necessarily mean that the service is currently running on the machine. See *inetd.conf* (5) for the configuration of Internet services offered. Note that not all networking services are started by *inetd*(8), and so won't appear in *inetd.conf* (5)In particular, news (NNTP) and mail (SMTP) servers are often initialized from the system boot scripts.

The location of the **services** file is defined by **_PATH_SERVICES** in <*netdb.h*>. This is usually set to */etc/services*.

Each line describes one service, and is of the form:

service-name *port/protocol* [*aliases* ...]

where:

service-name
: is the friendly name the service is known by and looked up under. It is case sensitive. Often, the client program is named after the *service-name*.

port : is the port number (in decimal) to use for this service.

protocol
: is the type of protocol to be used. This field should match an entry in the *protocols*(5) file. Typical values include **tcp** and **udp**.

aliases : is an optional space or tab separated list of other names for this service. Again, the names are case sensitive.

Either spaces or tabs may be used to separate the fields.

Comments are started by the hash sign (#) and continue until the end of the line. Blank lines are skipped.

The *service-name* should begin in the first column of the file, since leading spaces are not stripped. *service-names* can be any printable characters excluding space and tab. However, a conservative choice of characters should be used to minimize compatibility problems. For example, a−z, 0−9, and hyphen (−) would seem a sensible choice.

Lines not matching this format should not be present in the file. (Currently, they are silently skipped by *getservent*(3), *getservbyname*(3), and *getservbyport*(3). However, this behavior should not be relied on.)

This file might be distributed over a network using a network-wide naming service like Yellow Pages/NIS or BIND/Hesiod.

A sample **services** file might look like this:

```
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp          # message send protocol
msp             18/udp          # message send protocol
chargen         19/tcp          ttytst source
```

```
chargen         19/udp          ttytst source
ftp             21/tcp
# 22 - unassigned
telnet          23/tcp
```

FILES

/etc/services
: The Internet network services list

<netdb.h>
: Definition of **_PATH_SERVICES**

SEE ALSO

listen(2), *endservent*(3), *getservbyname*(3), *getservbyport*(3), *getservent*(3), *setservent*(3), *inetd.conf*(5), *protocols*(5), *inetd*(8)

Assigned Numbers RFC, most recently RFC 1700, (AKA STD0002).

NAME

shells − pathnames of valid login shells

DESCRIPTION

/etc/shells is a text file which contains the full pathnames of valid login shells. This file is consulted by *chsh*(1) and available to be queried by other programs.

Be aware that there are programs which consult this file to find out if a user is a normal user; for example, FTP daemons traditionally disallow access to users with shells not included in this file.

FILES

/etc/shells

EXAMPLES

/etc/shells may contain the following paths:

```
/bin/sh
/bin/bash
/bin/csh
```

SEE ALSO

chsh(1), *getusershell*(3), *pam_shells*(8)

NAME

slabinfo − kernel slab allocator statistics

SYNOPSIS

cat /proc/slabinfo

DESCRIPTION

Frequently used objects in the Linux kernel (buffer heads, inodes, dentries, etc.) have their own cache. The file */proc/slabinfo* gives statistics on these caches. The following (edited) output shows an example of the contents of this file:

```
$ sudo cat /proc/slabinfo
slabinfo − version: 2.1
# name    <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> ...
sigqueue      100  100  160   25  1 : tunables  0  0  0 : slabdata   4   4  0
sighand_cache 355   405 2112  15  8 : tunables  0  0  0 : slabdata  27  27  0
kmalloc-8192   96   96  8192   4  8 : tunables  0  0  0 : slabdata  24  24  0
...
```

The first line of output includes a version number, which allows an application that is reading the file to handle changes in the file format. (See VERSIONS, below.) The next line lists the names of the columns in the remaining lines.

Each of the remaining lines displays information about a specified cache. Following the cache name, the output shown in each line shows three components for each cache:

- statistics
- tunables
- slabdata

The statistics are as follows:

active_objs
: The number of objects that are currently active (i.e., in use).

num_objs
: The total number of allocated objects (i.e., objects that are both in use and not in use).

objsize The size of objects in this slab, in bytes.

objperslab
: The number of objects stored in each slab.

pagesperslab
: The number of pages allocated for each slab.

The *tunables* entries in each line show tunable parameters for the corresponding cache. When using the default SLUB allocator, there are no tunables, the */proc/slabinfo* file is not writable, and the value 0 is shown in these fields. When using the older SLAB allocator, the tunables for a particular cache can be set by writing lines of the following form to */proc/slabinfo*:

```
# echo 'name limit batchcount sharedfactor' > /proc/slabinfo
```

Here, *name* is the cache name, and *limit*, *batchcount*, and *sharedfactor* are integers defining new values for the corresponding tunables. The *limit* value should be a positive value, *batchcount* should be a positive value that is less than or equal to *limit*, and *sharedfactor* should be nonnegative. If any of the specified values is invalid, the cache settings are left unchanged.

The *tunables* entries in each line contain the following fields:

limit The maximum number of objects that will be cached.

batchcount
: On SMP systems, this specifies the number of objects to transfer at one time when refilling the available object list.

sharedfactor
: [To be documented]

The *slabdata* entries in each line contain the following fields:

active_slabs
The number of active slabs.

nums_slabs
The total number of slabs.

sharedavail
[To be documented]

Note that because of object alignment and slab cache overhead, objects are not normally packed tightly into pages. Pages with even one in-use object are considered in-use and cannot be freed.

Kernels configured with **CONFIG_DEBUG_SLAB** will also have additional statistics fields in each line, and the first line of the file will contain the string "(statistics)". The statistics field include : the high water mark of active objects; the number of times objects have been allocated; the number of times the cache has grown (new pages added to this cache); the number of times the cache has been reaped (unused pages removed from this cache); and the number of times there was an error allocating new pages to this cache.

VERSIONS

The */proc/slabinfo* file first appeared in Linux 2.1.23. The file is versioned, and over time there have been a number of versions with different layouts:

1.0 Present throughout the Linux 2.2.x kernel series.

1.1 Present in the Linux 2.4.x kernel series.

1.2 A format that was briefly present in the Linux 2.5 development series.

2.0 Present in Linux 2.6.x kernels up to and including Linux 2.6.9.

2.1 The current format, which first appeared in Linux 2.6.10.

NOTES

Only root can read and (if the kernel was configured with **CONFIG_SLAB**) write the */proc/slabinfo* file.

The total amount of memory allocated to the SLAB/SLUB cache is shown in the *Slab* field of */proc/meminfo*.

SEE ALSO

slabtop(1)

The kernel source file *Documentation/vm/slub.txt* and *tools/vm/slabinfo.c*.

NAME

sysfs − a filesystem for exporting kernel objects

DESCRIPTION

The **sysfs** filesystem is a pseudo-filesystem which provides an interface to kernel data structures. (More precisely, the files and directories in **sysfs** provide a view of the *kobject* structures defined internally within the kernel.) The files under **sysfs** provide information about devices, kernel modules, filesystems, and other kernel components.

The **sysfs** filesystem is commonly mounted at */sys*. Typically, it is mounted automatically by the system, but it can also be mounted manually using a command such as:

```
mount -t sysfs sysfs /sys
```

Many of the files in the **sysfs** filesystem are read-only, but some files are writable, allowing kernel variables to be changed. To avoid redundancy, symbolic links are heavily used to connect entries across the filesystem tree.

Files and directories

The following list describes some of the files and directories under the */sys* hierarchy.

/sys/block
: This subdirectory contains one symbolic link for each block device that has been discovered on the system. The symbolic links point to corresponding directories under */sys/devices*.

/sys/bus
: This directory contains one subdirectory for each of the bus types in the kernel. Inside each of these directories are two subdirectories:

  devices This subdirectory contains symbolic links to entries in */sys/devices* that correspond to the devices discovered on this bus.

  drivers This subdirectory contains one subdirectory for each device driver that is loaded on this bus.

/sys/class
: This subdirectory contains a single layer of further subdirectories for each of the device classes that have been registered on the system (e.g., terminals, network devices, block devices, graphics devices, sound devices, and so on). Inside each of these subdirectories are symbolic links for each of the devices in this class. These symbolic links refer to entries in the */sys/devices* directory.

/sys/class/net
: Each of the entries in this directory is a symbolic link representing one of the real or virtual networking devices that are visible in the network namespace of the process that is accessing the directory. Each of these symbolic links refers to entries in the */sys/devices* directory.

/sys/dev
: This directory contains two subdirectories *block/* and *char/*, corresponding, respectively, to the block and character devices on the system. Inside each of these subdirectories are symbolic links with names of the form *major-ID*:*minor-ID*, where the ID values correspond to the major and minor ID of a specific device. Each symbolic link points to the **sysfs** directory for a device. The symbolic links inside */sys/dev* thus provide an easy way to look up the **sysfs** interface using the device IDs returned by a call to *stat*(2) (or similar).

  The following shell session shows an example from */sys/dev*:

```
$ stat -c "%t %T" /dev/null
1 3
$ readlink /sys/dev/char/1\:3
../../devices/virtual/mem/null
$ ls -Fd /sys/devices/virtual/mem/null
/sys/devices/virtual/mem/null/
$ ls -d1 /sys/devices/virtual/mem/null/*
/sys/devices/virtual/mem/null/dev
/sys/devices/virtual/mem/null/power/
/sys/devices/virtual/mem/null/subsystem@
```

```
/sys/devices/virtual/mem/null/uevent
```

/sys/devices
: This is a directory that contains a filesystem representation of the kernel device tree, which is a hierarchy of *device* structures within the kernel.

/sys/firmware
: This subdirectory contains interfaces for viewing and manipulating firmware-specific objects and attributes.

/sys/fs
: This directory contains subdirectories for some filesystems. A filesystem will have a subdirectory here only if it chose to explicitly create the subdirectory.

/sys/fs/cgroup
: This directory conventionally is used as a mount point for a *tmpfs*(5) filesystem containing mount points for *cgroups*(7) filesystems.

/sys/fs/smackfs
: The directory contains configuration files for the SMACK LSM. See the kernel source file *Documentation/admin−guide/LSM/Smack.rst*.

/sys/hypervisor
: [To be documented]

/sys/kernel
: This subdirectory contains various files and subdirectories that provide information about the running kernel.

/sys/kernel/cgroup/
: For information about the files in this directory, see *cgroups*(7).

/sys/kernel/debug/tracing
: Mount point for the *tracefs* filesystem used by the kernel's *ftrace* facility. (For information on *ftrace*, see the kernel source file *Documentation/trace/ftrace.txt*.)

/sys/kernel/mm
: This subdirectory contains various files and subdirectories that provide information about the kernel's memory management subsystem.

/sys/kernel/mm/hugepages
: This subdirectory contains one subdirectory for each of the huge page sizes that the system supports. The subdirectory name indicates the huge page size (e.g., *hugepages−2048kB*). Within each of these subdirectories is a set of files that can be used to view and (in some cases) change settings associated with that huge page size. For further information, see the kernel source file *Documentation/admin−guide/mm/hugetlbpage.rst*.

/sys/module
: This subdirectory contains one subdirectory for each module that is loaded into the kernel. The name of each directory is the name of the module. In each of the subdirectories, there may be following files:

  coresize
  : [to be documented]

  initsize [to be documented]

  initstate
  : [to be documented]

  refcnt [to be documented]

  srcversion
  : [to be documented]

  taint [to be documented]

  uevent [to be documented]

  version [to be documented]

In each of the subdirectories, there may be following subdirectories:

drivers [To be documented]

holders [To be documented]

notes [To be documented]

parameters
This directory contains one file for each module parameter, with each file containing the value of the corresponding parameter. Some of these files are writable, allowing the

sections
This subdirectories contains files with information about module sections. This information is mainly used for debugging.

[To be documented]

/sys/power
[To be documented]

STANDARDS
Linux.

HISTORY
Linux 2.6.0.

NOTES
This manual page is incomplete, possibly inaccurate, and is the kind of thing that needs to be updated very often.

SEE ALSO
proc(5), *udev*(7)

P. Mochel. (2005). *The sysfs filesystem*. Proceedings of the 2005 Ottawa Linux Symposium.

The kernel source file *Documentation/filesystems/sysfs.txt* and various other files in *Documentation/ABI* and *Documentation/*/sysfs.txt*

NAME

termcap − terminal capability database

DESCRIPTION

The termcap database is an obsolete facility for describing the capabilities of character-cell terminals and printers. It is retained only for compatibility with old programs; new programs should use the *terminfo*(5) database and associated libraries.

/etc/termcap is an ASCII file (the database master) that lists the capabilities of many different types of terminals. Programs can read termcap to find the particular escape codes needed to control the visual attributes of the terminal actually in use. (Other aspects of the terminal are handled by *stty*(1)The termcap database is indexed on the **TERM** environment variable.

Termcap entries must be defined on a single logical line, with '\' used to suppress the newline. Fields are separated by ':'. The first field of each entry starts at the left-hand margin, and contains a list of names for the terminal, separated by '|'.

The first subfield may (in BSD termcap entries from 4.3BSD and earlier) contain a short name consisting of two characters. This short name may consist of capital or small letters. In 4.4BSD, termcap entries this field is omitted.

The second subfield (first, in the newer 4.4BSD format) contains the name used by the environment variable **TERM**. It should be spelled in lowercase letters. Selectable hardware capabilities should be marked by appending a hyphen and a suffix to this name. See below for an example. Usual suffixes are w (more than 80 characters wide), am (automatic margins), nam (no automatic margins), and rv (reverse video display). The third subfield contains a long and descriptive name for this termcap entry.

Subsequent fields contain the terminal capabilities; any continued capability lines must be indented one tab from the left margin.

Although there is no defined order, it is suggested to write first boolean, then numeric, and then string capabilities, each sorted alphabetically without looking at lower or upper spelling. Capabilities of similar functions can be written in one line.

Example for:

Head line: vt|vt101|DEC VT 101 terminal in 80 character mode:\
Head line: Vt|vt101-w|DEC VT 101 terminal in (wide) 132 character mode:\
Boolean: :bs:\
Numeric: :co#80:\
String: :sr=\E[H:\

Boolean capabilities

5i	Printer will not echo on screen
am	Automatic margins which means automatic line wrap
bs	Control-H (8 dec.) performs a backspace
bw	Backspace on left margin wraps to previous line and right margin
da	Display retained above screen
db	Display retained below screen
eo	A space erases all characters at cursor position
es	Escape sequences and special characters work in status line
gn	Generic device
hc	This is a hardcopy terminal
HC	The cursor is hard to see when not on bottom line
hs	Has a status line
hz	Hazeltine bug, the terminal can not print tilde characters
in	Terminal inserts null bytes, not spaces, to fill whitespace
km	Terminal has a meta key
mi	Cursor movement works in insert mode
ms	Cursor movement works in standout/underline mode
NP	No pad character
NR	ti does not reverse te
nx	No padding, must use XON/XOFF
os	Terminal can overstrike
ul	Terminal underlines although it can not overstrike

xb	Beehive glitch, f1 sends ESCAPE, f2 sends **^C**
xn	Newline/wraparound glitch
xo	Terminal uses xon/xoff protocol
xs	Text typed over standout text will be displayed in standout
xt	Teleray glitch, destructive tabs and odd standout mode

Numeric capabilities

co	Number of columns
dB	Delay in milliseconds for backspace on hardcopy terminals
dC	Delay in milliseconds for carriage return on hardcopy terminals
dF	Delay in milliseconds for form feed on hardcopy terminals
dN	Delay in milliseconds for new line on hardcopy terminals
dT	Delay in milliseconds for tabulator stop on hardcopy terminals
dV	Delay in milliseconds for vertical tabulator stop on hardcopy terminals
it	Difference between tab positions
lh	Height of soft labels
lm	Lines of memory
lw	Width of soft labels
li	Number of lines
Nl	Number of soft labels
pb	Lowest baud rate which needs padding
sg	Standout glitch
ug	Underline glitch
vt	virtual terminal number
ws	Width of status line if different from screen width

String capabilities

!1	shifted save key
!2	shifted suspend key
!3	shifted undo key
#1	shifted help key
#2	shifted home key
#3	shifted input key
#4	shifted cursor left key
%0	redo key
%1	help key
%2	mark key
%3	message key
%4	move key
%5	next-object key
%6	open key
%7	options key
%8	previous-object key
%9	print key
%a	shifted message key
%b	shifted move key
%c	shifted next key
%d	shifted options key
%e	shifted previous key
%f	shifted print key
%g	shifted redo key
%h	shifted replace key
%i	shifted cursor right key
%j	shifted resume key
&0	shifted cancel key
&1	reference key
&2	refresh key
&3	replace key
&4	restart key

&5	resume key
&6	save key
&7	suspend key
&8	undo key
&9	shifted begin key
*0	shifted find key
*1	shifted command key
*2	shifted copy key
*3	shifted create key
*4	shifted delete character
*5	shifted delete line
*6	select key
*7	shifted end key
*8	shifted clear line key
*9	shifted exit key
@0	find key
@1	begin key
@2	cancel key
@3	close key
@4	command key
@5	copy key
@6	create key
@7	end key
@8	enter/send key
@9	exit key
al	Insert one line
AL	Insert %1 lines
ac	Pairs of block graphic characters to map alternate character set
ae	End alternative character set
as	Start alternative character set for block graphic characters
bc	Backspace, if not **^H**
bl	Audio bell
bt	Move to previous tab stop
cb	Clear from beginning of line to cursor
cc	Dummy command character
cd	Clear to end of screen
ce	Clear to end of line
ch	Move cursor horizontally only to column %1
cl	Clear screen and cursor home
cm	Cursor move to row %1 and column %2 (on screen)
CM	Move cursor to row %1 and column %2 (in memory)
cr	Carriage return
cs	Scroll region from line %1 to %2
ct	Clear tabs
cv	Move cursor vertically only to line %1
dc	Delete one character
DC	Delete %1 characters
dl	Delete one line
DL	Delete %1 lines
dm	Begin delete mode
do	Cursor down one line
DO	Cursor down #1 lines
ds	Disable status line
eA	Enable alternate character set
ec	Erase %1 characters starting at cursor
ed	End delete mode
ei	End insert mode
ff	Formfeed character on hardcopy terminals
fs	Return character to its position before going to status line

F1	The string sent by function key f11
F2	The string sent by function key f12
F3	The string sent by function key f13
...	...
F9	The string sent by function key f19
FA	The string sent by function key f20
FB	The string sent by function key f21
...	...
FZ	The string sent by function key f45
Fa	The string sent by function key f46
Fb	The string sent by function key f47
...	...
Fr	The string sent by function key f63
hd	Move cursor a half line down
ho	Cursor home
hu	Move cursor a half line up
i1	Initialization string 1 at login
i3	Initialization string 3 at login
is	Initialization string 2 at login
ic	Insert one character
IC	Insert %1 characters
if	Initialization file
im	Begin insert mode
ip	Insert pad time and needed special characters after insert
iP	Initialization program
K1	upper left key on keypad
K2	center key on keypad
K3	upper right key on keypad
K4	bottom left key on keypad
K5	bottom right key on keypad
k0	Function key 0
k1	Function key 1
k2	Function key 2
k3	Function key 3
k4	Function key 4
k5	Function key 5
k6	Function key 6
k7	Function key 7
k8	Function key 8
k9	Function key 9
k;	Function key 10
ka	Clear all tabs key
kA	Insert line key
kb	Backspace key
kB	Back tab stop
kC	Clear screen key
kd	Cursor down key
kD	Key for delete character under cursor
ke	turn keypad off
kE	Key for clear to end of line
kF	Key for scrolling forward/down
kh	Cursor home key
kH	Cursor hown down key
kI	Insert character/Insert mode key
kl	Cursor left key
kL	Key for delete line
kM	Key for exit insert mode
kN	Key for next page
kP	Key for previous page

kr	Cursor right key
kR	Key for scrolling backward/up
ks	Turn keypad on
kS	Clear to end of screen key
kt	Clear this tab key
kT	Set tab here key
ku	Cursor up key
l0	Label of zeroth function key, if not f0
l1	Label of first function key, if not f1
l2	Label of first function key, if not f2
...	...
la	Label of tenth function key, if not f10
le	Cursor left one character
ll	Move cursor to lower left corner
LE	Cursor left %1 characters
LF	Turn soft labels off
LO	Turn soft labels on
mb	Start blinking
MC	Clear soft margins
md	Start bold mode
me	End all mode like so, us, mb, md, and mr
mh	Start half bright mode
mk	Dark mode (Characters invisible)
ML	Set left soft margin
mm	Put terminal in meta mode
mo	Put terminal out of meta mode
mp	Turn on protected attribute
mr	Start reverse mode
MR	Set right soft margin
nd	Cursor right one character
nw	Carriage return command
pc	Padding character
pf	Turn printer off
pk	Program key %1 to send string %2 as if typed by user
pl	Program key %1 to execute string %2 in local mode
pn	Program soft label %1 to show string %2
po	Turn the printer on
pO	Turn the printer on for %1 (<256) bytes
ps	Print screen contents on printer
px	Program key %1 to send string %2 to computer
r1	Reset string 1 to set terminal to sane modes
r2	Reset string 2 to set terminal to sane modes
r3	Reset string 3 to set terminal to sane modes
RA	disable automatic margins
rc	Restore saved cursor position
rf	Reset string filename
RF	Request for input from terminal
RI	Cursor right %1 characters
rp	Repeat character %1 for %2 times
rP	Padding after character sent in replace mode
rs	Reset string
RX	Turn off XON/XOFF flow control
sa	Set %1 %2 %3 %4 %5 %6 %7 %8 %9 attributes
SA	enable automatic margins
sc	Save cursor position
se	End standout mode
sf	Normal scroll one line
SF	Normal scroll %1 lines
so	Start standout mode

sr Reverse scroll
SR scroll back %1 lines
st Set tabulator stop in all rows at current column
SX Turn on XON/XOFF flow control
ta move to next hardware tab
tc Read in terminal description from another entry
te End program that uses cursor motion
ti Begin program that uses cursor motion
ts Move cursor to column %1 of status line
uc Underline character under cursor and move cursor right
ue End underlining
up Cursor up one line
UP Cursor up %1 lines
us Start underlining
vb Visible bell
ve Normal cursor visible
vi Cursor invisible
vs Standout cursor
wi Set window from line %1 to %2 and column %3 to %4
XF XOFF character if not **^S**

There are several ways of defining the control codes for string capabilities:

Every normal character represents itself, except '^', '\', and '%'.

A **^x** means Control-x. Control-A equals 1 decimal.

\x means a special code. x can be one of the following characters:
E Escape (27)
n Linefeed (10)
r Carriage return (13)
t Tabulation (9)
b Backspace (8)
f Form feed (12)
0 Null character. A \xxx specifies the octal character xxx.

i Increments parameters by one.

r Single parameter capability

\+ Add value of next character to this parameter and do binary output

2 Do ASCII output of this parameter with a field with of 2

d Do ASCII output of this parameter with a field with of 3

% Print a '%'

If you use binary output, then you should avoid the null character ('\0') because it terminates the string. You should reset tabulator expansion if a tabulator can be the binary output of a parameter.

Warning:
The above metacharacters for parameters may be wrong: they document Minix termcap which may not be compatible with Linux termcap.

The block graphic characters can be specified by three string capabilities:

as start the alternative charset

ae end the alternative charset

ac pairs of characters. The first character is the name of the block graphic symbol and the second characters is its definition.

The following names are available:

\+ right arrow (>)
, left arrow (<)
. down arrow (v)

0	full square (#)
I	lantern (#)
-	upper arrow (^)
’	rhombus (+)
a	chess board (:)
f	degree (’)
g	plus-minus (#)
h	square (#)
j	right bottom corner (+)
k	right upper corner (+)
l	left upper corner (+)
m	left bottom corner (+)
n	cross (+)
o	upper horizontal line (-)
q	middle horizontal line (-)
s	bottom horizontal line (_)
t	left tee (+)
u	right tee (+)
v	bottom tee (+)
w	normal tee (+)
x	vertical line (|)
~	paragraph (???)

The values in parentheses are suggested defaults which are used by the *curses* library, if the capabilities are missing.

SEE ALSO

ncurses(3), *termcap*(3), *terminfo*(5)

NAME

tmpfs − a virtual memory filesystem

DESCRIPTION

The **tmpfs** facility allows the creation of filesystems whose contents reside in virtual memory. Since the files on such filesystems typically reside in RAM, file access is extremely fast.

The filesystem is automatically created when mounting a filesystem with the type **tmpfs** via a command such as the following:

```
$ sudo mount −t tmpfs −o size=10M tmpfs /mnt/mytmpfs
```

A **tmpfs** filesystem has the following properties:

- The filesystem can employ swap space when physical memory pressure demands it.
- The filesystem consumes only as much physical memory and swap space as is required to store the current contents of the filesystem.
- During a remount operation (*mount −o remount*), the filesystem size can be changed (without losing the existing contents of the filesystem).

If a **tmpfs** filesystem is unmounted, its contents are discarded (lost).

Mount options

The **tmpfs** filesystem supports the following mount options:

size=*bytes*
: Specify an upper limit on the size of the filesystem. The size is given in bytes, and rounded up to entire pages. The limit is removed if the size is **0**.

  The size may have a **k**, **m**, or **g** suffix for Ki, Mi, Gi (binary kilo (kibi), binary mega (mebi), and binary giga (gibi)).

  The size may also have a % suffix to limit this instance to a percentage of physical RAM.

  The default, when neither **size** nor **nr_blocks** is specified, is *size=50%*.

nr_blocks=*blocks*
: The same as **size**, but in blocks of **PAGE_CACHE_SIZE**.

  Blocks may be specified with **k**, **m**, or **g** suffixes like **size**, but not a % suffix.

nr_inodes=*inodes*
: The maximum number of inodes for this instance. The default is half of the number of your physical RAM pages, or (on a machine with highmem) the number of lowmem RAM pages, whichever is smaller. The limit is removed if the number is **0**.

  Inodes may be specified with **k**, **m**, or **g** suffixes like **size**, but not a % suffix.

noswap(since Linux 6.4)
: Disables swap. Remounts must respect the original settings. By default swap is enabled.

mode=*mode*
: Set initial permissions of the root directory.

gid=*gid* (since Linux 2.5.7)
: Set the initial group ID of the root directory.

uid=*uid* (since Linux 2.5.7)
: Set the initial user ID of the root directory.

huge=*huge_option* (since Linux 4.7.0)
: Set the huge table memory allocation policy for all files in this instance (if **CONFIG_TRANSPARENT_HUGEPAGE** is enabled).

  The *huge_option* value is one of the following:

  never Do not allocate huge pages. This is the default.

  always Attempt to allocate huge pages every time a new page is needed.

within_size
Only allocate huge page if it will be fully within *i_size*. Also respect *fadvise*(2) and *madvise*(2) hints

advise Only allocate huge pages if requested with *fadvise*(2) or *madvise*(2).

deny For use in emergencies, to force the huge option off from all mounts.

force Force the huge option on for all mounts; useful for testing.

mpol=*mpol_option* (since Linux 2.6.15)
Set the NUMA memory allocation policy for all files in this instance (if **CONFIG_NUMA** is enabled).

The *mpol_option* value is one of the following:

default Use the process allocation policy (see *set_mempolicy*(2)).

prefer:*node*
Preferably allocate memory from the given *node*.

bind:*nodelist*
Allocate memory only from nodes in *nodelist*.

interleave
Allocate from each node in turn.

interleave:*nodelist*
Allocate from each node of *in* turn.

local Preferably allocate memory from the local node.

In the above, *nodelist* is a comma-separated list of decimal numbers and ranges that specify NUMA nodes. A range is a pair of hyphen-separated decimal numbers, the smallest and largest node numbers in the range. For example, *mpol=bind:0−3,5,7,9−15*.

VERSIONS

The **tmpfs** facility was added in Linux 2.4, as a successor to the older **ramfs** facility, which did not provide limit checking or allow for the use of swap space.

NOTES

In order for user-space tools and applications to create **tmpfs** filesystems, the kernel must be configured with the **CONFIG_TMPFS** option.

The **tmpfs** filesystem supports extended attributes (see *xattr*(7)), but *user* extended attributes are not permitted.

An internal shared memory filesystem is used for System V shared memory (**shmget**(2)) and shared anonymous mappings (**mmap**(2) with the **MAP_SHARED** and **MAP_ANONYMOUS** flags). This filesystem is available regardless of whether the kernel was configured with the **CONFIG_TMPFS** option.

A **tmpfs** filesystem mounted at */dev/shm* is used for the implementation of POSIX shared memory (**shm_overview**(7)) and POSIX semaphores (**sem_overview**(7)).

The amount of memory consumed by all **tmpfs** filesystems is shown in the *Shmem* field of */proc/meminfo* and in the *shared* field displayed by *free*(1)

The **tmpfs** facility was formerly called **shmfs**.

SEE ALSO

df(1), *du*(1), *memfd_create*(2), *mmap*(2), *set_mempolicy*(2), *shm_open*(3), *mount*(8)

The kernel source files *Documentation/filesystems/tmpfs.txt* and *Documentation/admin−guide/mm/transhuge.rst*.

NAME

ttytype − terminal device to default terminal type mapping

DESCRIPTION

The */etc/ttytype* file associates *termcap*(5) and *terminfo*(5) terminal type names with tty lines. Each line consists of a terminal type, followed by whitespace, followed by a tty name (a device name without the */dev/* prefix).

This association is used by the program *tset*(1) to set the environment variable **TERM** to the default terminal name for the user's current tty.

This facility was designed for a traditional time-sharing environment featuring character-cell terminals hardwired to a UNIX minicomputer. It is little used on modern workstation and personal UNIX systems.

FILES

/etc/ttytype
: the tty definitions file.

EXAMPLES

A typical */etc/ttytype* is:

```
con80x25 tty1
vt320 ttys0
```

SEE ALSO

termcap(5), *terminfo*(5), *agetty*(8), *mingetty*(8)

NAME

tzfile − timezone information

DESCRIPTION

The timezone information files used by *tzset*(3) are typically found under a directory with a name like */usr/share/zoneinfo*. These files use the format described in Internet RFC 8536. Each file is a sequence of 8-bit bytes. In a file, a binary integer is represented by a sequence of one or more bytes in network order (bigendian, or high-order byte first), with all bits significant, a signed binary integer is represented using two's complement, and a boolean is represented by a one-byte binary integer that is either 0 (false) or 1 (true). The format begins with a 44-byte header containing the following fields:

- The magic four-byte ASCII sequence "TZif" identifies the file as a timezone information file.
- A byte identifying the version of the file's format (as of 2021, either an ASCII NUL, "2", "3", or "4").
- Fifteen bytes containing zeros reserved for future use.
- Six four-byte integer values, in the following order:

  tzh_ttisutcnt
  The number of UT/local indicators stored in the file. (UT is Universal Time.)

  tzh_ttisstdcnt
  The number of standard/wall indicators stored in the file.

  tzh_leapcnt
  The number of leap seconds for which data entries are stored in the file.

  tzh_timecnt
  The number of transition times for which data entries are stored in the file.

  tzh_typecnt
  The number of local time types for which data entries are stored in the file (must not be zero).

  tzh_charcnt
  The number of bytes of time zone abbreviation strings stored in the file.

The above header is followed by the following fields, whose lengths depend on the contents of the header:

- **tzh_timecnt** four-byte signed integer values sorted in ascending order. These values are written in network byte order. Each is used as a transition time (as returned by *time*(2)) at which the rules for computing local time change.
- **tzh_timecnt** one-byte unsigned integer values; each one but the last tells which of the different types of local time types described in the file is associated with the time period starting with the same-indexed transition time and continuing up to but not including the next transition time. (The last time type is present only for consistency checking with the POSIX.1-2017-style TZ string described below.) These values serve as indices into the next field.
- **tzh_typecnt ttinfo** entries, each defined as follows:

```
struct ttinfo {
        int32_t         tt_utoff;
        unsigned char   tt_isdst;
        unsigned char   tt_desigidx;
};
```

  Each structure is written as a four-byte signed integer value for **tt_utoff**, in network byte order, followed by a one-byte boolean for **tt_isdst** and a one-byte value for **tt_desigidx**. In each structure, **tt_utoff** gives the number of seconds to be added to UT, **tt_isdst** tells whether **tm_isdst** should be set by *localtime*(3) and **tt_desigidx** serves as an index into the array of time zone abbreviation bytes that follow the **ttinfo** entries in the file; if the designated string is "-00", the **ttinfo** entry is a placeholder indicating that local time is unspecified. The **tt_utoff** value is never equal to −2**31, to let 32-bit clients negate it without overflow. Also, in realistic applications **tt_utoff** is in the range [−89999, 93599] (i.e., more than −25 hours and less than 26 hours); this allows easy support by implementations that already support the POSIX-required range

[−24:59:59, 25:59:59].

- **tzh_charcnt** bytes that represent time zone designations, which are null-terminated byte strings, each indexed by the **tt_desigidx** values mentioned above. The byte strings can overlap if one is a suffix of the other. The encoding of these strings is not specified.
- **tzh_leapcnt** pairs of four-byte values, written in network byte order; the first value of each pair gives the nonnegative time (as returned by *time*(2)) at which a leap second occurs or at which the leap second table expires; the second is a signed integer specifying the correction, which is the *total* number of leap seconds to be applied during the time period starting at the given time. The pairs of values are sorted in strictly ascending order by time. Each pair denotes one leap second, either positive or negative, except that if the last pair has the same correction as the previous one, the last pair denotes the leap second table's expiration time. Each leap second is at the end of a UTC calendar month. The first leap second has a nonnegative occurrence time, and is a positive leap second if and only if its correction is positive; the correction for each leap second after the first differs from the previous leap second by either 1 for a positive leap second, or −1 for a negative leap second. If the leap second table is empty, the leap-second correction is zero for all timestamps; otherwise, for timestamps before the first occurrence time, the leap-second correction is zero if the first pair's correction is 1 or −1, and is unspecified otherwise (which can happen only in files truncated at the start).
- **tzh_ttisstdcnt** standard/wall indicators, each stored as a one-byte boolean; they tell whether the transition times associated with local time types were specified as standard time or local (wall clock) time.
- **tzh_ttisutcnt** UT/local indicators, each stored as a one-byte boolean; they tell whether the transition times associated with local time types were specified as UT or local time. If a UT/local indicator is set, the corresponding standard/wall indicator must also be set.

The standard/wall and UT/local indicators were designed for transforming a TZif file's transition times into transitions appropriate for another time zone specified via a POSIX.1-2017-style TZ string that lacks rules. For example, when TZ="EET-2EEST" and there is no TZif file "EET-2EEST", the idea was to adapt the transition times from a TZif file with the well-known name "posixrules" that is present only for this purpose and is a copy of the file "Europe/Brussels", a file with a different UT offset. POSIX does not specify this obsolete transformational behavior, the default rules are installation-dependent, and no implementation is known to support this feature for timestamps past 2037, so users desiring (say) Greek time should instead specify TZ="Europe/Athens" for better historical coverage, falling back on TZ="EET-2EEST,M3.5.0/3,M10.5.0/4" if POSIX conformance is required and older timestamps need not be handled accurately.

The *localtime*(3) function normally uses the first **ttinfo** structure in the file if either **tzh_timecnt** is zero or the time argument is less than the first transition time recorded in the file.

Version 2 format

For version-2-format timezone files, the above header and data are followed by a second header and data, identical in format except that eight bytes are used for each transition time or leap second time. (Leap second counts remain four bytes.) After the second header and data comes a newline-enclosed string in the style of the contents of a POSIX.1-2017 TZ environment variable, for use in handling instants after the last transition time stored in the file or for all instants if the file has no transitions. The TZ string is empty (i.e., nothing between the newlines) if there is no POSIX.1-2017-style representation for such instants. If nonempty, the TZ string must agree with the local time type after the last transition time if present in the eight-byte data; for example, given the string "WET0WEST,M3.5.0/1,M10.5.0" then if a last transition time is in July, the transition's local time type must specify a daylight-saving time abbreviated "WEST" that is one hour east of UT. Also, if there is at least one transition, time type 0 is associated with the time period from the indefinite past up to but not including the earliest transition time.

Version 3 format

For version-3-format timezone files, the TZ string may use two minor extensions to the POSIX.1-2017 TZ format, as described in *newtzset*(3)First, the hours part of its transition times may be signed and range from −167 through 167 instead of the POSIX-required unsigned values from 0 through 24. Second, DST is in effect all year if it starts January 1 at 00:00 and ends December 31 at 24:00 plus the difference between daylight saving and standard time.

Version 4 format

For version-4-format TZif files, the first leap second record can have a correction that is neither +1 nor −1, to represent truncation of the TZif file at the start. Also, if two or more leap second transitions are present and the last entry's correction equals the previous one, the last entry denotes the expiration of the leap second table instead of a leap second; timestamps after this expiration are unreliable in that future releases will likely add leap second entries after the expiration, and the added leap seconds will change how post-expiration timestamps are treated.

Interoperability considerations

Future changes to the format may append more data.

Version 1 files are considered a legacy format and should not be generated, as they do not support transition times after the year 2038. Readers that understand only Version 1 must ignore any data that extends beyond the calculated end of the version 1 data block.

Other than version 1, writers should generate the lowest version number needed by a file's data. For example, a writer should generate a version 4 file only if its leap second table either expires or is truncated at the start. Likewise, a writer not generating a version 4 file should generate a version 3 file only if TZ string extensions are necessary to accurately model transition times.

The sequence of time changes defined by the version 1 header and data block should be a contiguous sub-sequence of the time changes defined by the version 2+ header and data block, and by the footer. This guideline helps obsolescent version 1 readers agree with current readers about timestamps within the contiguous sub-sequence. It also lets writers not supporting obsolescent readers use a **tzh_timecnt** of zero in the version 1 data block to save space.

When a TZif file contains a leap second table expiration time, TZif readers should either refuse to process post-expiration timestamps, or process them as if the expiration time did not exist (possibly with an error indication).

Time zone designations should consist of at least three (3) and no more than six (6) ASCII characters from the set of alphanumerics, "-", and "+". This is for compatibility with POSIX requirements for time zone abbreviations.

When reading a version 2 or higher file, readers should ignore the version 1 header and data block except for the purpose of skipping over them.

Readers should calculate the total lengths of the headers and data blocks and check that they all fit within the actual file size, as part of a validity check for the file.

When a positive leap second occurs, readers should append an extra second to the local minute containing the second just before the leap second. If this occurs when the UTC offset is not a multiple of 60 seconds, the leap second occurs earlier than the last second of the local minute and the minute's remaining local seconds are numbered through 60 instead of the usual 59; the UTC offset is unaffected.

Common interoperability issues

This section documents common problems in reading or writing TZif files. Most of these are problems in generating TZif files for use by older readers. The goals of this section are:

- to help TZif writers output files that avoid common pitfalls in older or buggy TZif readers,
- to help TZif readers avoid common pitfalls when reading files generated by future TZif writers, and
- to help any future specification authors see what sort of problems arise when the TZif format is changed.

When new versions of the TZif format have been defined, a design goal has been that a reader can successfully use a TZif file even if the file is of a later TZif version than what the reader was designed for. When complete compatibility was not achieved, an attempt was made to limit glitches to rarely used timestamps and allow simple partial workarounds in writers designed to generate new-version data useful even for older-version readers. This section attempts to document these compatibility issues and workarounds, as well as to document other common bugs in readers.

Interoperability problems with TZif include the following:

- Some readers examine only version 1 data. As a partial workaround, a writer can output as much version 1 data as possible. However, a reader should ignore version 1 data, and should use version 2+ data even if the reader's native timestamps have only 32 bits.
- Some readers designed for version 2 might mishandle timestamps after a version 3 or higher file's last transition, because they cannot parse extensions to POSIX.1-2017 in the TZ-like string. As a partial workaround, a writer can output more transitions than necessary, so that only far-future timestamps are mishandled by version 2 readers.
- Some readers designed for version 2 do not support permanent daylight saving time with transitions after 24:00 – e.g., a TZ string "EST5EDT,0/0,J365/25" denoting permanent Eastern Daylight Time (−04). As a workaround, a writer can substitute standard time for two time zones east, e.g., "XXX3EDT4,0/0,J365/23" for a time zone with a never-used standard time (XXX, −03) and negative daylight saving time (EDT, −04) all year. Alternatively, as a partial workaround a writer can substitute standard time for the next time zone east – e.g., "AST4" for permanent Atlantic Standard Time (−04).
- Some readers designed for version 2 or 3, and that require strict conformance to RFC 8536, reject version 4 files whose leap second tables are truncated at the start or that end in expiration times.
- Some readers ignore the footer, and instead predict future timestamps from the time type of the last transition. As a partial workaround, a writer can output more transitions than necessary.
- Some readers do not use time type 0 for timestamps before the first transition, in that they infer a time type using a heuristic that does not always select time type 0. As a partial workaround, a writer can output a dummy (no-op) first transition at an early time.
- Some readers mishandle timestamps before the first transition that has a timestamp not less than −2**31. Readers that support only 32-bit timestamps are likely to be more prone to this problem, for example, when they process 64-bit transitions only some of which are representable in 32 bits. As a partial workaround, a writer can output a dummy transition at timestamp −2**31.
- Some readers mishandle a transition if its timestamp has the minimum possible signed 64-bit value. Timestamps less than −2**59 are not recommended.
- Some readers mishandle TZ strings that contain "<" or ">". As a partial workaround, a writer can avoid using "<" or ">" for time zone abbreviations containing only alphabetic characters.
- Many readers mishandle time zone abbreviations that contain non-ASCII characters. These characters are not recommended.
- Some readers may mishandle time zone abbreviations that contain fewer than 3 or more than 6 characters, or that contain ASCII characters other than alphanumerics, "-", and "+". These abbreviations are not recommended.
- Some readers mishandle TZif files that specify daylight-saving time UT offsets that are less than the UT offsets for the corresponding standard time. These readers do not support locations like Ireland, which uses the equivalent of the TZ string "IST-1GMT0,M10.5.0,M3.5.0/1", observing standard time (IST, +01) in summer and daylight saving time (GMT, +00) in winter. As a partial workaround, a writer can output data for the equivalent of the TZ string "GMT0IST,M3.5.0/1,M10.5.0", thus swapping standard and daylight saving time. Although this workaround misidentifies which part of the year uses daylight saving time, it records UT offsets and time zone abbreviations correctly.
- Some readers generate ambiguous timestamps for positive leap seconds that occur when the UTC offset is not a multiple of 60 seconds. For example, in a timezone with UTC offset +01:23:45 and with a positive leap second 78796801 (1972-06-30 23:59:60 UTC), some readers will map both 78796800 and 78796801 to 01:23:45 local time the next day instead of mapping the latter to 01:23:46, and they will map 78796815 to 01:23:59 instead of to 01:23:60. This has not yet been a practical problem, since no civil authority has observed such UTC offsets since leap seconds were introduced in 1972.

Some interoperability problems are reader bugs that are listed here mostly as warnings to developers of readers.

- Some readers do not support negative timestamps. Developers of distributed applications should keep this in mind if they need to deal with pre-1970 data.
- Some readers mishandle timestamps before the first transition that has a nonnegative timestamp. Readers that do not support negative timestamps are likely to be more prone to this problem.
- Some readers mishandle time zone abbreviations like "-08" that contain "+", "-", or digits.
- Some readers mishandle UT offsets that are out of the traditional range of −12 through +12 hours, and so do not support locations like Kiritimati that are outside this range.
- Some readers mishandle UT offsets in the range [−3599, −1] seconds from UT, because they integer-divide the offset by 3600 to get 0 and then display the hour part as "+00".
- Some readers mishandle UT offsets that are not a multiple of one hour, or of 15 minutes, or of 1 minute.

SEE ALSO

time(2), *localtime*(3), *tzset*(3), *tzselect*(8), *zdump*(8), *zic*(8).

Olson A, Eggert P, Murchison K. The Time Zone Information Format (TZif). 2019 Feb. Internet RFC 8536 doi:10.17487/RFC8536.

NAME

utmp, wtmp − login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file allows one to discover information about who is currently using the system. There may be more users currently using the system, because not all programs use utmp logging.

Warning: *utmp* must not be writable by the user class "other", because many system programs (foolishly) depend on its integrity. You risk faked system logfiles and modifications of system files if you leave *utmp* writable to any user other than the owner and group owner of the file.

The file is a sequence of *utmp* structures, declared as follows in <*utmp.h*> (note that this is only one of several definitions around; details depend on the version of libc):

```
/* Values for ut_type field, below */

#define EMPTY         0 /* Record does not contain valid info
                           (formerly known as UT_UNKNOWN on Linux) */
#define RUN_LVL       1 /* Change in system run-level (see
                           init(1)) */
#define BOOT_TIME     2 /* Time of system boot (in ut_tv) */
#define NEW_TIME      3 /* Time after system clock change
                           (in ut_tv) */
#define OLD_TIME      4 /* Time before system clock change
                           (in ut_tv) */
#define INIT_PROCESS  5 /* Process spawned by init(1) */
#define LOGIN_PROCESS 6 /* Session leader process for user login */
#define USER_PROCESS  7 /* Normal process */
#define DEAD_PROCESS  8 /* Terminated process */
#define ACCOUNTING    9 /* Not implemented */

#define UT_LINESIZE      32
#define UT_NAMESIZE      32
#define UT_HOSTSIZE     256

struct exit_status {              /* Type for ut_exit, below */
    short e_termination;          /* Process termination status */
    short e_exit;                 /* Process exit status */
};

struct utmp {
    short   ut_type;              /* Type of record */
    pid_t   ut_pid;               /* PID of login process */
    char    ut_line[UT_LINESIZE]; /* Device name of tty - "/dev/" */
    char    ut_id[4];             /* Terminal name suffix,
                                     or inittab(5) ID */
    char    ut_user[UT_NAMESIZE]; /* Username */
    char    ut_host[UT_HOSTSIZE]; /* Hostname for remote login, or
                                     kernel version for run-level
                                     messages */
    struct  exit_status ut_exit;  /* Exit status of a process
                                     marked as DEAD_PROCESS; not
                                     used by Linux init(1) */
    /* The ut_session and ut_tv fields must be the same size when
       compiled 32- and 64-bit.  This allows data files and shared
       memory to be shared between 32- and 64-bit applications. */
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
    int32_t ut_session;           /* Session ID (getsid(2)),
                                     used for windowing */
```

```
    struct {
        int32_t tv_sec;         /* Seconds */
        int32_t tv_usec;        /* Microseconds */
    } ut_tv;                    /* Time entry was made */
#else
     long   ut_session;         /* Session ID */
     struct timeval ut_tv;      /* Time entry was made */
#endif

    int32_t ut_addr_v6[4];      /* Internet address of remote
                                   host; IPv4 address uses
                                   just ut_addr_v6[0] */
    char __unused[20];          /* Reserved for future use */
};

/* Backward compatibility hacks */
#define ut_name ut_user
#ifndef _NO_UT_TIME
#define ut_time ut_tv.tv_sec
#endif
#define ut_xtime ut_tv.tv_sec
#define ut_addr ut_addr_v6[0]
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of login in the form of *time*(2). String fields are terminated by a null byte ('\0') if they are shorter than the size of the field.

The first entries ever created result from *init*(1) processing *inittab*(5)Before an entry is processed, though, *init*(1) cleans up utmp by setting *ut_type* to **DEAD_PROCESS**, clearing *ut_user*, *ut_host*, and *ut_time* with null bytes for each record which *ut_type* is not **DEAD_PROCESS** or **RUN_LVL** and where no process with PID *ut_pid* exists. If no empty record with the needed *ut_id* can be found, *init*(1) creates a new one. It sets *ut_id* from the inittab, *ut_pid* and *ut_time* to the current values, and *ut_type* to **INIT_PROCESS**.

mingetty(8) (or *agetty*(8)) locates the entry by the PID, changes *ut_type* to **LOGIN_PROCESS**, changes *ut_time*, sets *ut_line*, and waits for connection to be established. *login*(1), after a user has been authenticated, changes *ut_type* to **USER_PROCESS**, changes *ut_time*, and sets *ut_host* and *ut_addr*. Depending on *mingetty*(8) (or *agetty*(8)) and *login*(1), records may be located by *ut_line* instead of the preferable *ut_pid*.

When *init*(1) finds that a process has exited, it locates its utmp entry by *ut_pid*, sets *ut_type* to **DEAD_PROCESS**, and clears *ut_user*, *ut_host*, and *ut_time* with null bytes.

xterm(1) and other terminal emulators directly create a **USER_PROCESS** record and generate the *ut_id* by using the string that suffix part of the terminal name (the characters following */dev/*[pt]*ty*). If they find a **DEAD_PROCESS** for this ID, they recycle it, otherwise they create a new entry. If they can, they will mark it as **DEAD_PROCESS** on exiting and it is advised that they null *ut_line*, *ut_time*, *ut_user*, and *ut_host* as well.

telnetd(8) sets up a **LOGIN_PROCESS** entry and leaves the rest to *login*(1) as usual. After the telnet session ends, *telnetd*(8) cleans up utmp in the described way.

The *wtmp* file records all logins and logouts. Its format is exactly like *utmp* except that a null username indicates a logout on the associated terminal. Furthermore, the terminal name ~ with username **shutdown** or **reboot** indicates a system shutdown or reboot and the pair of terminal names |/**}** logs the old/new system time when *date*(1) changes it. *wtmp* is maintained by *login*(1), *init*(1), and some versions of *getty*(8) (e.g., *mingetty*(8) or *agetty*(8)). None of these programs creates the file, so if it is removed, record-keeping is turned off.

FILES

/var/run/utmp
/var/log/wtmp

VERSIONS

POSIX.1 does not specify a *utmp* structure, but rather one named *utmpx* (as part of the XSI extension), with specifications for the fields *ut_type*, *ut_pid*, *ut_line*, *ut_id*, *ut_user*, and *ut_tv*. POSIX.1 does not specify the lengths of the *ut_line* and *ut_user* fields.

Linux defines the *utmpx* structure to be the same as the *utmp* structure.

STANDARDS

Linux.

HISTORY

Linux utmp entries conform neither to v7/BSD nor to System V; they are a mix of the two.

v7/BSD has fewer fields; most importantly it lacks *ut_type*, which causes native v7/BSD-like programs to display (for example) dead or login entries. Further, there is no configuration file which allocates slots to sessions. BSD does so because it lacks *ut_id* fields.

In Linux (as in System V), the *ut_id* field of a record will never change once it has been set, which reserves that slot without needing a configuration file. Clearing *ut_id* may result in race conditions leading to corrupted utmp entries and potential security holes. Clearing the abovementioned fields by filling them with null bytes is not required by System V semantics, but makes it possible to run many programs which assume BSD semantics and which do not modify utmp. Linux uses the BSD conventions for line contents, as documented above.

System V has no *ut_host* or *ut_addr_v6* fields.

NOTES

Unlike various other systems, where utmp logging can be disabled by removing the file, utmp must always exist on Linux. If you want to disable *who*(1), then do not make utmp world readable.

The file format is machine-dependent, so it is recommended that it be processed only on the machine architecture where it was created.

Note that on *biarch* platforms, that is, systems which can run both 32-bit and 64-bit applications (x86-64, ppc64, s390x, etc.), *ut_tv* is the same size in 32-bit mode as in 64-bit mode. The same goes for *ut_session* and *ut_time* if they are present. This allows data files and shared memory to be shared between 32-bit and 64-bit applications. This is achieved by changing the type of *ut_session* to *int32_t*, and that of *ut_tv* to a struct with two *int32_t* fields *tv_sec* and *tv_usec*. Since *ut_tv* may not be the same as *struct timeval*, then instead of the call:

```
gettimeofday((struct timeval *) &ut.ut_tv, NULL);
```

the following method of setting this field is recommended:

```
struct utmp ut;
struct timeval tv;

gettimeofday(&tv, NULL);
ut.ut_tv.tv_sec = tv.tv_sec;
ut.ut_tv.tv_usec = tv.tv_usec;
```

SEE ALSO

ac(1), *date*(1), *init*(1), *last*(1), *login*(1), *logname*(1), *lslogins*(1), *users*(1), *utmpdump*(1), *who*(1), *getutent*(3), *getutmp*(3), *login*(3), *logout*(3), *logwtmp*(3), *updwtmp*(3)

NAME

intro − introduction to games

DESCRIPTION

Section 6 of the manual describes the games and funny little programs available on the system.

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

NAME

intro − introduction to overview and miscellany section

DESCRIPTION

Section 7 of the manual provides overviews on various topics, and describes conventions and protocols, character set standards, the standard filesystem layout, and miscellaneous other things.

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

SEE ALSO

standards(7)

NAME

address_families − socket address families (domains)

SYNOPSIS

#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>

int socket(int *domain***, int** *type***, int** *protocol***);**

DESCRIPTION

The *domain* argument of the *socket*(2) specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <*sys/socket.h*>. The formats currently understood by the Linux kernel include:

AF_UNIX
AF_LOCAL
Local communication. For further information, see *unix*(7).

AF_INET
IPv4 Internet protocols. For further information, see *ip*(7).

AF_AX25
Amateur radio AX.25 protocol. For further information, see *ax25*(4)

AF_IPX
IPX − Novell protocols.

AF_APPLETALK
AppleTalk For further information, see *ddp*(7).

AF_NETROM
AX.25 packet layer protocol. For further information, see *netrom*(4), *The Packet Radio Protocols and Linux* and the *AX.25*, *NET/ROM*, and *ROSE network programming* chapters of the *Linux Amateur Radio AX.25 HOWTO*.

AF_BRIDGE
Can't be used for creating sockets; mostly used for bridge links in *rtnetlink*(7) protocol commands.

AF_ATMPVC
Access to raw ATM Permanent Virtual Circuits (PVCs). For further information, see the *ATM on Linux HOWTO*.

AF_X25
ITU-T X.25 / ISO/IEC 8208 protocol. For further information, see *x25*(7).

AF_INET6
IPv6 Internet protocols. For further information, see *ipv6*(7).

AF_ROSE
RATS (Radio Amateur Telecommunications Society). Open Systems environment (ROSE) AX.25 packet layer protocol. For further information, see the resources listed for **AF_NETROM**.

AF_DECnet
DECet protocol sockets. See *Documentation/networking/decnet.txt* in the Linux kernel source tree for details.

AF_NETBEUI
Reserved for "802.2LLC project"; never used.

AF_SECURITY
This was a short-lived (between Linux 2.1.30 and 2.1.99pre2) protocol family for firewall upcalls.

AF_KEY
Key management protocol, originally developed for usage with IPsec (since Linux 2.1.38). This has no relation to *keyctl*(2) and the in-kernel key storage facility. See RFC 2367 *PF_KEY Key Management API, Version 2* for details.

AF_NETLINK
Kernel user interface device. For further information, see *netlink*(7).

AF_PACKET
Low-level packet interface. For further information, see *packet*(7).

AF_ECONET
Acorn Econet protocol (removed in Linux 3.5). See the Econet documentation for details.

AF_ATMSVC
Access to ATM Switched Virtual Circuits (SVCs) See the *ATM on Linux HOWTO* for details.

AF_RDS
Reliable Datagram Sockets (RDS) protocol (since Linux 2.6.30). RDS over RDMA has no relation to **AF_SMC** or **AF_XDP**. For further information, see *rds*(7), *rds-rdma*(7), and *Documentation/networking/rds.txt* in the Linux kernel source tree.

AF_IRDA
Socket interface over IrDA (moved to staging in Linux 4.14, removed in Linux 4.17). For further information, see *irda*(7)

AF_PPPOX
Generic PPP transport layer, for setting up L2 tunnels (L2TP and PPPoE). See *Documentation/networking/l2tp.txt* in the Linux kernel source tree for details.

AF_WANPIPE
Legacy protocol for wide area network (WAN) connectivity that was used by Sangoma WAN cards (called "WANPIPE"); removed in Linux 2.6.21.

AF_LLC
Logical link control (IEEE 802.2 LLC) protocol, upper part of data link layer of ISO/OSI networking protocol stack (since Linux 2.4); has no relation to **AF_PACKET**. See chapter *13.5.3. Logical Link Control* in *Understanding Linux Kernel Internals* (O'Reilly Media, 2006) and *IEEE Standards for Local Area Networks: Logical Link Control* (The Institute of Electronics and Electronics Engineers, Inc., New York, New York, 1985) for details. See also some historical notes regarding its development.

AF_IB InfiniBand native addressing (since Linux 3.11).

AF_MPLS
Multiprotocol Label Switching (since Linux 4.1); mostly used for configuring MPLS routing via *netlink*(7), as it doesn't expose ability to create sockets to user space.

AF_CAN
Controller Area Network automotive bus protocol (since Linux 2.6.25). See *Documentation/networking/can.rst* in the Linux kernel source tree for details.

AF_TIPC
TIPC, "cluster domain sockets" protocol (since Linux 2.6.16). See *TIPC Programmer's Guide* and the protocol description for details.

AF_BLUETOOTH
Bluetooth low-level socket protocol (since Linux 3.11). See *Bluetooth Management API overview* and *An Introduction to Bluetooth Programming* by Albert Huang for details.

AF_IUCV
IUCV (inter-user communication vehicle) z/VM protocol for hypervisor-guest interaction (since Linux 2.6.21); has no relation to **AF_VSOCK** and/or **AF_SMC** See *IUCV protocol overview* for details.

AF_RXRPC
Rx, Andrew File System remote procedure call protocol (since Linux 2.6.22). See *Documentation/networking/rxrpc.txt* in the Linux kernel source tree for details.

AF_ISDN
New "modular ISDN" driver interface protocol (since Linux 2.6.27). See the mISDN wiki for details.

AF_PHONET
Nokia cellular modem IPC/RPC interface (since Linux 2.6.31). See *Documentation/networking/phonet.txt* in the Linux kernel source tree for details.

AF_IEEE802154
IEEE 802.15.4 WPAN (wireless personal area network) raw packet protocol (since Linux 2.6.31). See *Documentation/networking/ieee802154.txt* in the Linux kernel source tree for details.

AF_CAIF
Ericsson's Communication CPU to Application CPU interface (CAIF) protocol (since Linux 2.6.36). See *Documentation/networking/caif/Linux−CAIF.txt* in the Linux kernel source tree for details.

AF_ALG
Interface to kernel crypto API (since Linux 2.6.38). See *Documentation/crypto/userspace−if.rst* in the Linux kernel source tree for details.

AF_VSOCK
VMWare VSockets protocol for hypervisor-guest interaction (since Linux 3.9); has no relation to **AF_IUCV** and **AF_SMC**. For further information, see *vsock*(7).

AF_KCM
KCM (kernel connection multiplexer) interface (since Linux 4.6). See *Documentation/networking/kcm.txt* in the Linux kernel source tree for details.

AF_QIPCRTR
Qualcomm IPC router interface protocol (since Linux 4.7).

AF_SMC
SMC-R (shared memory communications over RDMA) protocol (since Linux 4.11), and SMC-D (shared memory communications, direct memory access) protocol for intra-node z/VM quest interaction (since Linux 4.19); has no relation to **AF_RDS**, **AF_IUCV** or **AF_VSOCK**. See RFC 7609 *IBM's Shared Memory Communications over RDMA (SMC-R) Protocol* for details regarding SMC-R. See *SMC-D Reference Information* for details regarding SMC-D.

AF_XDP
XDP (express data path) interface (since Linux 4.18). See *Documentation/networking/af_xdp.rst* in the Linux kernel source tree for details.

SEE ALSO
socket(2), *socket*(7)

NAME

aio − POSIX asynchronous I/O overview

DESCRIPTION

The POSIX asynchronous I/O (AIO) interface allows applications to initiate one or more I/O operations that are performed asynchronously (i.e., in the background). The application can elect to be notified of completion of the I/O operation in a variety of ways: by delivery of a signal, by instantiation of a thread, or no notification at all.

The POSIX AIO interface consists of the following functions:

aio_read(3)
: Enqueue a read request. This is the asynchronous analog of *read*(2).

aio_write(3)
: Enqueue a write request. This is the asynchronous analog of *write*(2).

aio_fsync(3)
: Enqueue a sync request for the I/O operations on a file descriptor. This is the asynchronous analog of *fsync*(2) and *fdatasync*(2).

aio_error(3)
: Obtain the error status of an enqueued I/O request.

aio_return(3)
: Obtain the return status of a completed I/O request.

aio_suspend(3)
: Suspend the caller until one or more of a specified set of I/O requests completes.

aio_cancel(3)
: Attempt to cancel outstanding I/O requests on a specified file descriptor.

lio_listio(3)
: Enqueue multiple I/O requests using a single function call.

The *aiocb* ("asynchronous I/O control block") structure defines parameters that control an I/O operation. An argument of this type is employed with all of the functions listed above. This structure has the following form:

```
#include <aiocb.h>

struct aiocb {
    /* The order of these fields is implementation-dependent */

    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;   /* Notification method */
    int             aio_lio_opcode; /* Operation to be performed;
                                       lio_listio() only */

    /* Various implementation-internal fields not shown */
};

/* Operation codes for 'aio_lio_opcode': */

enum { LIO_READ, LIO_WRITE, LIO_NOP };
```

The fields of this structure are as follows:

aio_fildes
: The file descriptor on which the I/O operation is to be performed.

aio_offset
: This is the file offset at which the I/O operation is to be performed.

aio_buf
: This is the buffer used to transfer data for a read or write operation.

aio_nbytes
: This is the size of the buffer pointed to by *aio_buf*.

aio_reqprio
: This field specifies a value that is subtracted from the calling thread's real-time priority in order to determine the priority for execution of this I/O request (see *pthread_setschedparam*(3)). The specified value must be between 0 and the value returned by *sysconf(_SC_AIO_PRIO_DELTA_MAX)*. This field is ignored for file synchronization operations.

aio_sigevent
: This field is a structure that specifies how the caller is to be notified when the asynchronous I/O operation completes. Possible values for *aio_sigevent.sigev_notify* are **SIGEV_NONE**, **SIGEV_SIGNAL**, and **SIGEV_THREAD**. See *sigevent*(3type) for further details.

aio_lio_opcode
: The type of operation to be performed; used only for *lio_listio*(3).

In addition to the standard functions listed above, the GNU C library provides the following extension to the POSIX AIO API:

aio_init(3)
: Set parameters for tuning the behavior of the glibc POSIX AIO implementation.

ERRORS

EINVAL
: The *aio_reqprio* field of the *aiocb* structure was less than 0, or was greater than the limit returned by the call *sysconf(_SC_AIO_PRIO_DELTA_MAX)*.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001. glibc 2.1.

NOTES

It is a good idea to zero out the control block buffer before use (see *memset*(3)). The control block buffer and the buffer pointed to by *aio_buf* must not be changed while the I/O operation is in progress. These buffers must remain valid until the I/O operation completes.

Simultaneous asynchronous read or write operations using the same *aiocb* structure yield undefined results.

The current Linux POSIX AIO implementation is provided in user space by glibc. This has a number of limitations, most notably that maintaining multiple threads to perform I/O operations is expensive and scales poorly. Work has been in progress for some time on a kernel state-machine-based implementation of asynchronous I/O (see *io_submit*(2), *io_setup*(2), *io_cancel*(2), *io_destroy*(2), *io_getevents*(2)), but this implementation hasn't yet matured to the point where the POSIX AIO implementation can be completely reimplemented using the kernel system calls.

EXAMPLES

The program below opens each of the files named in its command-line arguments and queues a request on the resulting file descriptor using *aio_read*(3). The program then loops, periodically monitoring each of the I/O operations that is still in progress using *aio_error*(3). Each of the I/O requests is set up to provide notification by delivery of a signal. After all I/O requests have completed, the program retrieves their status using *aio_return*(3).

The **SIGQUIT** signal (generated by typing control-\) causes the program to request cancelation of each of the outstanding requests using *aio_cancel*(3).

Here is an example of what we might see when running this program. In this example, the program queues two requests to standard input, and these are satisfied by two lines of input containing "abc" and

"x".

```
$ ./a.out /dev/stdin /dev/stdin
opened /dev/stdin on descriptor 3
opened /dev/stdin on descriptor 4
aio_error():
    for request 0 (descriptor 3): In progress
    for request 1 (descriptor 4): In progress
abc
I/O completion signal received
aio_error():
    for request 0 (descriptor 3): I/O succeeded
    for request 1 (descriptor 4): In progress
aio_error():
    for request 1 (descriptor 4): In progress
x
I/O completion signal received
aio_error():
    for request 1 (descriptor 4): I/O succeeded
All I/O requests completed
aio_return():
    for request 0 (descriptor 3): 4
    for request 1 (descriptor 4): 2
```

Program source

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <aio.h>
#include <signal.h>

#define BUF_SIZE 20     /* Size of buffers for read operations */

#define errExit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)

struct ioRequest {      /* Application-defined structure for tracking
                           I/O requests */
    int           reqNum;
    int           status;
    struct aiocb *aiocbp;
};

static volatile sig_atomic_t gotSIGQUIT = 0;
                        /* On delivery of SIGQUIT, we attempt to
                           cancel all outstanding I/O requests */

static void             /* Handler for SIGQUIT */
quitHandler(int sig)
{
    gotSIGQUIT = 1;
}

#define IO_SIGNAL SIGUSR1   /* Signal used to notify I/O completion */

static void                 /* Handler for I/O completion signal */
aioSigHandler(int sig, siginfo_t *si, void *ucontext)
{
```

```
        if (si->si_code == SI_ASYNCIO) {
            write(STDOUT_FILENO, "I/O completion signal received\n", 31);

            /* The corresponding ioRequest structure would be available as
                   struct ioRequest *ioReq = si->si_value.sival_ptr;
               and the file descriptor would then be available via
                   ioReq->aiocbp->aio_fildes */
        }
    }

    int
    main(int argc, char *argv[])
    {
        struct sigaction sa;
        int s;
        int numReqs;        /* Total number of queued I/O requests */
        int openReqs;       /* Number of I/O requests still in progress */

        if (argc < 2) {
            fprintf(stderr, "Usage: %s <pathname> <pathname>...\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }

        numReqs = argc - 1;

        /* Allocate our arrays. */

        struct ioRequest *ioList = calloc(numReqs, sizeof(*ioList));
        if (ioList == NULL)
            errExit("calloc");

        struct aiocb *aiocbList = calloc(numReqs, sizeof(*aiocbList));
        if (aiocbList == NULL)
            errExit("calloc");

        /* Establish handlers for SIGQUIT and the I/O completion signal. */

        sa.sa_flags = SA_RESTART;
        sigemptyset(&sa.sa_mask);

        sa.sa_handler = quitHandler;
        if (sigaction(SIGQUIT, &sa, NULL) == -1)
            errExit("sigaction");

        sa.sa_flags = SA_RESTART | SA_SIGINFO;
        sa.sa_sigaction = aioSigHandler;
        if (sigaction(IO_SIGNAL, &sa, NULL) == -1)
            errExit("sigaction");

        /* Open each file specified on the command line, and queue
           a read request on the resulting file descriptor. */

        for (size_t j = 0; j < numReqs; j++) {
            ioList[j].reqNum = j;
            ioList[j].status = EINPROGRESS;
            ioList[j].aiocbp = &aiocbList[j];

            ioList[j].aiocbp->aio_fildes = open(argv[j + 1], O_RDONLY);
```

```
        if (ioList[j].aiocbp->aio_fildes == -1)
            errExit("open");
        printf("opened %s on descriptor %d\n", argv[j + 1],
                ioList[j].aiocbp->aio_fildes);

        ioList[j].aiocbp->aio_buf = malloc(BUF_SIZE);
        if (ioList[j].aiocbp->aio_buf == NULL)
            errExit("malloc");

        ioList[j].aiocbp->aio_nbytes = BUF_SIZE;
        ioList[j].aiocbp->aio_reqprio = 0;
        ioList[j].aiocbp->aio_offset = 0;
        ioList[j].aiocbp->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        ioList[j].aiocbp->aio_sigevent.sigev_signo = IO_SIGNAL;
        ioList[j].aiocbp->aio_sigevent.sigev_value.sival_ptr =
                                &ioList[j];

        s = aio_read(ioList[j].aiocbp);
        if (s == -1)
            errExit("aio_read");
    }

    openReqs = numReqs;

    /* Loop, monitoring status of I/O requests. */

    while (openReqs > 0) {
        sleep(3);       /* Delay between each monitoring step */

        if (gotSIGQUIT) {

            /* On receipt of SIGQUIT, attempt to cancel each of the
               outstanding I/O requests, and display status returned
               from the cancelation requests. */

            printf("got SIGQUIT; canceling I/O requests: \n");

            for (size_t j = 0; j < numReqs; j++) {
                if (ioList[j].status == EINPROGRESS) {
                    printf("    Request %zu on descriptor %d:", j,
                            ioList[j].aiocbp->aio_fildes);
                    s = aio_cancel(ioList[j].aiocbp->aio_fildes,
                            ioList[j].aiocbp);
                    if (s == AIO_CANCELED)
                        printf("I/O canceled\n");
                    else if (s == AIO_NOTCANCELED)
                        printf("I/O not canceled\n");
                    else if (s == AIO_ALLDONE)
                        printf("I/O all done\n");
                    else
                        perror("aio_cancel");
                }
            }

            gotSIGQUIT = 0;
        }

        /* Check the status of each I/O request that is still
           in progress. */
```

```
            printf("aio_error():\n");
            for (size_t j = 0; j < numReqs; j++) {
                if (ioList[j].status == EINPROGRESS) {
                    printf("    for request %zu (descriptor %d): ",
                            j, ioList[j].aiocbp->aio_fildes);
                    ioList[j].status = aio_error(ioList[j].aiocbp);

                    switch (ioList[j].status) {
                    case 0:
                        printf("I/O succeeded\n");
                        break;
                    case EINPROGRESS:
                        printf("In progress\n");
                        break;
                    case ECANCELED:
                        printf("Canceled\n");
                        break;
                    default:
                        perror("aio_error");
                        break;
                    }

                    if (ioList[j].status != EINPROGRESS)
                        openReqs--;
                }
            }
        }

        printf("All I/O requests completed\n");

        /* Check status return of all I/O requests. */

        printf("aio_return():\n");
        for (size_t j = 0; j < numReqs; j++) {
            ssize_t s;

            s = aio_return(ioList[j].aiocbp);
            printf("    for request %zu (descriptor %d): %zd\n",
                    j, ioList[j].aiocbp->aio_fildes, s);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

io_cancel(2), *io_destroy*(2), *io_getevents*(2), *io_setup*(2), *io_submit*(2), *aio_cancel*(3), *aio_error*(3), *aio_init*(3), *aio_read*(3), *aio_return*(3), *aio_write*(3), *lio_listio*(3)

"Asynchronous I/O Support in Linux 2.5", Bhattacharya, Pratt, Pulavarty, and Morgan, Proceedings of the Linux Symposium, 2003,

NAME

armscii-8 − Armenian character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The Armenian Standard Code for Information Interchange, 8-bit coded character set.

ArmSCII-8 characters

The following table displays the characters in ArmSCII-8 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
242	162	A2		ARMENIAN SMALL LIGATURE ECH YIWN
243	163	A3		ARMENIAN FULL STOP
244	164	A4	)	RIGHT PARENTHESIS
245	165	A5	(	LEFT PARENTHESIS
246	166	A6	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
247	167	A7	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
250	168	A8	—	EM DASH
251	169	A9	.	FULL STOP
252	170	AA		ARMENIAN COMMA
253	171	AB	,	COMMA
254	172	AC	-	HYPHEN-MINUS
255	173	AD		ARMENIAN HYPHEN
256	174	AE	…	HORIZONTAL ELLIPSIS
257	175	AF		ARMENIAN EXCLAMATION MARK
260	176	B0		ARMENIAN EMPHASIS MARK
261	177	B1		ARMENIAN QUESTION MARK
262	178	B2		ARMENIAN CAPITAL LETTER AYB
263	179	B3		ARMENIAN SMALL LETTER AYB
264	180	B4		ARMENIAN CAPITAL LETTER BEN
265	181	B5		ARMENIAN SMALL LETTER BEN
266	182	B6		ARMENIAN CAPITAL LETTER GIM
267	183	B7		ARMENIAN SMALL LETTER GIM
270	184	B8		ARMENIAN CAPITAL LETTER DA
271	185	B9		ARMENIAN SMALL LETTER DA
272	186	BA		ARMENIAN CAPITAL LETTER ECH
273	187	BB		ARMENIAN SMALL LETTER ECH
274	188	BC		ARMENIAN CAPITAL LETTER ZA
275	189	BD		ARMENIAN SMALL LETTER ZA
276	190	BE		ARMENIAN CAPITAL LETTER EH
277	191	BF		ARMENIAN SMALL LETTER EH
300	192	C0		ARMENIAN CAPITAL LETTER ET
301	193	C1		ARMENIAN SMALL LETTER ET
302	194	C2		ARMENIAN CAPITAL LETTER TO
303	195	C3		ARMENIAN SMALL LETTER TO
304	196	C4		ARMENIAN CAPITAL LETTER ZHE
305	197	C5		ARMENIAN SMALL LETTER ZHE
306	198	C6		ARMENIAN CAPITAL LETTER INI
307	199	C7		ARMENIAN SMALL LETTER INI
310	200	C8		ARMENIAN CAPITAL LETTER LIWN
311	201	C9		ARMENIAN SMALL LETTER LIWN
312	202	CA		ARMENIAN CAPITAL LETTER XEH
313	203	CB		ARMENIAN SMALL LETTER XEH
314	204	CC		ARMENIAN CAPITAL LETTER CA
315	205	CD		ARMENIAN SMALL LETTER CA
316	206	CE		ARMENIAN CAPITAL LETTER KEN
317	207	CF		ARMENIAN SMALL LETTER KEN
320	208	D0		ARMENIAN CAPITAL LETTER HO
321	209	D1		ARMENIAN SMALL LETTER HO

322	210	D2	ARMENIAN CAPITAL LETTER JA
323	211	D3	ARMENIAN SMALL LETTER JA
324	212	D4	ARMENIAN CAPITAL LETTER GHAD
325	213	D5	ARMENIAN SMALL LETTER GHAD
326	214	D6	ARMENIAN CAPITAL LETTER CHEH
327	215	D7	ARMENIAN SMALL LETTER CHEH
330	216	D8	ARMENIAN CAPITAL LETTER MEN
331	217	D9	ARMENIAN SMALL LETTER MEN
332	218	DA	ARMENIAN CAPITAL LETTER YI
333	219	DB	ARMENIAN SMALL LETTER YI
334	220	DC	ARMENIAN CAPITAL LETTER NOW
335	221	DD	ARMENIAN SMALL LETTER NOW
336	222	DE	ARMENIAN CAPITAL LETTER SHA
337	223	DF	ARMENIAN SMALL LETTER SHA
340	224	E0	ARMENIAN CAPITAL LETTER VO
341	225	E1	ARMENIAN SMALL LETTER VO
342	226	E2	ARMENIAN CAPITAL LETTER CHA
343	227	E3	ARMENIAN SMALL LETTER CHA
344	228	E4	ARMENIAN CAPITAL LETTER PEH
345	229	E5	ARMENIAN SMALL LETTER PEH
346	230	E6	ARMENIAN CAPITAL LETTER JHEH
347	231	E7	ARMENIAN SMALL LETTER JHEH
350	232	E8	ARMENIAN CAPITAL LETTER RA
351	233	E9	ARMENIAN SMALL LETTER RA
352	234	EA	ARMENIAN CAPITAL LETTER SEH
353	235	EB	ARMENIAN SMALL LETTER SEH
354	236	EC	ARMENIAN CAPITAL LETTER VEW
355	237	ED	ARMENIAN SMALL LETTER VEW
356	238	EE	ARMENIAN CAPITAL LETTER TIWN
357	239	EF	ARMENIAN SMALL LETTER TIWN
360	240	F0	ARMENIAN CAPITAL LETTER REH
361	241	F1	ARMENIAN SMALL LETTER REH
362	242	F2	ARMENIAN CAPITAL LETTER CO
363	243	F3	ARMENIAN SMALL LETTER CO
364	244	F4	ARMENIAN CAPITAL LETTER YIWN
365	245	F5	ARMENIAN SMALL LETTER YIWN
366	246	F6	ARMENIAN CAPITAL LETTER PIWR
367	247	F7	ARMENIAN SMALL LETTER PIWR
370	248	F8	ARMENIAN CAPITAL LETTER KEH
371	249	F9	ARMENIAN SMALL LETTER KEH
372	250	FA	ARMENIAN CAPITAL LETTER OH
373	251	FB	ARMENIAN SMALL LETTER OH
374	252	FC	ARMENIAN CAPITAL LETTER FEH
375	253	FD	ARMENIAN SMALL LETTER FEH
376	254	FE	ARMENIAN APOSTROPHE

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

arp − Linux ARP kernel module.

DESCRIPTION

This kernel protocol module implements the Address Resolution Protocol defined in RFC 826. It is used to convert between Layer2 hardware addresses and IPv4 protocol addresses on directly connected networks. The user normally doesn't interact directly with this module except to configure it; instead it provides a service for other protocols in the kernel.

A user process can receive ARP packets by using *packet*(7) sockets. There is also a mechanism for managing the ARP cache in user-space by using *netlink*(7) sockets. The ARP table can also be controlled via *ioctl*(2) on any **AF_INET** socket.

The ARP module maintains a cache of mappings between hardware addresses and protocol addresses. The cache has a limited size so old and less frequently used entries are garbage-collected. Entries which are marked as permanent are never deleted by the garbage-collector. The cache can be directly manipulated by the use of ioctls and its behavior can be tuned by the */proc* interfaces described below.

When there is no positive feedback for an existing mapping after some time (see the */proc* interfaces below), a neighbor cache entry is considered stale. Positive feedback can be gotten from a higher layer; for example from a successful TCP ACK. Other protocols can signal forward progress using the **MSG_CONFIRM** flag to *sendmsg*(2). When there is no forward progress, ARP tries to reprobe. It first tries to ask a local arp daemon **app_solicit** times for an updated MAC address. If that fails and an old MAC address is known, a unicast probe is sent **ucast_solicit** times. If that fails too, it will broadcast a new ARP request to the network. Requests are sent only when there is data queued for sending.

Linux will automatically add a nonpermanent proxy arp entry when it receives a request for an address it forwards to and proxy arp is enabled on the receiving interface. When there is a reject route for the target, no proxy arp entry is added.

Ioctls

Three ioctls are available on all **AF_INET** sockets. They take a pointer to a *struct arpreq* as their argument.

```
struct arpreq {
    struct sockaddr arp_pa;      /* protocol address */
    struct sockaddr arp_ha;      /* hardware address */
    int             arp_flags;   /* flags */
    struct sockaddr arp_netmask; /* netmask of protocol address */
    char            arp_dev[16];
};
```

SIOCSARP, **SIOCDARP** and **SIOCGARP** respectively set, delete, and get an ARP mapping. Setting and deleting ARP maps are privileged operations and may be performed only by a process with the **CAP_NET_ADMIN** capability or an effective UID of 0.

arp_pa must be an **AF_INET** address and *arp_ha* must have the same type as the device which is specified in *arp_dev*. *arp_dev* is a zero-terminated string which names a device.

arp_flags	
flag	meaning
ATF_COM	Lookup complete
ATF_PERM	Permanent entry
ATF_PUBL	Publish entry
ATF_USETRAILERS	Trailers requested
ATF_NETMASK	Use a netmask
ATF_DONTPUB	Don't answer

If the **ATF_NETMASK** flag is set, then *arp_netmask* should be valid. Linux 2.2 does not support proxy network ARP entries, so this should be set to 0xffffffff, or 0 to remove an existing proxy arp entry. **ATF_USETRAILERS** is obsolete and should not be used.

/proc interfaces

ARP supports a range of */proc* interfaces to configure parameters on a global or per-interface basis. The interfaces can be accessed by reading or writing the */proc/sys/net/ipv4/neigh/*/** files. Each

interface in the system has its own directory in */proc/sys/net/ipv4/neigh/*. The setting in the "default" directory is used for all newly created devices. Unless otherwise specified, time-related interfaces are specified in seconds.

anycast_delay (since Linux 2.2)
: The maximum number of jiffies to delay before replying to a IPv6 neighbor solicitation message. Anycast support is not yet implemented. Defaults to 1 second.

app_solicit (since Linux 2.2)
: The maximum number of probes to send to the user space ARP daemon via netlink before dropping back to multicast probes (see *mcast_solicit*). Defaults to 0.

base_reachable_time (since Linux 2.2)
: Once a neighbor has been found, the entry is considered to be valid for at least a random value between *base_reachable_time*/2 and 3**base_reachable_time*/2. An entry's validity will be extended if it receives positive feedback from higher level protocols. Defaults to 30 seconds. This file is now obsolete in favor of *base_reachable_time_ms*.

base_reachable_time_ms (since Linux 2.6.12)
: As for *base_reachable_time*, but measures time in milliseconds. Defaults to 30000 milliseconds.

delay_first_probe_time (since Linux 2.2)
: Delay before first probe after it has been decided that a neighbor is stale. Defaults to 5 seconds.

gc_interval (since Linux 2.2)
: How frequently the garbage collector for neighbor entries should attempt to run. Defaults to 30 seconds.

gc_stale_time (since Linux 2.2)
: Determines how often to check for stale neighbor entries. When a neighbor entry is considered stale, it is resolved again before sending data to it. Defaults to 60 seconds.

gc_thresh1 (since Linux 2.2)
: The minimum number of entries to keep in the ARP cache. The garbage collector will not run if there are fewer than this number of entries in the cache. Defaults to 128.

gc_thresh2 (since Linux 2.2)
: The soft maximum number of entries to keep in the ARP cache. The garbage collector will allow the number of entries to exceed this for 5 seconds before collection will be performed. Defaults to 512.

gc_thresh3 (since Linux 2.2)
: The hard maximum number of entries to keep in the ARP cache. The garbage collector will always run if there are more than this number of entries in the cache. Defaults to 1024.

locktime (since Linux 2.2)
: The minimum number of jiffies to keep an ARP entry in the cache. This prevents ARP cache thrashing if there is more than one potential mapping (generally due to network misconfiguration). Defaults to 1 second.

mcast_solicit (since Linux 2.2)
: The maximum number of attempts to resolve an address by multicast/broadcast before marking the entry as unreachable. Defaults to 3.

proxy_delay (since Linux 2.2)
: When an ARP request for a known proxy-ARP address is received, delay up to *proxy_delay* jiffies before replying. This is used to prevent network flooding in some cases. Defaults to 0.8 seconds.

proxy_qlen (since Linux 2.2)
: The maximum number of packets which may be queued to proxy-ARP addresses. Defaults to 64.

retrans_time (since Linux 2.2)
: The number of jiffies to delay before retransmitting a request. Defaults to 1 second. This file is now obsolete in favor of *retrans_time_ms*.

retrans_time_ms (since Linux 2.6.12)
: The number of milliseconds to delay before retransmitting a request. Defaults to 1000 milliseconds.

ucast_solicit (since Linux 2.2)
: The maximum number of attempts to send unicast probes before asking the ARP daemon (see *app_solicit*). Defaults to 3.

unres_qlen (since Linux 2.2)
: The maximum number of packets which may be queued for each unresolved address by other network layers. Defaults to 3.

VERSIONS

The *struct arpreq* changed in Linux 2.0 to include the *arp_dev* member and the ioctl numbers changed at the same time. Support for the old ioctls was dropped in Linux 2.2.

Support for proxy arp entries for networks (netmask not equal 0xffffffff) was dropped in Linux 2.2. It is replaced by automatic proxy arp setup by the kernel for all reachable hosts on other interfaces (when forwarding and proxy arp is enabled for the interface).

The *neigh/** interfaces did not exist before Linux 2.2.

BUGS

Some timer settings are specified in jiffies, which is architecture- and kernel version-dependent; see *time*(7).

There is no way to signal positive feedback from user space. This means connection-oriented protocols implemented in user space will generate excessive ARP traffic, because ndisc will regularly reprobe the MAC address. The same problem applies for some kernel protocols (e.g., NFS over UDP).

This man page mashes together functionality that is IPv4-specific with functionality that is shared between IPv4 and IPv6.

SEE ALSO

capabilities(7), *ip*(7), *arpd*(8)

RFC 826 for a description of ARP. RFC 2461 for a description of IPv6 neighbor discovery and the base algorithms used. Linux 2.2+ IPv4 ARP uses the IPv6 algorithms when applicable.

NAME

ascii − ASCII character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (e.g., ISO/IEC 8859-1) contain ASCII as their lower half. The international counterpart of ASCII is known as ISO/IEC 646-IRV.

The following table contains the 128 ASCII characters.

C program '\X' escapes are noted.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0' (null character)	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ '\\'
035	29	1D	GS (group separator)	135	93	5D	]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	−	155	109	6D	m
056	46	2E	.	156	110	6E	n

```
057   47    2F    /
060   48    30    0
061   49    31    1
062   50    32    2
063   51    33    3
064   52    34    4
065   53    35    5
066   54    36    6
067   55    37    7
070   56    38    8
071   57    39    9
072   58    3A    :
073   59    3B    ;
074   60    3C    <
075   61    3D    =
076   62    3E    >
077   63    3F    ?
```

```
157   111   6F    o
160   112   70    p
161   113   71    q
162   114   72    r
163   115   73    s
164   116   74    t
165   117   75    u
166   118   76    v
167   119   77    w
170   120   78    x
171   121   79    y
172   122   7A    z
173   123   7B    {
174   124   7C    |
175   125   7D    }
176   126   7E    ~
177   127   7F    DEL
```

Tables

For convenience, below are more compact tables in hex and decimal.

```
   2 3 4 5 6 7
 ------------
0:   0 @ P ` p
1: ! 1 A Q a q
2: " 2 B R b r
3: # 3 C S c s
4: $ 4 D T d t
5: % 5 E U e u
6: & 6 F V f v
7: ' 7 G W g w
8: ( 8 H X h x
9: ) 9 I Y i y
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

```
  30 40 50 60 70 80 90 100 110 120
 --------------------------------
0:    (  2  <  F  P  Z  d   n   x
1:    )  3  =  G  Q  [  e   o   y
2:    *  4  >  H  R  \  f   p   z
3: !  +  5  ?  I  S  ]  g   q   {
4: "  ,  6  @  J  T  ^  h   r   |
5: #  -  7  A  K  U  _  i   s   }
6: $  .  8  B  L  V  `  j   t   ~
7: %  /  9  C  M  W  a  k   u  DEL
8: &  0  :  D  N  X  b  l   v
9: '  1  ;  E  O  Y  c  m   w
```

NOTES

History

/etc/ascii (VII) appears in the UNIX Programmer's Manual.

On older terminals, the underscore code is displayed as a left arrow, called backarrow, the caret is displayed as an up-arrow and the vertical bar has a hole in the middle.

Uppercase and lowercase characters differ by just one bit and the ASCII character 2 differs from the double quote by just one bit, too. That made it much easier to encode characters mechanically or with a non-microcontroller-based electronic keyboard and that pairing was found on old teletypes.

The ASCII standard was published by the United States of America Standards Institute (USASI) in 1968.

SEE ALSO

charsets(7), *iso_8859-1*(7), *iso_8859-2*(7), *iso_8859-3*(7), *iso_8859-4*(7), *iso_8859-5*(7), *iso_8859-6*(7), *iso_8859-7*(7), *iso_8859-8*(7), *iso_8859-9*(7), *iso_8859-10*(7), *iso_8859-11*(7), *iso_8859-13*(7), *iso_8859-14*(7), *iso_8859-15*(7), *iso_8859-16*(7), *utf-8*(7)

NAME

attributes − POSIX safety concepts

DESCRIPTION

Note: the text of this man page is based on the material taken from the "POSIX Safety Concepts" section of the GNU C Library manual. Further details on the topics described here can be found in that manual.

Various function manual pages include a section ATTRIBUTES that describes the safety of calling the function in various contexts. This section annotates functions with the following safety markings:

MT-Safe
: *MT-Safe* or Thread-Safe functions are safe to call in the presence of other threads. MT, in MT-Safe, stands for Multi Thread.

  Being MT-Safe does not imply a function is atomic, nor that it uses any of the memory synchronization mechanisms POSIX exposes to users. It is even possible that calling MT-Safe functions in sequence does not yield an MT-Safe combination. For example, having a thread call two MT-Safe functions one right after the other does not guarantee behavior equivalent to atomic execution of a combination of both functions, since concurrent calls in other threads may interfere in a destructive way.

  Whole-program optimizations that could inline functions across library interfaces may expose unsafe reordering, and so performing inlining across the GNU C Library interface is not recommended. The documented MT-Safety status is not guaranteed under whole-program optimization. However, functions defined in user-visible headers are designed to be safe for inlining.

MT-Unsafe
: *MT-Unsafe* functions are not safe to call in a multithreaded programs.

Other keywords that appear in safety notes are defined in subsequent sections.

Conditionally safe features

For some features that make functions unsafe to call in certain contexts, there are known ways to avoid the safety problem other than refraining from calling the function altogether. The keywords that follow refer to such features, and each of their definitions indicates how the whole program needs to be constrained in order to remove the safety problem indicated by the keyword. Only when all the reasons that make a function unsafe are observed and addressed, by applying the documented constraints, does the function become safe to call in a context.

init Functions marked with *init* as an MT-Unsafe feature perform MT-Unsafe initialization when they are first called.

Calling such a function at least once in single-threaded mode removes this specific cause for the function to be regarded as MT-Unsafe. If no other cause for that remains, the function can then be safely called after other threads are started.

race Functions annotated with *race* as an MT-Safety issue operate on objects in ways that may cause data races or similar forms of destructive interference out of concurrent execution. In some cases, the objects are passed to the functions by users; in others, they are used by the functions to return values to users; in others, they are not even exposed to users.

const Functions marked with *const* as an MT-Safety issue non-atomically modify internal objects that are better regarded as constant, because a substantial portion of the GNU C Library accesses them without synchronization. Unlike *race*, which causes both readers and writers of internal objects to be regarded as MT-Unsafe, this mark is applied to writers only. Writers remain MT-Unsafe to call, but the then-mandatory constness of objects they modify enables readers to be regarded as MT-Safe (as long as no other reasons for them to be unsafe remain), since the lack of synchronization is not a problem when the objects are effectively constant.

The identifier that follows the *const* mark will appear by itself as a safety note in readers. Programs that wish to work around this safety issue, so as to call writers, may use a non-recursive read-write lock associated with the identifier, and guard *all* calls to functions marked with *const* followed by the identifier with a write lock, and *all* calls to functions marked with the identifier by itself with a read lock.

sig Functions marked with *sig* as a MT-Safety issue may temporarily install a signal handler for internal purposes, which may interfere with other uses of the signal, identified after a colon.

This safety problem can be worked around by ensuring that no other uses of the signal will take place for the duration of the call. Holding a non-recursive mutex while calling all functions that use the same temporary signal; blocking that signal before the call and resetting its handler afterwards is recommended.

term Functions marked with *term* as an MT-Safety issue may change the terminal settings in the recommended way, namely: call *tcgetattr*(3), modify some flags, and then call *tcsetattr*(3), this creates a window in which changes made by other threads are lost. Thus, functions marked with *term* are MT-Unsafe.

It is thus advisable for applications using the terminal to avoid concurrent and reentrant interactions with it, by not using it in signal handlers or blocking signals that might use it, and holding a lock while calling these functions and interacting with the terminal. This lock should also be used for mutual exclusion with functions marked with *race:tcattr(fd)*, where *fd* is a file descriptor for the controlling terminal. The caller may use a single mutex for simplicity, or use one mutex per terminal, even if referenced by different file descriptors.

Other safety remarks

Additional keywords may be attached to functions, indicating features that do not make a function unsafe to call, but that may need to be taken into account in certain classes of programs:

locale Functions annotated with *locale* as an MT-Safety issue read from the locale object without any form of synchronization. Functions annotated with *locale* called concurrently with locale changes may behave in ways that do not correspond to any of the locales active during their execution, but an unpredictable mix thereof.

We do not mark these functions as MT-Unsafe, however, because functions that modify the locale object are marked with *const:locale* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the locale can be considered effectively constant in these contexts, which makes the former safe.

env Functions marked with *env* as an MT-Safety issue access the environment with *getenv*(3) or similar, without any guards to ensure safety in the presence of concurrent modifications.

We do not mark these functions as MT-Unsafe, however, because functions that modify the environment are all marked with *const:env* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the environment can be considered effectively constant in these contexts, which makes the former safe.

hostid The function marked with *hostid* as an MT-Safety issue reads from the system-wide data structures that hold the "host ID" of the machine. These data structures cannot generally be modified atomically. Since it is expected that the "host ID" will not normally change, the function that reads from it (**gethostid**(3)) is regarded as safe, whereas the function that modifies it (**sethostid**(3)) is marked with *const:hostid*, indicating it may require special care if it is to be called. In this specific case, the special care amounts to system-wide (not merely intra-process) coordination.

sigintr Functions marked with *sigintr* as an MT-Safety issue access the GNU C Library *_sigintr* internal data structure without any guards to ensure safety in the presence of concurrent modifications.

We do not mark these functions as MT-Unsafe, however, because functions that modify this data structure are all marked with *const:sigintr* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the data structure can be considered effectively constant in these contexts, which makes the former safe.

cwd Functions marked with *cwd* as an MT-Safety issue may temporarily change the current working directory during their execution, which may cause relative pathnames to be resolved in unexpected ways in other threads or within asynchronous signal or cancelation handlers.

This is not enough of a reason to mark so-marked functions as MT-Unsafe, but when this behavior is optional (e.g., *nftw*(3) with **FTW_CHDIR**), avoiding the option may be a good alternative to using full pathnames or file descriptor-relative (e.g., *openat*(2)) system calls.

:identifier

Annotations may sometimes be followed by identifiers, intended to group several functions that, for example, access the data structures in an unsafe way, as in *race* and *const*, or to provide more specific information, such as naming a signal in a function marked with *sig*. It is envisioned that it may be applied to *lock* and *corrupt* as well in the future.

In most cases, the identifier will name a set of functions, but it may name global objects or function arguments, or identifiable properties or logical components associated with them, with a notation such as, for example, *:buf(arg)* to denote a buffer associated with the argument *arg*, or *:tcattr(fd)* to denote the terminal attributes of a file descriptor *fd*.

The most common use for identifiers is to provide logical groups of functions and arguments that need to be protected by the same synchronization primitive in order to ensure safe operation in a given context.

/condition

Some safety annotations may be conditional, in that they only apply if a boolean expression involving arguments, global variables or even the underlying kernel evaluates to true. For example, */!ps* and */one_per_line* indicate the preceding marker only applies when argument *ps* is NULL, or global variable *one_per_line* is nonzero.

When all marks that render a function unsafe are adorned with such conditions, and none of the named conditions hold, then the function can be regarded as safe.

SEE ALSO

pthreads(7), *signal-safety*(7)

NAME

boot − System bootup process based on UNIX System V Release 4

DESCRIPTION

The **bootup process** (or "**boot sequence**") varies in details among systems, but can be roughly divided into phases controlled by the following components:

(1) hardware

(2) operating system (OS) loader

(3) kernel

(4) root user-space process (*init* and *inittab*)

(5) boot scripts

Each of these is described below in more detail.

Hardware

After power-on or hard reset, control is given to a program stored in read-only memory (normally PROM); for historical reasons involving the personal computer, this program is often called "the **BIOS**".

This program normally performs a basic self-test of the machine and accesses nonvolatile memory to read further parameters. This memory in the PC is battery-backed CMOS memory, so most people refer to it as "the **CMOS**"; outside of the PC world, it is usually called "the **NVRAM**" (nonvolatile RAM).

The parameters stored in the NVRAM vary among systems, but as a minimum, they should specify which device can supply an OS loader, or at least which devices may be probed for one; such a device is known as "the **boot device**". The hardware boot stage loads the OS loader from a fixed position on the boot device, and then transfers control to it.

Note: The device from which the OS loader is read may be attached via a network, in which case the details of booting are further specified by protocols such as DHCP, TFTP, PXE, Etherboot, etc.

OS loader

The main job of the OS loader is to locate the kernel on some device, load it, and run it. Most OS loaders allow interactive use, in order to enable specification of an alternative kernel (maybe a backup in case the one last compiled isn't functioning) and to pass optional parameters to the kernel.

In a traditional PC, the OS loader is located in the initial 512-byte block of the boot device; this block is known as "the **MBR**" (Master Boot Record).

In most systems, the OS loader is very limited due to various constraints. Even on non-PC systems, there are some limitations on the size and complexity of this loader, but the size limitation of the PC MBR (512 bytes, including the partition table) makes it almost impossible to squeeze much functionality into it.

Therefore, most systems split the role of loading the OS between a primary OS loader and a secondary OS loader; this secondary OS loader may be located within a larger portion of persistent storage, such as a disk partition.

In Linux, the OS loader is often *grub*(8) (an alternative is *lilo*(8)).

Kernel

When the kernel is loaded, it initializes various components of the computer and operating system; each portion of software responsible for such a task is usually consider "a **driver**" for the applicable component. The kernel starts the virtual memory swapper (it is a kernel process, called "kswapd" in a modern Linux kernel), and mounts some filesystem at the root path, `/`.

Some of the parameters that may be passed to the kernel relate to these activities (for example, the default root filesystem can be overridden); for further information on Linux kernel parameters, read *boot-param*(7).

Only then does the kernel create the initial userland process, which is given the number 1 as its **PID** (process ID). Traditionally, this process executes the program */sbin/init*, to which are passed the parameters that haven't already been handled by the kernel.

Root user-space process

Note: The following description applies to an OS based on UNIX System V Release 4. However, a number of widely used systems have adopted a related but fundamentally different approach known as *systemd*(1), for which the bootup process is detailed in its associated *bootup*(7)

When */sbin/init* starts, it reads */etc/inittab* for further instructions. This file defines what should be run when the */sbin/init* program is instructed to enter a particular run level, giving the administrator an easy way to establish an environment for some usage; each run level is associated with a set of services (for example, run level **S** is single-user mode, and run level **2** entails running most network services).

The administrator may change the current run level via *init*(1), and query the current run level via *runlevel*(8)

However, since it is not convenient to manage individual services by editing this file, */etc/inittab* only bootstraps a set of scripts that actually start/stop the individual services.

Boot scripts

Note: The following description applies to an OS based on UNIX System V Release 4. However, a number of widely used systems (Slackware Linux, FreeBSD, OpenBSD) have a somewhat different scheme for boot scripts.

For each managed service (mail, nfs server, cron, etc.), there is a single startup script located in a specific directory (*/etc/init.d* in most versions of Linux). Each of these scripts accepts as a single argument the word "start" (causing it to start the service) or the word "stop" (causing it to stop the service). The script may optionally accept other "convenience" parameters (e.g., "restart" to stop and then start, "status" to display the service status, etc.). Running the script without parameters displays the possible arguments.

Sequencing directories

To make specific scripts start/stop at specific run levels and in a specific order, there are *sequencing directories*, normally of the form */etc/rc[0−6S].d*. In each of these directories, there are links (usually symbolic) to the scripts in the */etc/init.d* directory.

A primary script (usually */etc/rc*) is called from *inittab*(5); this primary script calls each service's script via a link in the relevant sequencing directory. Each link whose name begins with 'S' is called with the argument "start" (thereby starting the service). Each link whose name begins with 'K' is called with the argument "stop" (thereby stopping the service).

To define the starting or stopping order within the same run level, the name of a link contains an **ordernumber**. Also, for clarity, the name of a link usually ends with the name of the service to which it refers. For example, the link */etc/rc2.d/S80sendmail* starts the *sendmail*(8) service on run level 2. This happens after */etc/rc2.d/S12syslog* is run but before */etc/rc2.d/S90xfs* is run.

To manage these links is to manage the boot order and run levels; under many systems, there are tools to help with this task (e.g., *chkconfig*(8)).

Boot configuration

A program that provides a service is often called a "**daemon**". Usually, a daemon may receive various command-line options and parameters. To allow a system administrator to change these inputs without editing an entire boot script, some separate configuration file is used, and is located in a specific directory where an associated boot script may find it (*/etc/sysconfig* on older Red Hat systems).

In older UNIX systems, such a file contained the actual command line options for a daemon, but in modern Linux systems (and also in HP-UX), it just contains shell variables. A boot script in */etc/init.d* reads and includes its configuration file (that is, it "**sources**" its configuration file) and then uses the variable values.

FILES

/etc/init.d/ , */etc/rc[S0−6].d/* , */etc/sysconfig/*

SEE ALSO

init(1), *systemd*(1), *inittab*(5), bootparam(7), *bootup*(7), *runlevel*(8), *shutdown*(8)

NAME

bootparam − introduction to boot time parameters of the Linux kernel

DESCRIPTION

The Linux kernel accepts certain 'command-line options' or 'boot time parameters' at the moment it is started. In general, this is used to supply the kernel with information about hardware parameters that the kernel would not be able to determine on its own, or to avoid/override the values that the kernel would otherwise detect.

When the kernel is booted directly by the BIOS, you have no opportunity to specify any parameters. So, in order to take advantage of this possibility you have to use a boot loader that is able to pass parameters, such as GRUB.

The argument list

The kernel command line is parsed into a list of strings (boot arguments) separated by spaces. Most of the boot arguments have the form:

```
name[=value_1][,value_2]...[,value_10]
```

where 'name' is a unique keyword that is used to identify what part of the kernel the associated values (if any) are to be given to. Note the limit of 10 is real, as the present code handles only 10 comma separated parameters per keyword. (However, you can reuse the same keyword with up to an additional 10 parameters in unusually complicated situations, assuming the setup function supports it.)

Most of the sorting is coded in the kernel source file *init/main.c*. First, the kernel checks to see if the argument is any of the special arguments 'root=', 'nfsroot=', 'nfsaddrs=', 'ro', 'rw', 'debug', or 'init'. The meaning of these special arguments is described below.

Then it walks a list of setup functions to see if the specified argument string (such as 'foo') has been associated with a setup function ('foo_setup()') for a particular device or part of the kernel. If you passed the kernel the line foo=3,4,5,6 then the kernel would search the bootsetups array to see if 'foo' was registered. If it was, then it would call the setup function associated with 'foo' (foo_setup()) and hand it the arguments 3, 4, 5, and 6 as given on the kernel command line.

Anything of the form 'foo=bar' that is not accepted as a setup function as described above is then interpreted as an environment variable to be set. A (useless?) example would be to use 'TERM=vt100' as a boot argument.

Any remaining arguments that were not picked up by the kernel and were not interpreted as environment variables are then passed onto PID 1, which is usually the *init*(1) program. The most common argument that is passed to the *init* process is the word 'single' which instructs it to boot the computer in single user mode, and not launch all the usual daemons. Check the manual page for the version of *init*(1) installed on your system to see what arguments it accepts.

General non-device-specific boot arguments

'init=...'

This sets the initial command to be executed by the kernel. If this is not set, or cannot be found, the kernel will try */sbin/init*, then */etc/init*, then */bin/init*, then */bin/sh* and panic if all of this fails.

'nfsaddrs=...'

This sets the NFS boot address to the given string. This boot address is used in case of a net boot.

'nfsroot=...'

This sets the NFS root name to the given string. If this string does not begin with '/' or ',' or a digit, then it is prefixed by '/tftpboot/'. This root name is used in case of a net boot.

'root=...'

This argument tells the kernel what device is to be used as the root filesystem while booting. The default of this setting is determined at compile time, and usually is the value of the root device of the system that the kernel was built on. To override this value, and select the second floppy drive as the root device, one would use 'root=/dev/fd1'.

The root device can be specified symbolically or numerically. A symbolic specification has the form */dev/XXYN*, where XX designates the device type (e.g., 'hd' for ST-506 compatible hard disk, with Y in 'a'–'d'; 'sd' for SCSI compatible disk, with Y in 'a'–'e'), Y the driver

letter or number, and N the number (in decimal) of the partition on this device.

Note that this has nothing to do with the designation of these devices on your filesystem. The '/dev/' part is purely conventional.

The more awkward and less portable numeric specification of the above possible root devices in major/minor format is also accepted. (For example, */dev/sda3* is major 8, minor 3, so you could use 'root=0x803' as an alternative.)

'rootdelay='
This parameter sets the delay (in seconds) to pause before attempting to mount the root filesystem.

'rootflags=...'
This parameter sets the mount option string for the root filesystem (see also *fstab*(5)).

'rootfstype=...'
The 'rootfstype' option tells the kernel to mount the root filesystem as if it where of the type specified. This can be useful (for example) to mount an ext3 filesystem as ext2 and then remove the journal in the root filesystem, in fact reverting its format from ext3 to ext2 without the need to boot the box from alternate media.

'ro' and **'rw'**
The 'ro' option tells the kernel to mount the root filesystem as 'read-only' so that filesystem consistency check programs (fsck) can do their work on a quiescent filesystem. No processes can write to files on the filesystem in question until it is 'remounted' as read/write capable, for example, by 'mount −w −n −o remount /'. (See also *mount*(8)

The 'rw' option tells the kernel to mount the root filesystem read/write. This is the default.

'resume=...'
This tells the kernel the location of the suspend-to-disk data that you want the machine to resume from after hibernation. Usually, it is the same as your swap partition or file. Example:

```
resume=/dev/hda2
```

'reserve=...'
This is used to protect I/O port regions from probes. The form of the command is:

reserve=*iobase,extent[,iobase,extent]...*

In some machines it may be necessary to prevent device drivers from checking for devices (auto-probing) in a specific region. This may be because of hardware that reacts badly to the probing, or hardware that would be mistakenly identified, or merely hardware you don't want the kernel to initialize.

The reserve boot-time argument specifies an I/O port region that shouldn't be probed. A device driver will not probe a reserved region, unless another boot argument explicitly specifies that it do so.

For example, the boot line

```
reserve=0x300,32  blah=0x300
```

keeps all device drivers except the driver for 'blah' from probing 0x300−0x31f.

'panic=N'
By default, the kernel will not reboot after a panic, but this option will cause a kernel reboot after N seconds (if N is greater than zero). This panic timeout can also be set by

```
echo N > /proc/sys/kernel/panic
```

'reboot=[warm|cold][,[bios|hard]]'
Since Linux 2.0.22, a reboot is by default a cold reboot. One asks for the old default with 'reboot=warm'. (A cold reboot may be required to reset certain hardware, but might destroy not yet written data in a disk cache. A warm reboot may be faster.) By default, a reboot is hard, by asking the keyboard controller to pulse the reset line low, but there is at least one type of motherboard where that doesn't work. The option 'reboot=bios' will instead jump through the BIOS.

'nosmp' and **'maxcpus=N'**
(Only when __SMP__ is defined.) A command-line option of 'nosmp' or 'maxcpus=0' will disable SMP activation entirely; an option 'maxcpus=N' limits the maximum number of CPUs activated in SMP mode to N.

Boot arguments for use by kernel developers

'debug'
Kernel messages are handed off to a daemon (e.g., *klogd*(8) or similar) so that they may be logged to disk. Messages with a priority above *console_loglevel* are also printed on the console. (For a discussion of log levels, see *syslog*(2).) By default, *console_loglevel* is set to log messages at levels higher than **KERN_DEBUG**. This boot argument will cause the kernel to also print messages logged at level **KERN_DEBUG**. The console loglevel can also be set on a booted system via the */proc/sys/kernel/printk* file (described in *syslog*(2)), the *syslog*(2) **SYSLOG_ACTION_CONSOLE_LEVEL** operation, or *dmesg*(8)

'profile=N'
It is possible to enable a kernel profiling function, if one wishes to find out where the kernel is spending its CPU cycles. Profiling is enabled by setting the variable *prof_shift* to a nonzero value. This is done either by specifying **CONFIG_PROFILE** at compile time, or by giving the 'profile=' option. Now the value that *prof_shift* gets will be N, when given, or **CONFIG_PROFILE_SHIFT**, when that is given, or 2, the default. The significance of this variable is that it gives the granularity of the profiling: each clock tick, if the system was executing kernel code, a counter is incremented:

```
profile[address >> prof_shift]++;
```

The raw profiling information can be read from */proc/profile*. Probably you'll want to use a tool such as readprofile.c to digest it. Writing to */proc/profile* will clear the counters.

Boot arguments for ramdisk use

(Only if the kernel was compiled with **CONFIG_BLK_DEV_RAM**.) In general it is a bad idea to use a ramdisk under Linux—the system will use available memory more efficiently itself. But while booting, it is often useful to load the floppy contents into a ramdisk. One might also have a system in which first some modules (for filesystem or hardware) must be loaded before the main disk can be accessed.

In Linux 1.3.48, ramdisk handling was changed drastically. Earlier, the memory was allocated statically, and there was a 'ramdisk=N' parameter to tell its size. (This could also be set in the kernel image at compile time.) These days ram disks use the buffer cache, and grow dynamically. For a lot of information on the current ramdisk setup, see the kernel source file *Documentation/blockdev/ramdisk.txt* (*Documentation/ramdisk.txt* in older kernels).

There are four parameters, two boolean and two integral.

'load_ramdisk=N'
If N=1, do load a ramdisk. If N=0, do not load a ramdisk. (This is the default.)

'prompt_ramdisk=N'
If N=1, do prompt for insertion of the floppy. (This is the default.) If N=0, do not prompt. (Thus, this parameter is never needed.)

'ramdisk_size=N' or (obsolete) **'ramdisk=N'**
Set the maximal size of the ramdisk(s) to N kB. The default is 4096 (4 MB).

'ramdisk_start=N'
Sets the starting block number (the offset on the floppy where the ramdisk starts) to N. This is needed in case the ramdisk follows a kernel image.

'noinitrd'
(Only if the kernel was compiled with **CONFIG_BLK_DEV_RAM** and **CONFIG_BLK_DEV_INITRD**.) These days it is possible to compile the kernel to use initrd. When this feature is enabled, the boot process will load the kernel and an initial ramdisk; then the kernel converts initrd into a "normal" ramdisk, which is mounted read-write as root device; then */linuxrc* is executed; afterward the "real" root filesystem is mounted, and the initrd filesystem is moved over to */initrd*; finally the usual boot sequence (e.g., invocation of */sbin/init*) is performed.

For a detailed description of the initrd feature, see the kernel source file *Documentation/admin−guide/initrd.rst* (or *Documentation/initrd.txt* before Linux 4.10).

The 'noinitrd' option tells the kernel that although it was compiled for operation with initrd, it should not go through the above steps, but leave the initrd data under */dev/initrd*. (This device can be used only once: the data is freed as soon as the last process that used it has closed */dev/initrd*.)

Boot arguments for SCSI devices

General notation for this section:

iobase -- the first I/O port that the SCSI host occupies. These are specified in hexadecimal notation, and usually lie in the range from 0x200 to 0x3ff.

irq -- the hardware interrupt that the card is configured to use. Valid values will be dependent on the card in question, but will usually be 5, 7, 9, 10, 11, 12, and 15. The other values are usually used for common peripherals like IDE hard disks, floppies, serial ports, and so on.

scsi−id -- the ID that the host adapter uses to identify itself on the SCSI bus. Only some host adapters allow you to change this value, as most have it permanently specified internally. The usual default value is 7, but the Seagate and Future Domain TMC-950 boards use 6.

parity -- whether the SCSI host adapter expects the attached devices to supply a parity value with all information exchanges. Specifying a one indicates parity checking is enabled, and a zero disables parity checking. Again, not all adapters will support selection of parity behavior as a boot argument.

'max_scsi_luns=...'

A SCSI device can have a number of 'subdevices' contained within itself. The most common example is one of the new SCSI CD-ROMs that handle more than one disk at a time. Each CD is addressed as a 'Logical Unit Number' (LUN) of that particular device. But most devices, such as hard disks, tape drives, and such are only one device, and will be assigned to LUN zero.

Some poorly designed SCSI devices cannot handle being probed for LUNs not equal to zero. Therefore, if the compile-time flag **CONFIG_SCSI_MULTI_LUN** is not set, newer kernels will by default probe only LUN zero.

To specify the number of probed LUNs at boot, one enters 'max_scsi_luns=n' as a boot arg, where n is a number between one and eight. To avoid problems as described above, one would use n=1 to avoid upsetting such broken devices.

SCSI tape configuration

Some boot time configuration of the SCSI tape driver can be achieved by using the following:

```
st=buf_size[,write_threshold[,max_bufs]]
```

The first two numbers are specified in units of kB. The default *buf_size* is 32k B, and the maximum size that can be specified is a ridiculous 16384 kB. The *write_threshold* is the value at which the buffer is committed to tape, with a default value of 30 kB. The maximum number of buffers varies with the number of drives detected, and has a default of two. An example usage would be:

```
st=32,30,2
```

Full details can be found in the file *Documentation/scsi/st.txt* (or *drivers/scsi/README.st* for older kernels) in the Linux kernel source.

Hard disks

IDE Disk/CD-ROM Driver Parameters

The IDE driver accepts a number of parameters, which range from disk geometry specifications, to support for broken controller chips. Drive-specific options are specified by using 'hdX=' with X in 'a'–'h'.

Non-drive-specific options are specified with the prefix 'hd='. Note that using a drive-specific prefix for a non-drive-specific option will still work, and the option will just be applied as expected.

Also note that 'hd=' can be used to refer to the next unspecified drive in the (a, ..., h) sequence. For the following discussions, the 'hd=' option will be cited for brevity. See the file

Documentation/ide/ide.txt (or *Documentation/ide.txt* in older kernels, or *drivers/block/README.ide* in ancient kernels) in the Linux kernel source for more details.

The 'hd=cyls,heads,sects[,wpcom[,irq]]' options
These options are used to specify the physical geometry of the disk. Only the first three values are required. The cylinder/head/sectors values will be those used by fdisk. The write precompensation value is ignored for IDE disks. The IRQ value specified will be the IRQ used for the interface that the drive resides on, and is not really a drive-specific parameter.

The 'hd=serialize' option
The dual IDE interface CMD-640 chip is broken as designed such that when drives on the secondary interface are used at the same time as drives on the primary interface, it will corrupt your data. Using this option tells the driver to make sure that both interfaces are never used at the same time.

The 'hd=noprobe' option
Do not probe for this drive. For example,

```
hdb=noprobe hdb=1166,7,17
```

would disable the probe, but still specify the drive geometry so that it would be registered as a valid block device, and hence usable.

The 'hd=nowerr' option
Some drives apparently have the **WRERR_STAT** bit stuck on permanently. This enables a work-around for these broken devices.

The 'hd=cdrom' option
This tells the IDE driver that there is an ATAPI compatible CD-ROM attached in place of a normal IDE hard disk. In most cases the CD-ROM is identified automatically, but if it isn't then this may help.

Standard ST-506 Disk Driver Options ('hd=')
The standard disk driver can accept geometry arguments for the disks similar to the IDE driver. Note however that it expects only three values (C/H/S); any more or any less and it will silently ignore you. Also, it accepts only 'hd=' as an argument, that is, 'hda=' and so on are not valid here. The format is as follows:

```
hd=cyls,heads,sects
```

If there are two disks installed, the above is repeated with the geometry parameters of the second disk.

Ethernet devices
Different drivers make use of different parameters, but they all at least share having an IRQ, an I/O port base value, and a name. In its most generic form, it looks something like this:

```
ether=irq,iobase[,param_1[,...param_8]],name
```

The first nonnumeric argument is taken as the name. The param_n values (if applicable) usually have different meanings for each different card/driver. Typical param_n values are used to specify things like shared memory address, interface selection, DMA channel and the like.

The most common use of this parameter is to force probing for a second ethercard, as the default is to probe only for one. This can be accomplished with a simple:

```
ether=0,0,eth1
```

Note that the values of zero for the IRQ and I/O base in the above example tell the driver(s) to autoprobe.

The Ethernet-HowTo has extensive documentation on using multiple cards and on the card/driver-specific implementation of the param_n values where used. Interested readers should refer to the section in that document on their particular card.

The floppy disk driver
There are many floppy driver options, and they are all listed in *Documentation/blockdev/floppy.txt* (or *Documentation/floppy.txt* in older kernels, or *drivers/block/README.fd* for ancient kernels) in the Linux kernel source. See that file for the details.

The sound driver

The sound driver can also accept boot arguments to override the compiled-in values. This is not recommended, as it is rather complex. It is described in the Linux kernel source file *Documentation/sound/oss/README.OSS* (*drivers/sound/Readme.linux* in older kernel versions). It accepts a boot argument of the form:

```
sound=device1[,device2[,device3...[,device10]]]
```

where each deviceN value is of the following format 0xTaaaId and the bytes are used as follows:

T − device type: 1=FM, 2=SB, 3=PAS, 4=GUS, 5=MPU401, 6=SB16, 7=SB16-MPU401

aaa − I/O address in hex.

I − interrupt line in hex (i.e., 10=a, 11=b, ...)

d − DMA channel.

As you can see, it gets pretty messy, and you are better off to compile in your own personal values as recommended. Using a boot argument of 'sound=0' will disable the sound driver entirely.

The line printer driver

'lp='

Syntax:

```
lp=0
lp=auto
lp=reset
lp=port[,port...]
```

You can tell the printer driver what ports to use and what ports not to use. The latter comes in handy if you don't want the printer driver to claim all available parallel ports, so that other drivers (e.g., PLIP, PPA) can use them instead.

The format of the argument is multiple port names. For example, lp=none,parport0 would use the first parallel port for lp1, and disable lp0. To disable the printer driver entirely, one can use lp=0.

SEE ALSO

klogd(8), *mount*(8)

For up-to-date information, see the kernel source file *Documentation/admin−guide/kernel−parameters.txt*.

NAME

BPF-HELPERS − list of eBPF helper functions

DESCRIPTION

The extended Berkeley Packet Filter (eBPF) subsystem consists in programs written in a pseudo−assembly language, then attached to one of the several kernel hooks and run in reaction of specific events. This framework differs from the older, "classic" BPF (or "cBPF") in several aspects, one of them being the ability to call special functions (or "helpers") from within a program. These functions are restricted to a white−list of helpers defined in the kernel.

These helpers are used by eBPF programs to interact with the system, or with the context in which they work. For instance, they can be used to print debugging messages, to get the time since the system was booted, to interact with eBPF maps, or to manipulate network packets. Since there are several eBPF program types, and that they do not run in the same context, each program type can only call a subset of those helpers.

Due to eBPF conventions, a helper can not have more than five arguments.

Internally, eBPF programs call directly into the compiled helper functions without requiring any foreign−function interface. As a result, calling helpers introduces no overhead, thus offering excellent performance.

This document is an attempt to list and document the helpers available to eBPF developers. They are sorted by chronological order (the oldest helpers in the kernel at the top).

HELPERS

void *bpf_map_lookup_elem(struct bpf_map ***map*, const void ***key***)**

Description

Perform a lookup in *map* for an entry associated to *key*.

Return Map value associated to *key*, or **NULL** if no entry was found.

long bpf_map_update_elem(struct bpf_map ***map*, const void ***key***, const void ***value***, u64** *flags***)**

Description

Add or update the value of the entry associated to *key* in *map* with *value*. *flags* is one of:

BPF_NOEXIST

The entry for *key* must not exist in the map.

BPF_EXIST

The entry for *key* must already exist in the map.

BPF_ANY

No condition on the existence of the entry for *key*.

Flag value **BPF_NOEXIST** cannot be used for maps of types **BPF_MAP_TYPE_ARRAY** or **BPF_MAP_TYPE_PERCPU_ARRAY** (all elements always exist), the helper would return an error.

Return 0 on success, or a negative error in case of failure.

long bpf_map_delete_elem(struct bpf_map ***map*, const void ***key***)**

Description

Delete entry with *key* from *map*.

Return 0 on success, or a negative error in case of failure.

long bpf_probe_read(void ***dst*, u32** *size***, const void ***unsafe_ptr***)**

Description

For tracing programs, safely attempt to read *size* bytes from kernel space address *unsafe_ptr* and store the data in *dst*.

Generally, use **bpf_probe_read_user**() or **bpf_probe_read_kernel**() instead.

Return 0 on success, or a negative error in case of failure.

u64 bpf_ktime_get_ns(void)

Description

Return the time elapsed since system boot, in nanoseconds. Does not include time the system was suspended. See: **clock_gettime**(**CLOCK_MONOTONIC**)

Return Current *ktime*.

long bpf_trace_printk(const char ****fmt*, u32** *fmt_size***, ...)**

Description

This helper is a "printk()−like" facility for debugging. It prints a message defined by format *fmt* (of size *fmt_size*) to file */sys/kernel/tracing/trace* from TraceFS, if available. It can take up to three additional **u64** arguments (as an eBPF helpers, the total number of arguments is limited to five).

Each time the helper is called, it appends a line to the trace. Lines are discarded while */sys/kernel/tracing/trace* is open, use */sys/kernel/tracing/trace_pipe* to avoid this. The format of the trace is customizable, and the exact output one will get depends on the options set in */sys/kernel/tracing/trace_options* (see also the *README* file under the same directory). However, it usually defaults to something like:

```
telnet-470   [001] .N.. 419421.045894: 0x00000001: <formatted msg>
```

In the above:

- **telnet** is the name of the current task.
- **470** is the PID of the current task.
- **001** is the CPU number on which the task is running.
- In **.N..**, each character refers to a set of options (whether irqs are enabled, scheduling options, whether hard/softirqs are running, level of preempt_disabled respectively). **N** means that **TIF_NEED_RESCHED** and **PREEMPT_NEED_RESCHED** are set.
- **419421.045894** is a timestamp.
- **0x00000001** is a fake value used by BPF for the instruction pointer register.
- **<formatted msg>** is the message formatted with *fmt*.

The conversion specifiers supported by *fmt* are similar, but more limited than for printk(). They are **%d**, **%i**, **%u**, **%x**, **%ld**, **%li**, **%lu**, **%lx**, **%lld**, **%lli**, **%llu**, **%llx**, **%p**, **%s**. No modifier (size of field, padding with zeroes, etc.) is available, and the helper will return **−EINVAL** (but print nothing) if it encounters an unknown specifier.

Also, note that **bpf_trace_printk**() is slow, and should only be used for debugging purposes. For this reason, a notice block (spanning several lines) is printed to kernel logs and states that the helper should not be used "for production use" the first time this helper is used (or more precisely, when **trace_printk**() buffers are allocated). For passing values to user space, perf events should be preferred.

Return The number of bytes written to the buffer, or a negative error in case of failure.

u32 bpf_get_prandom_u32(void)

Description

Get a pseudo−random number.

From a security point of view, this helper uses its own pseudo−random internal state, and cannot be used to infer the seed of other random functions in the kernel. However, it is essential to note that the generator used by the helper is not cryptographically secure.

Return A random 32−bit unsigned value.

u32 bpf_get_smp_processor_id(void)

Description

Get the SMP (symmetric multiprocessing) processor id. Note that all programs run with migration disabled, which means that the SMP processor id is stable during all the execution of the program.

Return The SMP id of the processor running the program.

long bpf_skb_store_bytes(struct sk_buff *_skb_, u32 _offset_, const void *_from_, u32 _len_, u64 _flags_)

Description

Store *len* bytes from address *from* into the packet associated to *skb*, at *offset*. *flags* are a combination of **BPF_F_RECOMPUTE_CSUM** (automatically recompute the checksum for the packet after storing the bytes) and **BPF_F_INVALIDATE_HASH** (set *skb*−>**hash**, *skb*−>**swhash** and *skb*−>**l4hash** to 0).

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_l3_csum_replace(struct sk_buff *_skb_, u32 _offset_, u64 _from_, u64 _to_, u64 _size_)

Description

Recompute the layer 3 (e.g. IP) checksum for the packet associated to *skb*. Computation is incremental, so the helper must know the former value of the header field that was modified (*from*), the new value of this field (*to*), and the number of bytes (2 or 4) for this field, stored in *size*. Alternatively, it is possible to store the difference between the previous and the new values of the header field in *to*, by setting *from* and *size* to 0. For both methods, *offset* indicates the location of the IP checksum within the packet.

This helper works in combination with **bpf_csum_diff**(), which does not update the checksum in−place, but offers more flexibility and can handle sizes larger than 2 or 4 for the checksum to update.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_l4_csum_replace(struct sk_buff *_skb_, u32 _offset_, u64 _from_, u64 _to_, u64 _flags_)

Description

Recompute the layer 4 (e.g. TCP, UDP or ICMP) checksum for the packet associated to *skb*. Computation is incremental, so the helper must know the former value of the header field that was modified (*from*), the new value of this field (*to*), and the number of bytes (2 or 4) for this field, stored on the lowest four bits of *flags*. Alternatively, it is possible to store the difference between the previous and the new values of the header field in *to*, by setting *from* and the four lowest bits of *flags* to 0. For both methods, *offset* indicates the location of the IP checksum within the packet. In addition to the size of the field, *flags* can be added (bitwise OR) actual flags. With **BPF_F_MARK_MANGLED_0**, a null checksum is left untouched (unless **BPF_F_MARK_ENFORCE** is added as well), and for updates resulting in a null checksum the value is set to **CSUM_MANGLED_0** instead. Flag **BPF_F_PSEUDO_HDR** indicates the checksum is to be computed against a pseudo−header.

This helper works in combination with **bpf_csum_diff**(), which does not update the checksum in−place, but offers more flexibility and can handle sizes larger than 2 or 4 for the checksum to update.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_tail_call(void *_ctx_, struct bpf_map *_prog_array_map_, u32 _index_)

Description

This special helper is used to trigger a "tail call", or in other words, to jump into another eBPF program. The same stack frame is used (but values on stack and in registers for the caller are not accessible to the callee). This mechanism allows for program chaining, either for raising the maximum number of available eBPF instructions, or to execute given programs in conditional blocks. For security reasons, there is an upper limit to the number of successive tail calls that can be performed.

Upon call of this helper, the program attempts to jump into a program referenced at index *index* in *prog_array_map*, a special map of type **BPF_MAP_TYPE_PROG_ARRAY**, and passes *ctx*, a pointer to the context.

If the call succeeds, the kernel immediately runs the first instruction of the new program. This is not a function call, and it never returns to the previous program. If the call fails, then the helper has no effect, and the caller continues to run its subsequent instructions. A call can fail if the destination program for the jump does not exist (i.e. *index* is superior to the number of entries in *prog_array_map*), or if the maximum number of tail calls has been reached for this chain of programs. This limit is defined in the kernel by the macro **MAX_TAIL_CALL_CNT** (not accessible to user space), which is currently set to 33.

Return 0 on success, or a negative error in case of failure.

long bpf_clone_redirect(struct sk_buff *_skb_, u32 _ifindex_, u64 _flags_)

Description

Clone and redirect the packet associated to *skb* to another net device of index *ifindex*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

In comparison with **bpf_redirect**() helper, **bpf_clone_redirect**() has the associated cost of duplicating the packet buffer, but this can be executed out of the eBPF program. Conversely, **bpf_redirect**() is more efficient, but it is handled through an action code where the redirection happens only after the eBPF program has returned.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure. Positive error indicates a potential drop or congestion in the target device. The particular positive error codes are not defined.

u64 bpf_get_current_pid_tgid(void)

Description

Get the current pid and tgid.

Return A 64−bit integer containing the current tgid and pid, and created as such: *current_task*−>**tgid << 32 |** *current_task*−>**pid**.

u64 bpf_get_current_uid_gid(void)

Description

Get the current uid and gid.

Return A 64−bit integer containing the current GID and UID, and created as such: *current_gid* **<< 32 |** *current_uid*.

long bpf_get_current_comm(void *****buf***, u32** *size_of_buf***)**

Description

Copy the **comm** attribute of the current task into *buf* of *size_of_buf*. The **comm** attribute contains the name of the executable (excluding the path) for the current task. The *size_of_buf* must be strictly positive. On success, the helper makes sure that the *buf* is NUL−terminated. On failure, it is filled with zeroes.

Return 0 on success, or a negative error in case of failure.

u32 bpf_get_cgroup_classid(struct sk_buff *****skb***)**

Description

Retrieve the classid for the current task, i.e. for the net_cls cgroup to which *skb* belongs.

This helper can be used on TC egress path, but not on ingress.

The net_cls cgroup provides an interface to tag network packets based on a user−provided identifier for all traffic coming from the tasks belonging to the related cgroup. See also the related kernel documentation, available from the Linux sources in file *Documentation/admin−guide/cgroup−v1/net_cls.rst*.

The Linux kernel has two versions for cgroups: there are cgroups v1 and cgroups v2. Both are available to users, who can use a mixture of them, but note that the net_cls cgroup is for cgroup v1 only. This makes it incompatible with BPF programs run on cgroups, which is a cgroup−v2−only feature (a socket can only hold data for one version of cgroups at a time).

This helper is only available is the kernel was compiled with the **CONFIG_CGROUP_NET_CLASSID** configuration option set to "**y**" or to "**m**".

Return The classid, or 0 for the default unconfigured classid.

long bpf_skb_vlan_push(struct sk_buff *****skb***, __be16** *vlan_proto***, u16** *vlan_tci***)**

Description

Push a *vlan_tci* (VLAN tag control information) of protocol *vlan_proto* to the packet associated to *skb*, then update the checksum. Note that if *vlan_proto* is different from **ETH_P_8021Q** and **ETH_P_8021AD**, it is considered to be **ETH_P_8021Q**.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_vlan_pop(struct sk_buff *****skb***)**

Description

Pop a VLAN header from the packet associated to *skb*.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet

access.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_get_tunnel_key(struct sk_buff ****skb*, struct bpf_tunnel_key ****key***, u32** *size***, u64** *flags***)**

Description

Get tunnel metadata. This helper takes a pointer *key* to an empty **struct bpf_tunnel_key** of **size**, that will be filled with tunnel metadata for the packet associated to *skb*. The *flags* can be set to **BPF_F_TUNINFO_IPV6**, which indicates that the tunnel is based on IPv6 protocol instead of IPv4.

The **struct bpf_tunnel_key** is an object that generalizes the principal parameters used by various tunneling protocols into a single struct. This way, it can be used to easily make a decision based on the contents of the encapsulation header, "summarized" in this struct. In particular, it holds the IP address of the remote end (IPv4 or IPv6, depending on the case) in *key*->**remote_ipv4** or *key*->**remote_ipv6**. Also, this struct exposes the *key*->**tunnel_id**, which is generally mapped to a VNI (Virtual Network Identifier), making it programmable together with the **bpf_skb_set_tunnel_key**() helper.

Let's imagine that the following code is part of a program attached to the TC ingress interface, on one end of a GRE tunnel, and is supposed to filter out all messages coming from remote ends with IPv4 address other than 10.0.0.1:

```
int ret;
struct bpf_tunnel_key key = {};
ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key), 0);
if (ret < 0)
        return TC_ACT_SHOT;     // drop packet
if (key.remote_ipv4 != 0x0a000001)
        return TC_ACT_SHOT;     // drop packet
return TC_ACT_OK;               // accept packet
```

This interface can also be used with all encapsulation devices that can operate in "collect metadata" mode: instead of having one network device per specific configuration, the "collect metadata" mode only requires a single device where the configuration can be extracted from this helper.

This can be used together with various tunnels such as VXLan, Geneve, GRE or IP in IP (IPIP).

Return 0 on success, or a negative error in case of failure.

long bpf_skb_set_tunnel_key(struct sk_buff ****skb*, struct bpf_tunnel_key ****key***, u32** *size***, u64** *flags***)**

Description

Populate tunnel metadata for packet associated to *skb.* The tunnel metadata is set to the contents of *key*, of *size*. The *flags* can be set to a combination of the following values:

BPF_F_TUNINFO_IPV6
: Indicate that the tunnel is based on IPv6 protocol instead of IPv4.

BPF_F_ZERO_CSUM_TX
: For IPv4 packets, add a flag to tunnel metadata indicating that checksum computation should be skipped and checksum set to zeroes.

BPF_F_DONT_FRAGMENT
: Add a flag to tunnel metadata indicating that the packet should not be fragmented.

BPF_F_SEQ_NUMBER
: Add a flag to tunnel metadata indicating that a sequence number should be added to tunnel header before sending the packet. This flag was added for

GRE encapsulation, but might be used with other protocols as well in the future.

BPF_F_NO_TUNNEL_KEY
Add a flag to tunnel metadata indicating that no tunnel key should be set in the resulting tunnel header.

Here is a typical usage on the transmit path:

```
struct bpf_tunnel_key key;
     populate key ...
bpf_skb_set_tunnel_key(skb, &key, sizeof(key), 0);
bpf_clone_redirect(skb, vxlan_dev_ifindex, 0);
```

See also the description of the **bpf_skb_get_tunnel_key**() helper for additional information.

Return 0 on success, or a negative error in case of failure.

u64 bpf_perf_event_read(struct bpf_map *_map_, u64 _flags_)

Description
Read the value of a perf event counter. This helper relies on a *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. The nature of the perf event counter is selected when *map* is updated with perf event file descriptors. The *map* is an array whose size is the number of available CPUs, and each cell contains a value relative to one CPU. The value to retrieve is indicated by *flags*, that contains the index of the CPU to look up, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the value for the current CPU should be retrieved.

Note that before Linux 4.13, only hardware perf event can be retrieved.

Also, be aware that the newer helper **bpf_perf_event_read_value**() is recommended over **bpf_perf_event_read**() in general. The latter has some ABI quirks where error and counter value are used as a return code (which is wrong to do since ranges may overlap). This issue is fixed with **bpf_perf_event_read_value**(), which at the same time provides more features over the **bpf_perf_event_read**() interface. Please refer to the description of **bpf_perf_event_read_value**() for details.

Return The value of the perf event counter read from the map, or a negative error code in case of failure.

long bpf_redirect(u32 _ifindex_, u64 _flags_)

Description
Redirect the packet to another net device of index *ifindex*. This helper is somewhat similar to **bpf_clone_redirect**(), except that the packet is not cloned, which provides increased performance.

Except for XDP, both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). Currently, XDP only supports redirection to the egress interface, and accepts no flag at all.

The same effect can also be attained with the more generic **bpf_redirect_map**(), which uses a BPF map to store the redirect target instead of providing it directly to the helper.

Return For XDP, the helper returns **XDP_REDIRECT** on success or **XDP_ABORTED** on error. For other program types, the values are **TC_ACT_REDIRECT** on success or **TC_ACT_SHOT** on error.

u32 bpf_get_route_realm(struct sk_buff *_skb_)

Description

Retrieve the realm or the route, that is to say the **tclassid** field of the destination for the *skb*. The identifier retrieved is a user−provided tag, similar to the one used with the net_cls cgroup (see description for **bpf_get_cgroup_classid**() helper), but here this tag is held by a route (a destination entry), not by a task.

Retrieving this identifier works with the clsact TC egress hook (see also **tc−bpf(8)**), or alternatively on conventional classful egress qdiscs, but not on TC ingress path. In case of clsact TC egress hook, this has the advantage that, internally, the destination entry has not been dropped yet in the transmit path. Therefore, the destination entry does not need to be artificially held via **netif_keep_dst**() for a classful qdisc until the *skb* is freed.

This helper is available only if the kernel was compiled with **CONFIG_IP_ROUTE_CLASSID** configuration option.

Return The realm of the route for the packet associated to *skb*, or 0 if none was found.

long bpf_perf_event_output(void *_ctx_, struct bpf_map *_map_, u64 _flags_, void *_data_, u64 _size_)

Description

Write raw *data* blob into a special BPF perf event held by *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. This perf event must have the following attributes: **PERF_SAMPLE_RAW** as **sample_type**, **PERF_TYPE_SOFTWARE** as **type**, and **PERF_COUNT_SW_BPF_OUTPUT** as **config**.

The *flags* are used to indicate the index in *map* for which the value must be put, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the index of the current CPU core should be used.

The value to write, of *size*, is passed through eBPF stack and pointed by *data*.

The context of the program *ctx* needs also be passed to the helper.

On user space, a program willing to read the values needs to call **perf_event_open**() on the perf event (either for one or for all CPUs) and to store the file descriptor into the *map*. This must be done before the eBPF program can send data into it. An example is available in file *samples/bpf/trace_output_user.c* in the Linux kernel source tree (the eBPF program counterpart is in *samples/bpf/trace_output_kern.c*).

bpf_perf_event_output() achieves better performance than **bpf_trace_printk**() for sharing data with user space, and is much better suitable for streaming data from eBPF programs.

Note that this helper is not restricted to tracing use cases and can be used with programs attached to TC or XDP as well, where it allows for passing data to user space listeners. Data can be:

- Only custom structs,
- Only the packet payload, or
- A combination of both.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_load_bytes(const void *_skb_, u32 _offset_, void *_to_, u32 _len_)

Description

This helper was provided as an easy way to load data from a packet. It can be used to load *len* bytes from *offset* from the packet associated to *skb*, into the buffer pointed by *to*.

Since Linux 4.7, usage of this helper has mostly been replaced by "direct packet access", enabling packet data to be manipulated with *skb*−>**data** and *skb*−>**data_end** pointing respectively to the first byte of packet data and to the byte after the last byte of packet data. However, it remains useful if one wishes to read large quantities of data at once from a packet into the eBPF stack.

Return 0 on success, or a negative error in case of failure.

long bpf_get_stackid(void *_ctx_, struct bpf_map *_map_, u64 _flags_)

Description

Walk a user or a kernel stack and return its id. To achieve this, the helper needs *ctx*, which is a pointer to the context on which the tracing program is executed, and a pointer to a *map* of type **BPF_MAP_TYPE_STACK_TRACE**.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set a combination of the following flags:

BPF_F_USER_STACK
Collect a user space stack instead of a kernel stack.

BPF_F_FAST_STACK_CMP
Compare stacks by hash only.

BPF_F_REUSE_STACKID
If two different stacks hash into the same *stackid*, discard the old one.

The stack id retrieved is a 32 bit long integer handle which can be further combined with other data (including other stack ids) and used as a key into maps. This can be useful for generating a variety of graphs (such as flame graphs or off−cpu graphs).

For walking a stack, this helper is an improvement over **bpf_probe_read**(), which can be used with unrolled loops but is not efficient and consumes a lot of eBPF instructions. Instead, **bpf_get_stackid**() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

```
# sysctl kernel.perf_event_max_stack=<new value>
```

Return The positive or null stack id on success, or a negative error in case of failure.

s64 bpf_csum_diff(__be32 *_from_, u32 _from_size_, __be32 *_to_, u32 _to_size_, __wsum _seed_)

Description

Compute a checksum difference, from the raw buffer pointed by *from*, of length *from_size* (that must be a multiple of 4), towards the raw buffer pointed by *to*, of size *to_size* (same remark). An optional *seed* can be added to the value (this can be cascaded, the seed may come from a previous call to the helper).

This is flexible enough to be used in several ways:

- With *from_size* == 0, *to_size* > 0 and *seed* set to checksum, it can be used when pushing new data.
- With *from_size* > 0, *to_size* == 0 and *seed* set to checksum, it can be used when removing data from a packet.
- With *from_size* > 0, *to_size* > 0 and *seed* set to 0, it can be used to compute a diff. Note that *from_size* and *to_size* do not need to be equal.

This helper can be used in combination with **bpf_l3_csum_replace**() and **bpf_l4_csum_replace**(), to which one can feed in the difference computed with **bpf_csum_diff**().

Return The checksum result, or a negative error code in case of failure.

long bpf_skb_get_tunnel_opt(struct sk_buff *_skb_, void *_opt_, u32 _size_)

Description

Retrieve tunnel options metadata for the packet associated to *skb*, and store the raw tunnel option data to the buffer *opt* of *size*.

This helper can be used with encapsulation devices that can operate in "collect metadata" mode (please refer to the related note in the description of **bpf_skb_get_tunnel_key**() for more details). A particular example where this can be used is in combination with the Geneve encapsulation protocol, where it allows for pushing (with **bpf_skb_get_tunnel_opt**() helper) and retrieving arbitrary TLVs (Type−Length−Value headers) from the eBPF program. This allows for full customization of these headers.

Return The size of the option data retrieved.

long bpf_skb_set_tunnel_opt(struct sk_buff *_skb_, void *_opt_, u32 _size_)

Description

Set tunnel options metadata for the packet associated to *skb* to the option data contained in the raw buffer *opt* of *size*.

See also the description of the **bpf_skb_get_tunnel_opt**() helper for additional information.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_change_proto(struct sk_buff *_skb_, __be16 _proto_, u64 _flags_)

Description

Change the protocol of the *skb* to *proto*. Currently supported are transition from IPv4 to IPv6, and from IPv6 to IPv4. The helper takes care of the groundwork for the transition, including resizing the socket buffer. The eBPF program is expected to fill the new headers, if any, via **skb_store_bytes**() and to recompute the checksums with **bpf_l3_csum_replace**() and **bpf_l4_csum_replace**(). The main case for this helper is to perform NAT64 operations out of an eBPF program.

Internally, the GSO type is marked as dodgy so that headers are checked and segments are recalculated by the GSO/GRO engine. The size for GSO target is adapted as well.

All values for *flags* are reserved for future usage, and must be left at zero.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_change_type(struct sk_buff *_skb_, u32 _type_)

Description

Change the packet type for the packet associated to *skb*. This comes down to setting *skb*−>**pkt_type** to *type*, except the eBPF program does not have a write access to *skb*−>**pkt_type** beside this helper. Using a helper here allows for graceful handling of errors.

The major use case is to change incoming *skb*s to **PACKET_HOST*** in a programmatic way instead of having to recirculate via **redirect**(..., **BPF_F_INGRESS**), for example.

Note that *type* only allows certain values. At this time, they are:

PACKET_HOST
Packet is for us.

PACKET_BROADCAST
Send packet to all.

PACKET_MULTICAST
Send packet to group.

PACKET_OTHERHOST
Send packet to someone else.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_under_cgroup(struct sk_buff *_skb_, struct bpf_map *_map_, u32 _index_)

Description
Check whether *skb* is a descendant of the cgroup2 held by *map* of type **BPF_MAP_TYPE_CGROUP_ARRAY**, at *index*.

Return The return value depends on the result of the test, and can be:

- 0, if the *skb* failed the cgroup2 descendant test.
- 1, if the *skb* succeeded the cgroup2 descendant test.
- A negative error code, if an error occurred.

u32 bpf_get_hash_recalc(struct sk_buff *_skb_)

Description
Retrieve the hash of the packet, *skb*−>**hash**. If it is not set, in particular if the hash was cleared due to mangling, recompute this hash. Later accesses to the hash can be done directly with *skb*−>**hash**.

Calling **bpf_set_hash_invalid**(), changing a packet prototype with **bpf_skb_change_proto**(), or calling **bpf_skb_store_bytes**() with the **BPF_F_INVALIDATE_HASH** are actions susceptible to clear the hash and to trigger a new computation for the next call to **bpf_get_hash_recalc**().

Return The 32−bit hash.

u64 bpf_get_current_task(void)

Description
Get the current task.

Return A pointer to the current task struct.

long bpf_probe_write_user(void *_dst_, const void *_src_, u32 _len_)

Description
Attempt in a safe way to write *len* bytes from the buffer *src* to *dst* in memory. It only works for threads that are in user context, and *dst* must be a valid user space address.

This helper should not be used to implement any kind of security mechanism because of TOC−TOU attacks, but rather to debug, divert, and manipulate execution of semi−cooperative processes.

Keep in mind that this feature is meant for experiments, and it has a risk of crashing the system and running programs. Therefore, when an eBPF program using this helper is attached, a warning including PID and process name is printed to kernel logs.

Return 0 on success, or a negative error in case of failure.

long bpf_current_task_under_cgroup(struct bpf_map *_map_, u32 _index_)

Description
Check whether the probe is being run is the context of a given subset of the cgroup2 hierarchy. The cgroup2 to test is held by *map* of type

BPF_MAP_TYPE_CGROUP_ARRAY, at *index*.

Return The return value depends on the result of the test, and can be:

- 1, if current task belongs to the cgroup2.
- 0, if current task does not belong to the cgroup2.
- A negative error code, if an error occurred.

long bpf_skb_change_tail(struct sk_buff ***skb*, u32** *len***, u64** *flags***)**

Description

Resize (trim or grow) the packet associated to *skb* to the new *len*. The *flags* are reserved for future usage, and must be left at zero.

The basic idea is that the helper performs the needed work to change the size of the packet, then the eBPF program rewrites the rest via helpers like **bpf_skb_store_bytes**(), **bpf_l3_csum_replace**(), **bpf_l3_csum_replace**() and others. This helper is a slow path utility intended for replies with control messages. And because it is targeted for slow path, the helper itself can afford to be slow: it implicitly linearizes, unclones and drops offloads from the *skb*.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_pull_data(struct sk_buff ***skb*, u32** *len***)**

Description

Pull in non−linear data in case the *skb* is non−linear and not all of *len* are part of the linear section. Make *len* bytes from *skb* readable and writable. If a zero value is passed for *len*, then all bytes in the linear part of *skb* will be made readable and writable.

This helper is only needed for reading and writing with direct packet access.

For direct packet access, testing that offsets to access are within packet boundaries (test on *skb*−>**data_end**) is susceptible to fail if offsets are invalid, or if the requested data is in non−linear parts of the *skb*. On failure the program can just bail out, or in the case of a non−linear buffer, use a helper to make the data available. The **bpf_skb_load_bytes**() helper is a first solution to access the data. Another one consists in using **bpf_skb_pull_data** to pull in once the non−linear parts, then retesting and eventually access the data.

At the same time, this also makes sure the *skb* is uncloned, which is a necessary condition for direct write. As this needs to be an invariant for the write part only, the verifier detects writes and adds a prologue that is calling **bpf_skb_pull_data()** to effectively unclone the *skb* from the very beginning in case it is indeed cloned.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

s64 bpf_csum_update(struct sk_buff ***skb*, __wsum** *csum***)**

Description

Add the checksum *csum* into *skb*−>**csum** in case the driver has supplied a checksum for the entire packet into that field. Return an error otherwise. This helper is intended to be used in combination with **bpf_csum_diff**(), in particular when the checksum

needs to be updated after data has been written into the packet through direct packet access.

Return The checksum on success, or a negative error code in case of failure.

void bpf_set_hash_invalid(struct sk_buff *_skb_)

Description

Invalidate the current *skb*->**hash**. It can be used after mangling on headers through direct packet access, in order to indicate that the hash is outdated and to trigger a recalculation the next time the kernel tries to access this hash or when the **bpf_get_hash_recalc**() helper is called.

Return void.

long bpf_get_numa_node_id(void)

Description

Return the id of the current NUMA node. The primary use case for this helper is the selection of sockets for the local NUMA node, when the program is attached to sockets using the **SO_ATTACH_REUSEPORT_EBPF** option (see also **socket(7)**), but the helper is also available to other eBPF program types, similarly to **bpf_get_smp_processor_id**().

Return The id of current NUMA node.

long bpf_skb_change_head(struct sk_buff *_skb_, u32 _len_, u64 _flags_)

Description

Grows headroom of packet associated to *skb* and adjusts the offset of the MAC header accordingly, adding *len* bytes of space. It automatically extends and reallocates memory as required.

This helper can be used on a layer 3 *skb* to push a MAC header for redirection into a layer 2 device.

All values for *flags* are reserved for future usage, and must be left at zero.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_xdp_adjust_head(struct xdp_buff *_xdp_md_, int _delta_)

Description

Adjust (move) *xdp_md*->**data** by *delta* bytes. Note that it is possible to use a negative value for *delta*. This helper can be used to prepare the packet for pushing or popping headers.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_probe_read_str(void *_dst_, u32 _size_, const void *_unsafe_ptr_)

Description

Copy a NUL terminated string from an unsafe kernel address *unsafe_ptr* to *dst*. See **bpf_probe_read_kernel_str**() for more details.

Generally, use **bpf_probe_read_user_str**() or **bpf_probe_read_kernel_str**() instead.

Return On success, the strictly positive length of the string, including the trailing NUL character. On error, a negative value.

u64 bpf_get_socket_cookie(struct sk_buff ****skb*)**

Description

If the **struct sk_buff** pointed by *skb* has a known socket, retrieve the cookie (generated by the kernel) of this socket. If no cookie has been set yet, generate a new cookie. Once generated, the socket cookie remains stable for the life of the socket. This helper can be useful for monitoring per socket networking traffic statistics as it provides a global socket identifier that can be assumed unique.

Return A 8−byte long unique number on success, or 0 if the socket field is missing inside *skb*.

u64 bpf_get_socket_cookie(struct bpf_sock_addr ****ctx*)**

Description

Equivalent to bpf_get_socket_cookie() helper that accepts *skb*, but gets socket from **struct bpf_sock_addr** context.

Return A 8−byte long unique number.

u64 bpf_get_socket_cookie(struct bpf_sock_ops ****ctx*)**

Description

Equivalent to **bpf_get_socket_cookie**() helper that accepts *skb*, but gets socket from **struct bpf_sock_ops** context.

Return A 8−byte long unique number.

u64 bpf_get_socket_cookie(struct sock ****sk*)**

Description

Equivalent to **bpf_get_socket_cookie**() helper that accepts *sk*, but gets socket from a BTF **struct sock**. This helper also works for sleepable programs.

Return A 8−byte long unique number or 0 if *sk* is NULL.

u32 bpf_get_socket_uid(struct sk_buff ****skb*)**

Description

Get the owner UID of the socked associated to *skb*.

Return The owner UID of the socket associated to *skb*. If the socket is **NULL**, or if it is not a full socket (i.e. if it is a time−wait or a request socket instead), **overflowuid** value is returned (note that **overflowuid** might also be the actual UID value for the socket).

long bpf_set_hash(struct sk_buff ****skb*, u32** *hash***)**

Description

Set the full hash for *skb* (set the field *skb*−>**hash**) to value *hash*.

Return

long bpf_setsockopt(void ****bpf_socket*, int** *level***, int** *optname***, void ****optval***, int** *optlen***)**

Description

Emulate a call to **setsockopt()** on the socket associated to *bpf_socket*, which must be a full socket. The *level* at which the option resides and the name *optname* of the option must be specified, see **setsockopt(2)** for more information. The option value of length *optlen* is pointed by *optval*.

bpf_socket should be one of the following:

- **struct bpf_sock_ops** for **BPF_PROG_TYPE_SOCK_OPS**.
- **struct bpf_sock_addr** for **BPF_CGROUP_INET4_CONNECT**, **BPF_CGROUP_INET6_CONNECT** and **BPF_CGROUP_UNIX_CONNECT**.

This helper actually implements a subset of **setsockopt()**. It supports the following *level*s:

- **SOL_SOCKET**, which supports the following *optname*s: **SO_RCVBUF**, **SO_SNDBUF**, **SO_MAX_PACING_RATE**, **SO_PRIORITY**, **SO_RCVLOWAT**, **SO_MARK**, **SO_BINDTODEVICE**, **SO_KEEPALIVE**, **SO_REUSEADDR**, **SO_REUSEPORT**, **SO_BINDTOIFINDEX**, **SO_TXREHASH**.
- **IPPROTO_TCP**, which supports the following *optname*s: **TCP_CONGESTION**, **TCP_BPF_IW**, **TCP_BPF_SNDCWND_CLAMP**, **TCP_SAVE_SYN**, **TCP_KEEPIDLE**, **TCP_KEEPINTVL**, **TCP_KEEPCNT**, **TCP_SYNCNT**, **TCP_USER_TIMEOUT**, **TCP_NOTSENT_LOWAT**, **TCP_NODELAY**, **TCP_MAXSEG**, **TCP_WINDOW_CLAMP**, **TCP_THIN_LINEAR_TIMEOUTS**, **TCP_BPF_DELACK_MAX**, **TCP_BPF_RTO_MIN**.
- **IPPROTO_IP**, which supports *optname* **IP_TOS**.
- **IPPROTO_IPV6**, which supports the following *optname*s: **IPV6_TCLASS**, **IPV6_AUTOFLOWLABEL**.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_adjust_room(struct sk_buff **skb*, s32 *len_diff***, u32** *mode***, u64** *flags***)**

Description

Grow or shrink the room for data in the packet associated to *skb* by *len_diff*, and according to the selected *mode*.

By default, the helper will reset any offloaded checksum indicator of the skb to CHECKSUM_NONE. This can be avoided by the following flag:

- **BPF_F_ADJ_ROOM_NO_CSUM_RESET**: Do not reset offloaded checksum data of the skb to CHECKSUM_NONE.

There are two supported modes at this time:

- **BPF_ADJ_ROOM_MAC**: Adjust room at the mac layer (room space is added or removed between the layer 2 and layer 3 headers).
- **BPF_ADJ_ROOM_NET**: Adjust room at the network layer (room space is added or removed between the layer 3 and layer 4 headers).

The following flags are supported at this time:

- **BPF_F_ADJ_ROOM_FIXED_GSO**: Do not adjust gso_size. Adjusting mss in this way is not allowed for datagrams.
- **BPF_F_ADJ_ROOM_ENCAP_L3_IPV4**, **BPF_F_ADJ_ROOM_ENCAP_L3_IPV6**: Any new space is reserved to hold a tunnel header. Configure skb offsets and other fields accordingly.
- **BPF_F_ADJ_ROOM_ENCAP_L4_GRE**, **BPF_F_ADJ_ROOM_ENCAP_L4_UDP**: Use with ENCAP_L3 flags to further specify the tunnel type.
- **BPF_F_ADJ_ROOM_ENCAP_L2**(*len*): Use with ENCAP_L3/L4 flags to further specify the tunnel type; *len* is the length of the inner MAC header.
- **BPF_F_ADJ_ROOM_ENCAP_L2_ETH**: Use with BPF_F_ADJ_ROOM_ENCAP_L2 flag to further specify the L2 type as Ethernet.
- **BPF_F_ADJ_ROOM_DECAP_L3_IPV4**, **BPF_F_ADJ_ROOM_DECAP_L3_IPV6**: Indicate the new IP header version after decapsulating the outer IP header. Used when the inner and outer IP versions are different.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_redirect_map(struct bpf_map ****map*, u64** *key***, u64** *flags***)**

Description

Redirect the packet to the endpoint referenced by *map* at index *key*. Depending on its type, this *map* can contain references to net devices (for forwarding packets through other ports), or to CPUs (for redirecting XDP frames to another CPU; but this is only implemented for native XDP (with driver support) as of this writing).

The lower two bits of *flags* are used as the return code if the map lookup fails. This is so that the return value can be one of the XDP program return codes up to **XDP_TX**, as chosen by the caller. The higher bits of *flags* can be set to BPF_F_BROADCAST or BPF_F_EXCLUDE_INGRESS as defined below.

With BPF_F_BROADCAST the packet will be broadcasted to all the interfaces in the map, with BPF_F_EXCLUDE_INGRESS the ingress interface will be excluded when do broadcasting.

See also **bpf_redirect**(), which only supports redirecting to an ifindex, but doesn't require a map to do so.

Return **XDP_REDIRECT** on success, or the value of the two lower bits of the *flags* argument on error.

long bpf_sk_redirect_map(struct sk_buff ****skb*, struct bpf_map ****map***, u32** *key***, u64** *flags***)**

Description

Redirect the packet to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKMAP**) at index *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

long bpf_sock_map_update(struct bpf_sock_ops ****skops*, struct bpf_map ****map***, void ****key***, u64** *flags***)**

Description

Add an entry to, or update a *map* referencing sockets. The *skops* is used as a new value for the entry associated to *key*. *flags* is one of:

BPF_NOEXIST
The entry for *key* must not exist in the map.

BPF_EXIST
The entry for *key* must already exist in the map.

BPF_ANY
No condition on the existence of the entry for *key*.

If the *map* has eBPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to eBPF programs, this results in an error.

Return 0 on success, or a negative error in case of failure.

long bpf_xdp_adjust_meta(struct xdp_buff ****xdp_md*, int** *delta***)**

Description

Adjust the address pointed by *xdp_md*−>**data_meta** by *delta* (which can be positive or negative). Note that this operation modifies the address stored in *xdp_md*−>**data**, so the latter must be loaded only after the helper has been called.

The use of *xdp_md*−>**data_meta** is optional and programs are not required to use it. The rationale is that when the packet is processed with XDP (e.g. as DoS filter), it is

possible to push further meta data along with it before passing to the stack, and to give the guarantee that an ingress eBPF program attached as a TC classifier on the same device can pick this up for further post−processing. Since TC works with socket buffers, it remains possible to set from XDP the **mark** or **priority** pointers, or other pointers for the socket buffer. Having this scratch space generic and programmable allows for more flexibility as the user is free to store whatever meta data they need.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_perf_event_read_value(struct bpf_map ****map*, u64** *flags***, struct bpf_perf_event_value ****buf***, u32** *buf_size***)**

Description

Read the value of a perf event counter, and store it into *buf* of size *buf_size*. This helper relies on a *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. The nature of the perf event counter is selected when *map* is updated with perf event file descriptors. The *map* is an array whose size is the number of available CPUs, and each cell contains a value relative to one CPU. The value to retrieve is indicated by *flags*, that contains the index of the CPU to look up, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the value for the current CPU should be retrieved.

This helper behaves in a way close to **bpf_perf_event_read**() helper, save that instead of just returning the value observed, it fills the *buf* structure. This allows for additional data to be retrieved: in particular, the enabled and running times (in *buf*−>**enabled** and *buf*−>**running**, respectively) are copied. In general, **bpf_perf_event_read_value**() is recommended over **bpf_perf_event_read**(), which has some ABI issues and provides fewer functionalities.

These values are interesting, because hardware PMU (Performance Monitoring Unit) counters are limited resources. When there are more PMU based perf events opened than available counters, kernel will multiplex these events so each event gets certain percentage (but not all) of the PMU time. In case that multiplexing happens, the number of samples or counter value will not reflect the case compared to when no multiplexing occurs. This makes comparison between different runs difficult. Typically, the counter value should be normalized before comparing to other experiments. The usual normalization is done as follows.

```
normalized_counter = counter * t_enabled / t_running
```

Where t_enabled is the time enabled for event and t_running is the time running for event since last normalization. The enabled and running times are accumulated since the perf event open. To achieve scaling factor between two invocations of an eBPF program, users can use CPU id as the key (which is typical for perf array usage model) to remember the previous value and do the calculation inside the eBPF program.

Return 0 on success, or a negative error in case of failure.

long bpf_perf_prog_read_value(struct bpf_perf_event_data ****ctx*, struct bpf_perf_event_value ****buf***, u32** *buf_size***)**

Description

For an eBPF program attached to a perf event, retrieve the value of the event counter associated to *ctx* and store it in the structure pointed by *buf* and of size *buf_size*. Enabled and running times are also stored in the structure (see description of helper **bpf_perf_event_read_value**() for more details).

Return 0 on success, or a negative error in case of failure.

long bpf_getsockopt(void ****bpf_socket*, int** *level***, int** *optname***, void ****optval***, int** *optlen***)**

Description

Emulate a call to **getsockopt()** on the socket associated to *bpf_socket*, which must be a full socket. The *level* at which the option resides and the name *optname* of the option must be specified, see **getsockopt(2)** for more information. The retrieved value is stored in the structure pointed by *opval* and of length *optlen*.

bpf_socket should be one of the following:

- **struct bpf_sock_ops** for **BPF_PROG_TYPE_SOCK_OPS**.
- **struct bpf_sock_addr** for **BPF_CGROUP_INET4_CONNECT**, **BPF_CGROUP_INET6_CONNECT** and **BPF_CGROUP_UNIX_CONNECT**.

This helper actually implements a subset of **getsockopt()**. It supports the same set of *optname*s that is supported by the **bpf_setsockopt**() helper. The exceptions are **TCP_BPF_*** is **bpf_setsockopt**() only and **TCP_SAVED_SYN** is **bpf_getsockopt**() only.

Return 0 on success, or a negative error in case of failure.

long bpf_override_return(struct pt_regs ****regs*, u64** *rc***)**

Description

Used for error injection, this helper uses kprobes to override the return value of the probed function, and to set it to *rc*. The first argument is the context *regs* on which the kprobe works.

This helper works by setting the PC (program counter) to an override function which is run in place of the original probed function. This means the probed function is not run at all. The replacement function just returns with the required value.

This helper has security implications, and thus is subject to restrictions. It is only available if the kernel was compiled with the **CONFIG_BPF_KPROBE_OVERRIDE** configuration option, and in this case it only works on functions tagged with **ALLOW_ERROR_INJECTION** in the kernel code.

Also, the helper is only available for the architectures having the CONFIG_FUNCTION_ERROR_INJECTION option. As of this writing, x86 architecture is the only one to support this feature.

Return

long bpf_sock_ops_cb_flags_set(struct bpf_sock_ops ****bpf_sock*, int** *argval***)**

Description

Attempt to set the value of the **bpf_sock_ops_cb_flags** field for the full TCP socket associated to *bpf_sock_ops* to *argval*.

The primary use of this field is to determine if there should be calls to eBPF programs of type **BPF_PROG_TYPE_SOCK_OPS** at various points in the TCP code. A program of the same type can change its value, per connection and as necessary, when the connection is established. This field is directly accessible for reading, but this helper must be used for updates in order to return an error if an eBPF program tries to set a callback that is not supported in the current kernel.

argval is a flag array which can combine these flags:

- **BPF_SOCK_OPS_RTO_CB_FLAG** (retransmission time out)
- **BPF_SOCK_OPS_RETRANS_CB_FLAG** (retransmission)

- **BPF_SOCK_OPS_STATE_CB_FLAG** (TCP state change)
- **BPF_SOCK_OPS_RTT_CB_FLAG** (every RTT)

Therefore, this function can be used to clear a callback flag by setting the appropriate bit to zero. e.g. to disable the RTO callback:

bpf_sock_ops_cb_flags_set(bpf_sock,
bpf_sock−>bpf_sock_ops_cb_flags &
~BPF_SOCK_OPS_RTO_CB_FLAG)

Here are some examples of where one could call such eBPF program:

- When RTO fires.
- When a packet is retransmitted.
- When the connection terminates.
- When a packet is sent.
- When a packet is received.

Return Code **−EINVAL** if the socket is not a full TCP socket; otherwise, a positive number containing the bits that could not be set is returned (which comes down to 0 if all bits were set as required).

long bpf_msg_redirect_map(struct sk_msg_buff ***msg*, struct bpf_map ***map***, u32** *key***, u64** *flags***)**

Description

This helper is used in programs implementing policies at the socket level. If the message *msg* is allowed to pass (i.e. if the verdict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKMAP**) at index *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

long bpf_msg_apply_bytes(struct sk_msg_buff ***msg*, u32** *bytes***)**

Description

For socket policies, apply the verdict of the eBPF program to the next *bytes* (number of bytes) of message *msg*.

For example, this helper can be used in the following cases:

- A single **sendmsg**() or **sendfile**() system call contains multiple logical messages that the eBPF program is supposed to read and for which it should apply a verdict.
- An eBPF program only cares to read the first *bytes* of a *msg*. If the message has a large payload, then setting up and calling the eBPF program repeatedly for all bytes, even though the verdict is already known, would create unnecessary overhead.

When called from within an eBPF program, the helper sets a counter internal to the BPF infrastructure, that is used to apply the last verdict to the next *bytes*. If *bytes* is smaller than the current data being processed from a **sendmsg**() or **sendfile**() system call, the first *bytes* will be sent and the eBPF program will be re−run with the pointer for start of data pointing to byte number *bytes* **+ 1**. If *bytes* is larger than the current data being processed, then the eBPF verdict will be applied to multiple **sendmsg**() or **sendfile**() calls until *bytes* are consumed.

Note that if a socket closes with the internal counter holding a non−zero value, this is not a problem because data is not being buffered for *bytes* and is sent as it is received.

Return

long bpf_msg_cork_bytes(struct sk_msg_buff ****msg*, u32** *bytes***)**

Description

For socket policies, prevent the execution of the verdict eBPF program for message *msg* until *bytes* (byte number) have been accumulated.

This can be used when one needs a specific number of bytes before a verdict can be assigned, even if the data spans multiple **sendmsg**() or **sendfile**() calls. The extreme case would be a user calling **sendmsg**() repeatedly with 1−byte long message segments. Obviously, this is bad for performance, but it is still valid. If the eBPF program needs *bytes* bytes to validate a header, this helper can be used to prevent the eBPF program to be called again until *bytes* have been accumulated.

Return

long bpf_msg_pull_data(struct sk_msg_buff ****msg*, u32** *start***, u32** *end***, u64** *flags***)**

Description

For socket policies, pull in non−linear data from user space for *msg* and set pointers *msg*−>**data** and *msg*−>**data_end** to *start* and *end* bytes offsets into *msg*, respectively.

If a program of type **BPF_PROG_TYPE_SK_MSG** is run on a *msg* it can only parse data that the (**data**, **data_end**) pointers have already consumed. For **sendmsg**() hooks this is likely the first scatterlist element. But for calls relying on the **sendpage** handler (e.g. **sendfile**()) this will be the range (**0**, **0**) because the data is shared with user space and by default the objective is to avoid allowing user space to modify data while (or after) eBPF verdict is being decided. This helper can be used to pull in data and to set the start and end pointer to given values. Data will be copied if necessary (i.e. if data was not linear and if start and end pointers do not point to the same chunk).

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

All values for *flags* are reserved for future usage, and must be left at zero.

Return 0 on success, or a negative error in case of failure.

long bpf_bind(struct bpf_sock_addr ****ctx*, struct sockaddr ****addr***, int** *addr_len***)**

Description

Bind the socket associated to *ctx* to the address pointed by *addr*, of length *addr_len*. This allows for making outgoing connection from the desired IP address, which can be useful for example when all processes inside a cgroup should use one single IP address on a host that has multiple IP configured.

This helper works for IPv4 and IPv6, TCP and UDP sockets. The domain (*addr*−>**sa_family**) must be **AF_INET** (or **AF_INET6**). It's advised to pass zero port (**sin_port** or **sin6_port**) which triggers IP_BIND_ADDRESS_NO_PORT−like behavior and lets the kernel efficiently pick up an unused port as long as 4−tuple is unique. Passing non−zero port might lead to degraded performance.

Return 0 on success, or a negative error in case of failure.

long bpf_xdp_adjust_tail(struct xdp_buff ****xdp_md*, int** *delta***)**

Description

Adjust (move) *xdp_md*−>**data_end** by *delta* bytes. It is possible to both shrink and grow the packet tail. Shrink done via *delta* being a negative integer.

A call to this helper is susceptible to change the underlying packet buffer. Therefore,

at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_skb_get_xfrm_state(struct sk_buff *_skb_, u32 _index_, struct bpf_xfrm_state *_xfrm_state_, u32 _size_, u64 _flags_)

Description

Retrieve the XFRM state (IP transform framework, see also **ip−xfrm(8)**) at *index* in XFRM "security path" for *skb*.

The retrieved value is stored in the **struct bpf_xfrm_state** pointed by *xfrm_state* and of length *size*.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_XFRM** configuration option.

Return 0 on success, or a negative error in case of failure.

long bpf_get_stack(void *_ctx_, void *_buf_, u32 _size_, u64 _flags_)

Description

Return a user or a kernel stack in bpf program provided buffer. To achieve this, the helper needs *ctx*, which is a pointer to the context on which the tracing program is executed. To store the stacktrace, the bpf program provides *buf* with a nonnegative *size*.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set the following flags:

BPF_F_USER_STACK

Collect a user space stack instead of a kernel stack.

BPF_F_USER_BUILD_ID

Collect (build_id, file_offset) instead of ips for user stack, only valid if **BPF_F_USER_STACK** is also specified.

file_offset is an offset relative to the beginning of the executable or shared object file backing the vma which the *ip* falls in. It is *not* an offset relative to that object's base address. Accordingly, it must be adjusted by adding (sh_addr − sh_offset), where sh_{addr,offset} correspond to the executable section containing *file_offset* in the object, for comparisons to symbols' st_value to be valid.

bpf_get_stack() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames, subject to sufficient large buffer size. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

```
# sysctl kernel.perf_event_max_stack=<new value>
```

Return The non−negative copied *buf* length equal to or less than *size* on success, or a negative error in case of failure.

long bpf_skb_load_bytes_relative(const void *_skb_, u32 _offset_, void *_to_, u32 _len_, u32 _start_header_)

Description

This helper is similar to **bpf_skb_load_bytes**() in that it provides an easy way to load *len* bytes from *offset* from the packet associated to *skb*, into the buffer pointed by *to*. The difference to **bpf_skb_load_bytes**() is that a fifth argument *start_header* exists in order to select a base offset to start from. *start_header* can be one of:

BPF_HDR_START_MAC
Base offset to load data from is *skb*'s mac header.

BPF_HDR_START_NET
Base offset to load data from is *skb*'s network header.

In general, "direct packet access" is the preferred method to access packet data, however, this helper is in particular useful in socket filters where *skb*−**>data** does not always point to the start of the mac header and where "direct packet access" is not available.

Return 0 on success, or a negative error in case of failure.

long bpf_fib_lookup(void *_ctx_, struct bpf_fib_lookup *_params_, int _plen_, u32 _flags_)

Description

Do FIB lookup in kernel tables using parameters in *params*. If lookup is successful and result shows packet is to be forwarded, the neighbor tables are searched for the nexthop. If successful (ie., FIB lookup shows forwarding and nexthop is resolved), the nexthop address is returned in ipv4_dst or ipv6_dst based on family, smac is set to mac address of egress device, dmac is set to nexthop mac address, rt_metric is set to metric from route (IPv4/IPv6 only), and ifindex is set to the device index of the nexthop from the FIB lookup.

plen argument is the size of the passed in struct. *flags* argument can be a combination of one or more of the following values:

BPF_FIB_LOOKUP_DIRECT
Do a direct table lookup vs full lookup using FIB rules.

BPF_FIB_LOOKUP_TBID
Used with BPF_FIB_LOOKUP_DIRECT. Use the routing table ID present in *params*−>tbid for the fib lookup.

BPF_FIB_LOOKUP_OUTPUT
Perform lookup from an egress perspective (default is ingress).

BPF_FIB_LOOKUP_SKIP_NEIGH
Skip the neighbour table lookup. *params*−>dmac and *params*−>smac will not be set as output. A common use case is to call **bpf_redirect_neigh**() after doing **bpf_fib_lookup**().

BPF_FIB_LOOKUP_SRC
Derive and set source IP addr in *params*−>ipv{4,6}_src for the nexthop. If the src addr cannot be derived, **BPF_FIB_LKUP_RET_NO_SRC_ADDR** is returned. In this case, *params*−>dmac and *params*−>smac are not set either.

ctx is either **struct xdp_md** for XDP programs or **struct sk_buff** tc cls_act programs.

Return

- < 0 if any input argument is invalid
- 0 on success (packet is forwarded, nexthop neighbor exists)
- > 0 one of **BPF_FIB_LKUP_RET_** codes explaining why the packet is not forwarded or needs assist from full stack

If lookup fails with BPF_FIB_LKUP_RET_FRAG_NEEDED, then the MTU was exceeded and output params−>mtu_result contains the MTU.

long bpf_sock_hash_update(struct bpf_sock_ops *_skops_, struct bpf_map *_map_, void *_key_, u64 _flags_)

Description

Add an entry to, or update a sockhash *map* referencing sockets. The *skops* is used as a new value for the entry associated to *key*. *flags* is one of:

BPF_NOEXIST
The entry for *key* must not exist in the map.

BPF_EXIST
The entry for *key* must already exist in the map.

BPF_ANY
No condition on the existence of the entry for *key*.

If the *map* has eBPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to eBPF programs, this results in an error.

Return 0 on success, or a negative error in case of failure.

long bpf_msg_redirect_hash(struct sk_msg_buff ****msg*, struct bpf_map ****map***, void ****key***, u64** *flags***)**

Description
This helper is used in programs implementing policies at the socket level. If the message *msg* is allowed to pass (i.e. if the verdict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKHASH**) using hash *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

long bpf_sk_redirect_hash(struct sk_buff ****skb*, struct bpf_map ****map***, void ****key***, u64** *flags***)**

Description
This helper is used in programs implementing policies at the skb socket level. If the sk_buff *skb* is allowed to pass (i.e. if the verdict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKHASH**) using hash *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

long bpf_lwt_push_encap(struct sk_buff ****skb*, u32** *type***, void ****hdr***, u32** *len***)**

Description
Encapsulate the packet associated to *skb* within a Layer 3 protocol header. This header is provided in the buffer at address *hdr*, with *len* its size in bytes. *type* indicates the protocol of the header and can be one of:

BPF_LWT_ENCAP_SEG6
IPv6 encapsulation with Segment Routing Header (**struct ipv6_sr_hdr**). *hdr* only contains the SRH, the IPv6 header is computed by the kernel.

BPF_LWT_ENCAP_SEG6_INLINE
Only works if *skb* contains an IPv6 packet. Insert a Segment Routing Header (**struct ipv6_sr_hdr**) inside the IPv6 header.

BPF_LWT_ENCAP_IP
IP encapsulation (GRE/GUE/IPIP/etc). The outer header must be IPv4 or IPv6, followed by zero or more additional headers, up to **LWT_BPF_MAX_HEADROOM** total bytes in all prepended headers. Please note that if **skb_is_gso**(*skb*) is true, no more than two headers can be prepended, and the inner header, if present, should be either GRE or UDP/GUE.

BPF_LWT_ENCAP_SEG6* types can be called by BPF programs of type **BPF_PROG_TYPE_LWT_IN**; **BPF_LWT_ENCAP_IP** type can be called by bpf programs of types **BPF_PROG_TYPE_LWT_IN** and

BPF_PROG_TYPE_LWT_XMIT.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_lwt_seg6_store_bytes(struct sk_buff ****skb*, u32** *offset***, const void *** *from***, u32** *len***)**

Description

Store *len* bytes from address *from* into the packet associated to *skb*, at *offset*. Only the flags, tag and TLVs inside the outermost IPv6 Segment Routing Header can be modified through this helper.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_lwt_seg6_adjust_srh(struct sk_buff * *skb***, u32** *offset***, s32** *delta***)**

Description

Adjust the size allocated to TLVs in the outermost IPv6 Segment Routing Header contained in the packet associated to *skb*, at position *offset* by *delta* bytes. Only offsets after the segments are accepted. *delta* can be as well positive (growing) as negative (shrinking).

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_lwt_seg6_action(struct sk_buff * *skb***, u32** *action***, void *** *param***, u32** *param_len***)**

Description

Apply an IPv6 Segment Routing action of type *action* to the packet associated to *skb*. Each action takes a parameter contained at address *param*, and of length *param_len* bytes. *action* can be one of:

SEG6_LOCAL_ACTION_END_X
: End.X action: Endpoint with Layer−3 cross−connect. Type of *param*: **struct in6_addr**.

SEG6_LOCAL_ACTION_END_T
: End.T action: Endpoint with specific IPv6 table lookup. Type of *param*: **int**.

SEG6_LOCAL_ACTION_END_B6
: End.B6 action: Endpoint bound to an SRv6 policy. Type of *param*: **struct ipv6_sr_hdr**.

SEG6_LOCAL_ACTION_END_B6_ENCAP
: End.B6.Encap action: Endpoint bound to an SRv6 encapsulation policy. Type of *param*: **struct ipv6_sr_hdr**.

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

long bpf_rc_repeat(void ****ctx*)**

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded repeat key message. This delays the generation of a key up event for previously generated key down event.

Some IR protocols like NEC have a special IR message for repeating last button, for when a button is held down.

The *ctx* should point to the lirc sample as passed into the program.

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

long bpf_rc_keydown(void ****ctx*, u32** *protocol***, u64** *scancode***, u32** *toggle***)**

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded key press with *scancode*, *toggle* value in the given *protocol*. The scancode will be translated to a keycode using the rc keymap, and reported as an input key down event. After a period a key up event is generated. This period can be extended by calling either **bpf_rc_keydown**() again with the same values, or calling **bpf_rc_repeat**().

Some protocols include a toggle bit, in case the button was released and pressed again between consecutive scancodes.

The *ctx* should point to the lirc sample as passed into the program.

The *protocol* is the decoded protocol number (see **enum rc_proto** for some predefined values).

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

u64 bpf_skb_cgroup_id(struct sk_buff ****skb*)**

Description

Return the cgroup v2 id of the socket associated with the *skb*. This is roughly similar to the **bpf_get_cgroup_classid**() helper for cgroup v1 by providing a tag resp. identifier that can be matched on or used for map lookups e.g. to implement policy. The cgroup v2 id of a given path in the hierarchy is exposed in user space through the f_handle API in order to get to the same 64−bit id.

This helper can be used on TC egress path, but not on ingress, and is available only if the kernel was compiled with the **CONFIG_SOCK_CGROUP_DATA** configuration option.

Return The id is returned or 0 in case the id could not be retrieved.

u64 bpf_get_current_cgroup_id(void)

Description

Get the current cgroup id based on the cgroup within which the current task is running.

Return A 64−bit integer containing the current cgroup id based on the cgroup within which the current task is running.

void *bpf_get_local_storage(void ***map*, u64** *flags***)**

Description

Get the pointer to the local storage area. The type and the size of the local storage is defined by the *map* argument. The *flags* meaning is specific for each map type, and has to be 0 for cgroup local storage.

Depending on the BPF program type, a local storage area can be shared between multiple instances of the BPF program, running simultaneously.

A user should care about the synchronization by himself. For example, by using the **BPF_ATOMIC** instructions to alter the shared data.

Return A pointer to the local storage area.

long bpf_sk_select_reuseport(struct sk_reuseport_md ***reuse*, struct bpf_map ***map***, void ***key***, u64** *flags***)**

Description

Select a **SO_REUSEPORT** socket from a **BPF_MAP_TYPE_REUSEPORT_SOCKARRAY** *map*. It checks the selected socket is matching the incoming request in the socket buffer.

Return 0 on success, or a negative error in case of failure.

u64 bpf_skb_ancestor_cgroup_id(struct sk_buff ***skb*, int** *ancestor_level***)**

Description

Return id of cgroup v2 that is ancestor of cgroup associated with the *skb* at the *ancestor_level*. The root cgroup is at *ancestor_level* zero and each step down the hierarchy increments the level. If *ancestor_level* == level of cgroup associated with *skb*, then return value will be same as that of **bpf_skb_cgroup_id**().

The helper is useful to implement policies based on cgroups that are upper in hierarchy than immediate cgroup associated with *skb*.

The format of returned id and helper limitations are same as in **bpf_skb_cgroup_id**().

Return The id is returned or 0 in case the id could not be retrieved.

struct bpf_sock *bpf_sk_lookup_tcp(void ***ctx*, struct bpf_sock_tuple ***tuple***, u32** *tuple_size***, u64** *netns***, u64** *flags***)**

Description

Look for TCP socket matching *tuple*, optionally in a child network namespace *netns*. The return value must be checked, and if non−**NULL**, released via **bpf_sk_release**().

The *ctx* should point to the context of the program, such as the skb or socket (depending on the hook in use). This is used to determine the base network namespace for the lookup.

tuple_size must be one of:

sizeof(*tuple***−>ipv4)**
Look for an IPv4 socket.

sizeof(*tuple***−>ipv6)**
Look for an IPv6 socket.

If the *netns* is a negative signed 32−bit integer, then the socket lookup table in the netns associated with the *ctx* will be used. For the TC hooks, this is the netns of the device in the skb. For socket hooks, this is the netns of the socket. If *netns* is any other signed 32−bit value greater than or equal to zero then it specifies the ID of the netns relative to the netns associated with the *ctx*. *netns* values beyond the range of 32−bit integers are reserved for future use.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_NET** configuration option.

Return Pointer to **struct bpf_sock**, or **NULL** in case of failure. For sockets with reuseport option, the **struct bpf_sock** result is from *reuse*->**socks**[] using the hash of the tuple.

struct bpf_sock *bpf_sk_lookup_udp(void *_ctx_, struct bpf_sock_tuple *_tuple_, u32 _tuple_size_, u64 _netns_, u64 _flags_)

Description

Look for UDP socket matching *tuple*, optionally in a child network namespace *netns*. The return value must be checked, and if non−**NULL**, released via **bpf_sk_release**().

The *ctx* should point to the context of the program, such as the skb or socket (depending on the hook in use). This is used to determine the base network namespace for the lookup.

tuple_size must be one of:

sizeof(_tuple_->**ipv4)**
Look for an IPv4 socket.

sizeof(_tuple_->**ipv6)**
Look for an IPv6 socket.

If the *netns* is a negative signed 32−bit integer, then the socket lookup table in the netns associated with the *ctx* will be used. For the TC hooks, this is the netns of the device in the skb. For socket hooks, this is the netns of the socket. If *netns* is any other signed 32−bit value greater than or equal to zero then it specifies the ID of the netns relative to the netns associated with the *ctx*. *netns* values beyond the range of 32−bit integers are reserved for future use.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_NET** configuration option.

Return Pointer to **struct bpf_sock**, or **NULL** in case of failure. For sockets with reuseport option, the **struct bpf_sock** result is from *reuse*->**socks**[] using the hash of the tuple.

long bpf_sk_release(void *_sock_)

Description

Release the reference held by *sock*. *sock* must be a non−**NULL** pointer that was returned from **bpf_sk_lookup_xxx**().

Return 0 on success, or a negative error in case of failure.

long bpf_map_push_elem(struct bpf_map *_map_, const void *_value_, u64 _flags_)

Description

Push an element *value* in *map*. *flags* is one of:

BPF_EXIST
If the queue/stack is full, the oldest element is removed to make room for this.

Return 0 on success, or a negative error in case of failure.

long bpf_map_pop_elem(struct bpf_map *_map_, void *_value_)

Description

Pop an element from *map*.

Return 0 on success, or a negative error in case of failure.

long bpf_map_peek_elem(struct bpf_map *_map_, void *_value_)

Description

Get an element from *map* without removing it.

Return 0 on success, or a negative error in case of failure.

long bpf_msg_push_data(struct sk_msg_buff *_msg_, u32 _start_, u32 _len_, u64 _flags_)

Description

For socket policies, insert *len* bytes into *msg* at offset *start*.

If a program of type **BPF_PROG_TYPE_SK_MSG** is run on a *msg* it may want to insert metadata or options into the *msg*. This can later be read and used by any of the lower layer BPF hooks.

This helper may fail if under memory pressure (a malloc fails) in these cases BPF programs will get an appropriate error and BPF programs will need to handle them.

Return 0 on success, or a negative error in case of failure.

long bpf_msg_pop_data(struct sk_msg_buff *_msg_, u32 _start_, u32 _len_, u64 _flags_)

Description

Will remove *len* bytes from a *msg* starting at byte *start*. This may result in **ENOMEM** errors under certain situations if an allocation and copy are required due to a full ring buffer. However, the helper will try to avoid doing the allocation if possible. Other errors can occur if input parameters are invalid either due to *start* byte not being valid part of *msg* payload and/or *pop* value being to large.

Return 0 on success, or a negative error in case of failure.

long bpf_rc_pointer_rel(void *_ctx_, s32 _rel_x_, s32 _rel_y_)

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded pointer movement.

The *ctx* should point to the lirc sample as passed into the program.

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

long bpf_spin_lock(struct bpf_spin_lock *_lock_)

Description

Acquire a spinlock represented by the pointer *lock*, which is stored as part of a value of a map. Taking the lock allows to safely update the rest of the fields in that value. The spinlock can (and must) later be released with a call to **bpf_spin_unlock**(*lock*).

Spinlocks in BPF programs come with a number of restrictions and constraints:

- **bpf_spin_lock** objects are only allowed inside maps of types **BPF_MAP_TYPE_HASH** and **BPF_MAP_TYPE_ARRAY** (this list could be extended in the future).
- BTF description of the map is mandatory.
- The BPF program can take ONE lock at a time, since taking two or more could cause dead locks.
- Only one **struct bpf_spin_lock** is allowed per map element.
- When the lock is taken, calls (either BPF to BPF or helpers) are not allowed.

- The **BPF_LD_ABS** and **BPF_LD_IND** instructions are not allowed inside a spin-lock−ed region.
- The BPF program MUST call **bpf_spin_unlock**() to release the lock, on all execution paths, before it returns.
- The BPF program can access **struct bpf_spin_lock** only via the **bpf_spin_lock**() and **bpf_spin_unlock**() helpers. Loading or storing data into the **struct bpf_spin_lock** *lock***;** field of a map is not allowed.
- To use the **bpf_spin_lock**() helper, the BTF description of the map value must be a struct and have **struct bpf_spin_lock** *anyname***;** field at the top level. Nested lock inside another struct is not allowed.
- The **struct bpf_spin_lock** *lock* field in a map value must be aligned on a multiple of 4 bytes in that value.
- Syscall with command **BPF_MAP_LOOKUP_ELEM** does not copy the **bpf_spin_lock** field to user space.
- Syscall with command **BPF_MAP_UPDATE_ELEM**, or update from a BPF program, do not update the **bpf_spin_lock** field.
- **bpf_spin_lock** cannot be on the stack or inside a networking packet (it can only be inside of a map values).
- **bpf_spin_lock** is available to root only.
- Tracing programs and socket filter programs cannot use **bpf_spin_lock**() due to insufficient preemption checks (but this may change in the future).
- **bpf_spin_lock** is not allowed in inner maps of map−in−map.

Return

long bpf_spin_unlock(struct bpf_spin_lock ***_lock_)**

Description

Release the *lock* previously locked by a call to **bpf_spin_lock**(*lock*).

Return

struct bpf_sock *bpf_sk_fullsock(struct bpf_sock ***_sk_)**

Description

This helper gets a **struct bpf_sock** pointer such that all the fields in this **bpf_sock** can be accessed.

Return A **struct bpf_sock** pointer on success, or **NULL** in case of failure.

struct bpf_tcp_sock *bpf_tcp_sock(struct bpf_sock ***_sk_)**

Description

This helper gets a **struct bpf_tcp_sock** pointer from a **struct bpf_sock** pointer.

Return A **struct bpf_tcp_sock** pointer on success, or **NULL** in case of failure.

long bpf_skb_ecn_set_ce(struct sk_buff ***_skb_)**

Description

Set ECN (Explicit Congestion Notification) field of IP header to **CE** (Congestion Encountered) if current value is **ECT** (ECN Capable Transport). Otherwise, do nothing. Works with IPv6 and IPv4.

Return 1 if the **CE** flag is set (either by the current helper call or because it was already present), 0 if it is not set.

struct bpf_sock *bpf_get_listener_sock(struct bpf_sock ***_sk_)**

Description

Return a **struct bpf_sock** pointer in **TCP_LISTEN** state. **bpf_sk_release**() is unnecessary and not allowed.

Return A **struct bpf_sock** pointer on success, or **NULL** in case of failure.

struct bpf_sock *bpf_skc_lookup_tcp(void *_ctx_, struct bpf_sock_tuple *_tuple_, u32 _tuple_size_, u64 _netns_, u64 _flags_)

Description

Look for TCP socket matching *tuple*, optionally in a child network namespace *netns*. The return value must be checked, and if non−**NULL**, released via **bpf_sk_release**().

This function is identical to **bpf_sk_lookup_tcp**(), except that it also returns timewait or request sockets. Use **bpf_sk_fullsock**() or **bpf_tcp_sock**() to access the full structure.

This helper is available only if the kernel was compiled with **CONFIG_NET** configuration option.

Return Pointer to **struct bpf_sock**, or **NULL** in case of failure. For sockets with reuseport option, the **struct bpf_sock** result is from *reuse*−>**socks**[] using the hash of the tuple.

long bpf_tcp_check_syncookie(void *_sk_, void *_iph_, u32 _iph_len_, struct tcphdr *_th_, u32 _th_len_)

Description

Check whether *iph* and *th* contain a valid SYN cookie ACK for the listening socket in *sk*.

iph points to the start of the IPv4 or IPv6 header, while *iph_len* contains **sizeof**(**struct iphdr**) or **sizeof**(**struct ipv6hdr**).

th points to the start of the TCP header, while *th_len* contains the length of the TCP header (at least **sizeof**(**struct tcphdr**)).

Return 0 if *iph* and *th* are a valid SYN cookie ACK, or a negative error otherwise.

long bpf_sysctl_get_name(struct bpf_sysctl *_ctx_, char *_buf_, size_t _buf_len_, u64 _flags_)

Description

Get name of sysctl in /proc/sys/ and copy it into provided by program buffer *buf* of size *buf_len*.

The buffer is always NUL terminated, unless it's zero−sized.

If *flags* is zero, full name (e.g. "net/ipv4/tcp_mem") is copied. Use **BPF_F_SYSCTL_BASE_NAME** flag to copy base name only (e.g. "tcp_mem").

Return Number of character copied (not including the trailing NUL).

−E2BIG if the buffer wasn't big enough (*buf* will contain truncated name in this case).

long bpf_sysctl_get_current_value(struct bpf_sysctl *_ctx_, char *_buf_, size_t _buf_len_)

Description

Get current value of sysctl as it is presented in /proc/sys (incl. newline, etc), and copy it as a string into provided by program buffer *buf* of size *buf_len*.

The whole value is copied, no matter what file position user space issued e.g. sys_read at.

The buffer is always NUL terminated, unless it's zero−sized.

Return Number of character copied (not including the trailing NUL).

−E2BIG if the buffer wasn't big enough (*buf* will contain truncated name in this case).

−EINVAL if current value was unavailable, e.g. because sysctl is uninitialized and

read returns −EIO for it.

long bpf_sysctl_get_new_value(struct bpf_sysctl ****ctx*, char ****buf***, size_t** *buf_len***)**

Description

Get new value being written by user space to sysctl (before the actual write happens) and copy it as a string into provided by program buffer *buf* of size *buf_len*.

User space may write new value at file position > 0.

The buffer is always NUL terminated, unless it's zero−sized.

Return Number of character copied (not including the trailing NUL).

−**E2BIG** if the buffer wasn't big enough (*buf* will contain truncated name in this case).

−**EINVAL** if sysctl is being read.

long bpf_sysctl_set_new_value(struct bpf_sysctl ****ctx*, const char ****buf***, size_t** *buf_len***)**

Description

Override new value being written by user space to sysctl with value provided by program in buffer *buf* of size *buf_len*.

buf should contain a string in same form as provided by user space on sysctl write.

User space may write new value at file position > 0. To override the whole sysctl value file position should be set to zero.

Return 0 on success.

−**E2BIG** if the *buf_len* is too big.

−**EINVAL** if sysctl is being read.

long bpf_strtol(const char ****buf*, size_t** *buf_len***, u64** *flags***, long ****res***)**

Description

Convert the initial part of the string from buffer *buf* of size *buf_len* to a long integer according to the given base and save the result in *res*.

The string may begin with an arbitrary amount of white space (as determined by **isspace**(3)) followed by a single optional '−' sign.

Five least significant bits of *flags* encode base, other bits are currently unused.

Base must be either 8, 10, 16 or 0 to detect it automatically similar to user space **strtol**(3).

Return Number of characters consumed on success. Must be positive but no more than *buf_len*.

−**EINVAL** if no valid digits were found or unsupported base was provided.

−**ERANGE** if resulting value was out of range.

long bpf_strtoul(const char ****buf*, size_t** *buf_len***, u64** *flags***, unsigned long ****res***)**

Description

Convert the initial part of the string from buffer *buf* of size *buf_len* to an unsigned long integer according to the given base and save the result in *res*.

The string may begin with an arbitrary amount of white space (as determined by **isspace**(3)).

Five least significant bits of *flags* encode base, other bits are currently unused.

Base must be either 8, 10, 16 or 0 to detect it automatically similar to user space **strtoul**(3).

Return Number of characters consumed on success. Must be positive but no more than *buf_len*.

−**EINVAL** if no valid digits were found or unsupported base was provided.

−**ERANGE** if resulting value was out of range.

void *bpf_sk_storage_get(struct bpf_map ****map*, void ****sk***, void ****value***, u64** *flags***)**

Description

Get a bpf−local−storage from a *sk*.

Logically, it could be thought of getting the value from a *map* with *sk* as the **key**. From this perspective, the usage is not much different from **bpf_map_lookup_elem**(*map*, **&***sk*) except this helper enforces the key must be a full socket and the map must be a **BPF_MAP_TYPE_SK_STORAGE** also.

Underneath, the value is stored locally at *sk* instead of the *map*. The *map* is used as the bpf−local−storage "type". The bpf−local−storage "type" (i.e. the *map*) is searched against all bpf−local−storages residing at *sk*.

sk is a kernel **struct sock** pointer for LSM program. *sk* is a **struct bpf_sock** pointer for other program types.

An optional *flags* (**BPF_SK_STORAGE_GET_F_CREATE**) can be used such that a new bpf−local−storage will be created if one does not exist. *value* can be used together with **BPF_SK_STORAGE_GET_F_CREATE** to specify the initial value of a bpf−local−storage. If *value* is **NULL**, the new bpf−local−storage will be zero initialized.

Return A bpf−local−storage pointer is returned on success.

NULL if not found or there was an error in adding a new bpf−local−storage.

long bpf_sk_storage_delete(struct bpf_map ****map*, void ****sk***)**

Description

Delete a bpf−local−storage from a *sk*.

Return 0 on success.

−**ENOENT** if the bpf−local−storage cannot be found. −**EINVAL** if sk is not a fullsock (e.g. a request_sock).

long bpf_send_signal(u32 *sig***)**

Description

Send signal *sig* to the process of the current task. The signal may be delivered to any of this process's threads.

Return 0 on success or successfully queued.

−**EBUSY** if work queue under nmi is full.

−**EINVAL** if *sig* is invalid.

−**EPERM** if no permission to send the *sig*.

−**EAGAIN** if bpf program can try again.

s64 bpf_tcp_gen_syncookie(void *_sk_, void *_iph_, u32 _iph_len_, struct tcphdr *_th_, u32 _th_len_)

Description

Try to issue a SYN cookie for the packet with corresponding IP/TCP headers, *iph* and *th*, on the listening socket in *sk*.

iph points to the start of the IPv4 or IPv6 header, while *iph_len* contains **sizeof**(**struct iphdr**) or **sizeof**(**struct ipv6hdr**).

th points to the start of the TCP header, while *th_len* contains the length of the TCP header with options (at least **sizeof**(**struct tcphdr**)).

Return On success, lower 32 bits hold the generated SYN cookie in followed by 16 bits which hold the MSS value for that cookie, and the top 16 bits are unused.

On failure, the returned value is one of the following:

−**EINVAL** SYN cookie cannot be issued due to error

−**ENOENT** SYN cookie should not be issued (no SYN flood)

−**EOPNOTSUPP** kernel configuration does not enable SYN cookies

−**EPROTONOSUPPORT** IP packet version is not 4 or 6

long bpf_skb_output(void *_ctx_, struct bpf_map *_map_, u64 _flags_, void *_data_, u64 _size_)

Description

Write raw *data* blob into a special BPF perf event held by *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. This perf event must have the following attributes: **PERF_SAMPLE_RAW** as **sample_type**, **PERF_TYPE_SOFTWARE** as **type**, and **PERF_COUNT_SW_BPF_OUTPUT** as **config**.

The *flags* are used to indicate the index in *map* for which the value must be put, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the index of the current CPU core should be used.

The value to write, of *size*, is passed through eBPF stack and pointed by *data*.

ctx is a pointer to in−kernel struct sk_buff.

This helper is similar to **bpf_perf_event_output**() but restricted to raw_tracepoint bpf programs.

Return 0 on success, or a negative error in case of failure.

long bpf_probe_read_user(void *_dst_, u32 _size_, const void *_unsafe_ptr_)

Description

Safely attempt to read *size* bytes from user space address *unsafe_ptr* and store the data in *dst*.

Return 0 on success, or a negative error in case of failure.

long bpf_probe_read_kernel(void *_dst_, u32 _size_, const void *_unsafe_ptr_)

Description

Safely attempt to read *size* bytes from kernel space address *unsafe_ptr* and store the data in *dst*.

Return 0 on success, or a negative error in case of failure.

long bpf_probe_read_user_str(void *_dst_, u32 _size_, const void *_unsafe_ptr_)

Description

Copy a NUL terminated string from an unsafe user address *unsafe_ptr* to *dst*. The *size* should include the terminating NUL byte. In case the string length is smaller than *size*, the target is not padded with further NUL bytes. If the string length is larger than *size*, just *size*−1 bytes are copied and the last byte is set to NUL.

On success, returns the number of bytes that were written, including the terminal NUL. This makes this helper useful in tracing programs for reading strings, and more importantly to get its length at runtime. See the following snippet:

```
SEC("kprobe/sys_open")
void bpf_sys_open(struct pt_regs *ctx)
{
        char buf[PATHLEN]; // PATHLEN is defined to 256
        int res = bpf_probe_read_user_str(buf, sizeof(buf),
                                          ctx->di);
        // Consume buf, for example push it to
        // userspace via bpf_perf_event_output(); we
        // can use res (the string length) as event
        // size, after checking its boundaries.
}
```

In comparison, using **bpf_probe_read_user**() helper here instead to read the string would require to estimate the length at compile time, and would often result in copying more memory than necessary.

Another useful use case is when parsing individual process arguments or individual environment variables navigating *current***->mm->arg_start** and *current*−**>mm−>env_start**: using this helper and the return value, one can quickly iterate at the right offset of the memory area.

Return On success, the strictly positive length of the output string, including the trailing NUL character. On error, a negative value.

long bpf_probe_read_kernel_str(void *****dst*, **u32 *size*, **const void *****unsafe_ptr*)

Description

Copy a NUL terminated string from an unsafe kernel address *unsafe_ptr* to *dst*. Same semantics as with **bpf_probe_read_user_str**() apply.

Return On success, the strictly positive length of the string, including the trailing NUL character. On error, a negative value.

long bpf_tcp_send_ack(void *****tp*, **u32 *rcv_nxt*)

Description

Send out a tcp−ack. *tp* is the in−kernel struct **tcp_sock**. *rcv_nxt* is the ack_seq to be sent out.

Return 0 on success, or a negative error in case of failure.

long bpf_send_signal_thread(u32 *sig*)

Description

Send signal *sig* to the thread corresponding to the current task.

Return 0 on success or successfully queued.

−**EBUSY** if work queue under nmi is full.

−**EINVAL** if *sig* is invalid.

−**EPERM** if no permission to send the *sig*.

−**EAGAIN** if bpf program can try again.

u64 bpf_jiffies64(void)

Description

Obtain the 64bit jiffies

Return The 64 bit jiffies

long bpf_read_branch_records(struct bpf_perf_event_data *_ctx_, void *_buf_, u32 _size_, u64 _flags_)

Description

For an eBPF program attached to a perf event, retrieve the branch records (**struct perf_branch_entry**) associated to *ctx* and store it in the buffer pointed by *buf* up to size *size* bytes.

Return On success, number of bytes written to *buf*. On error, a negative value.

The *flags* can be set to **BPF_F_GET_BRANCH_RECORDS_SIZE** to instead return the number of bytes required to store all the branch entries. If this flag is set, *buf* may be NULL.

−EINVAL if arguments invalid or **size** not a multiple of **sizeof**(**struct perf_branch_entry**).

−ENOENT if architecture does not support branch records.

long bpf_get_ns_current_pid_tgid(u64 _dev_, u64 _ino_, struct bpf_pidns_info *_nsdata_, u32 _size_)

Description

Returns 0 on success, values for *pid* and *tgid* as seen from the current *namespace* will be returned in *nsdata*.

Return 0 on success, or one of the following in case of failure:

−EINVAL if dev and inum supplied don't match dev_t and inode number with nsfs of current task, or if dev conversion to dev_t lost high bits.

−ENOENT if pidns does not exists for the current task.

long bpf_xdp_output(void *_ctx_, struct bpf_map *_map_, u64 _flags_, void *_data_, u64 _size_)

Description

Write raw *data* blob into a special BPF perf event held by *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. This perf event must have the following attributes: **PERF_SAMPLE_RAW** as **sample_type**, **PERF_TYPE_SOFTWARE** as **type**, and **PERF_COUNT_SW_BPF_OUTPUT** as **config**.

The *flags* are used to indicate the index in *map* for which the value must be put, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the index of the current CPU core should be used.

The value to write, of *size*, is passed through eBPF stack and pointed by *data*.

ctx is a pointer to in−kernel struct xdp_buff.

This helper is similar to **bpf_perf_eventoutput**() but restricted to raw_tracepoint bpf programs.

Return 0 on success, or a negative error in case of failure.

u64 bpf_get_netns_cookie(void *_ctx_)

Description

Retrieve the cookie (generated by the kernel) of the network namespace the input *ctx* is associated with. The network namespace cookie remains stable for its lifetime and provides a global identifier that can be assumed unique. If *ctx* is NULL, then the

helper returns the cookie for the initial network namespace. The cookie itself is very similar to that of **bpf_get_socket_cookie**() helper, but for network namespaces instead of sockets.

Return A 8−byte long opaque number.

u64 bpf_get_current_ancestor_cgroup_id(int *ancestor_level***)**

Description

Return id of cgroup v2 that is ancestor of the cgroup associated with the current task at the *ancestor_level*. The root cgroup is at *ancestor_level* zero and each step down the hierarchy increments the level. If *ancestor_level* == level of cgroup associated with the current task, then return value will be the same as that of **bpf_get_current_cgroup_id**().

The helper is useful to implement policies based on cgroups that are upper in hierarchy than immediate cgroup associated with the current task.

The format of returned id and helper limitations are same as in **bpf_get_current_cgroup_id**().

Return The id is returned or 0 in case the id could not be retrieved.

long bpf_sk_assign(struct sk_buff ***skb*, void ***sk***, u64** *flags***)**

Description

Helper is overloaded depending on BPF program type. This description applies to **BPF_PROG_TYPE_SCHED_CLS** and **BPF_PROG_TYPE_SCHED_ACT** programs.

Assign the *sk* to the *skb*. When combined with appropriate routing configuration to receive the packet towards the socket, will cause *skb* to be delivered to the specified socket. Subsequent redirection of *skb* via **bpf_redirect**(), **bpf_clone_redirect**() or other methods outside of BPF may interfere with successful delivery to the socket.

This operation is only valid from TC ingress path.

The *flags* argument must be zero.

Return 0 on success, or a negative error in case of failure:

−**EINVAL** if specified *flags* are not supported.

−**ENOENT** if the socket is unavailable for assignment.

−**ENETUNREACH** if the socket is unreachable (wrong netns).

−**EOPNOTSUPP** if the operation is not supported, for example a call from outside of TC ingress.

long bpf_sk_assign(struct bpf_sk_lookup ***ctx*, struct bpf_sock ***sk***, u64** *flags***)**

Description

Helper is overloaded depending on BPF program type. This description applies to **BPF_PROG_TYPE_SK_LOOKUP** programs.

Select the *sk* as a result of a socket lookup.

For the operation to succeed passed socket must be compatible with the packet description provided by the *ctx* object.

L4 protocol (**IPPROTO_TCP** or **IPPROTO_UDP**) must be an exact match. While IP family (**AF_INET** or **AF_INET6**) must be compatible, that is IPv6 sockets that are not v6−only can be selected for IPv4 packets.

Only TCP listeners and UDP unconnected sockets can be selected. *sk* can also be NULL to reset any previous selection.

flags argument can combination of following values:

- **BPF_SK_LOOKUP_F_REPLACE** to override the previous socket selection, potentially done by a BPF program that ran before us.
- **BPF_SK_LOOKUP_F_NO_REUSEPORT** to skip load−balancing within reuseport group for the socket being selected.

On success *ctx−>sk* will point to the selected socket.

Return 0 on success, or a negative errno in case of failure.

- **−EAFNOSUPPORT** if socket family (*sk−>family*) is not compatible with packet family (*ctx−>family*).
- **−EEXIST** if socket has been already selected, potentially by another program, and **BPF_SK_LOOKUP_F_REPLACE** flag was not specified.
- **−EINVAL** if unsupported flags were specified.
- **−EPROTOTYPE** if socket L4 protocol (*sk−>protocol*) doesn't match packet protocol (*ctx−>protocol*).
- **−ESOCKTNOSUPPORT** if socket is not in allowed state (TCP listening or UDP unconnected).

u64 bpf_ktime_get_boot_ns(void)

Description

Return the time elapsed since system boot, in nanoseconds. Does include the time the system was suspended. See: **clock_gettime**(**CLOCK_BOOTTIME**)

Return Current *ktime*.

long bpf_seq_printf(struct seq_file *_m_, const char *_fmt_, u32 _fmt_size_, const void *_data_, u32 _data_len_)

Description

bpf_seq_printf() uses seq_file **seq_printf**() to print out the format string. The *m* represents the seq_file. The *fmt* and *fmt_size* are for the format string itself. The *data* and *data_len* are format string arguments. The *data* are a **u64** array and corresponding format string values are stored in the array. For strings and pointers where pointees are accessed, only the pointer values are stored in the *data* array. The *data_len* is the size of *data* in bytes − must be a multiple of 8.

Formats **%s**, **%p{i,I}{4,6}** requires to read kernel memory. Reading kernel memory may fail due to either invalid address or valid address but requiring a major memory fault. If reading kernel memory fails, the string for **%s** will be an empty string, and the ip address for **%p{i,I}{4,6}** will be 0. Not returning error to bpf program is consistent with what **bpf_trace_printk**() does for now.

Return 0 on success, or a negative error in case of failure:

−EBUSY if per−CPU memory copy buffer is busy, can try again by returning 1 from bpf program.

−EINVAL if arguments are invalid, or if *fmt* is invalid/unsupported.

−E2BIG if *fmt* contains too many format specifiers.

−EOVERFLOW if an overflow happened: The same object will be tried again.

long bpf_seq_write(struct seq_file *_m_, const void *_data_, u32 _len_)

Description

bpf_seq_write() uses seq_file **seq_write**() to write the data. The *m* represents the seq_file. The *data* and *len* represent the data to write in bytes.

Return 0 on success, or a negative error in case of failure:

−**EOVERFLOW** if an overflow happened: The same object will be tried again.

u64 bpf_sk_cgroup_id(void ***sk*)**

Description

Return the cgroup v2 id of the socket *sk*.

sk must be a non−**NULL** pointer to a socket, e.g. one returned from **bpf_sk_lookup_xxx**(), **bpf_sk_fullsock**(), etc. The format of returned id is same as in **bpf_skb_cgroup_id**().

This helper is available only if the kernel was compiled with the **CONFIG_SOCK_CGROUP_DATA** configuration option.

Return The id is returned or 0 in case the id could not be retrieved.

u64 bpf_sk_ancestor_cgroup_id(void ***sk*, int** *ancestor_level***)**

Description

Return id of cgroup v2 that is ancestor of cgroup associated with the *sk* at the *ancestor_level*. The root cgroup is at *ancestor_level* zero and each step down the hierarchy increments the level. If *ancestor_level* == level of cgroup associated with *sk*, then return value will be same as that of **bpf_sk_cgroup_id**().

The helper is useful to implement policies based on cgroups that are upper in hierarchy than immediate cgroup associated with *sk*.

The format of returned id and helper limitations are same as in **bpf_sk_cgroup_id**().

Return The id is returned or 0 in case the id could not be retrieved.

long bpf_ringbuf_output(void ***ringbuf*, void ***data***, u64** *size***, u64** *flags***)**

Description

Copy *size* bytes from *data* into a ring buffer *ringbuf*. If **BPF_RB_NO_WAKEUP** is specified in *flags*, no notification of new data availability is sent. If **BPF_RB_FORCE_WAKEUP** is specified in *flags*, notification of new data availability is sent unconditionally. If **0** is specified in *flags*, an adaptive notification of new data availability is sent.

An adaptive notification is a notification sent whenever the user−space process has caught up and consumed all available payloads. In case the user−space process is still processing a previous payload, then no notification is needed as it will process the newly added payload automatically.

Return 0 on success, or a negative error in case of failure.

void *bpf_ringbuf_reserve(void ***ringbuf*, u64** *size***, u64** *flags***)**

Description

Reserve *size* bytes of payload in a ring buffer *ringbuf*. *flags* must be 0.

Return Valid pointer with *size* bytes of memory available; NULL, otherwise.

void bpf_ringbuf_submit(void ***data*, u64** *flags***)**

Description

Submit reserved ring buffer sample, pointed to by *data*. If **BPF_RB_NO_WAKEUP** is specified in *flags*, no notification of new data availability is sent. If **BPF_RB_FORCE_WAKEUP** is specified in *flags*, notification of new data availability is sent unconditionally. If **0** is specified in *flags*, an adaptive notification of new data availability is sent.

See 'bpf_ringbuf_output()' for the definition of adaptive notification.

Return Nothing. Always succeeds.

void bpf_ringbuf_discard(void ****data*, u64** *flags***)**

Description

Discard reserved ring buffer sample, pointed to by *data*. If **BPF_RB_NO_WAKEUP** is specified in *flags*, no notification of new data availability is sent. If **BPF_RB_FORCE_WAKEUP** is specified in *flags*, notification of new data availability is sent unconditionally. If **0** is specified in *flags*, an adaptive notification of new data availability is sent.

See 'bpf_ringbuf_output()' for the definition of adaptive notification.

Return Nothing. Always succeeds.

u64 bpf_ringbuf_query(void ****ringbuf*, u64** *flags***)**

Description

Query various characteristics of provided ring buffer. What exactly is queries is determined by *flags*:

- **BPF_RB_AVAIL_DATA**: Amount of data not yet consumed.
- **BPF_RB_RING_SIZE**: The size of ring buffer.
- **BPF_RB_CONS_POS**: Consumer position (can wrap around).
- **BPF_RB_PROD_POS**: Producer(s) position (can wrap around).

Data returned is just a momentary snapshot of actual values and could be inaccurate, so this facility should be used to power heuristics and for reporting, not to make 100% correct calculation.

Return Requested value, or 0, if *flags* are not recognized.

long bpf_csum_level(struct sk_buff ****skb*, u64** *level***)**

Description

Change the skbs checksum level by one layer up or down, or reset it entirely to none in order to have the stack perform checksum validation. The level is applicable to the following protocols: TCP, UDP, GRE, SCTP, FCOE. For example, a decap of | ETH | IP | UDP | GUE | IP | TCP | into | ETH | IP | TCP | through **bpf_skb_adjust_room**() helper with passing in **BPF_F_ADJ_ROOM_NO_CSUM_RESET** flag would require one call to **bpf_csum_level**() with **BPF_CSUM_LEVEL_DEC** since the UDP header is removed. Similarly, an encap of the latter into the former could be accompanied by a helper call to **bpf_csum_level**() with **BPF_CSUM_LEVEL_INC** if the skb is still intended to be processed in higher layers of the stack instead of just egressing at tc.

There are three supported level settings at this time:

- **BPF_CSUM_LEVEL_INC**: Increases skb−>csum_level for skbs with CHECKSUM_UNNECESSARY.
- **BPF_CSUM_LEVEL_DEC**: Decreases skb−>csum_level for skbs with CHECKSUM_UNNECESSARY.
- **BPF_CSUM_LEVEL_RESET**: Resets skb−>csum_level to 0 and sets CHECKSUM_NONE to force checksum validation by the stack.
- **BPF_CSUM_LEVEL_QUERY**: No−op, returns the current skb−>csum_level.

Return 0 on success, or a negative error in case of failure. In the case of **BPF_CSUM_LEVEL_QUERY**, the current skb−>csum_level is returned or the error code −EACCES in case the skb is not subject to CHECKSUM_UNNECESSARY.

struct tcp6_sock *bpf_skc_to_tcp6_sock(void ***sk*)**

Description
Dynamically cast a *sk* pointer to a *tcp6_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

struct tcp_sock *bpf_skc_to_tcp_sock(void ***sk*)**

Description
Dynamically cast a *sk* pointer to a *tcp_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

struct tcp_timewait_sock *bpf_skc_to_tcp_timewait_sock(void ***sk*)**

Description
Dynamically cast a *sk* pointer to a *tcp_timewait_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

struct tcp_request_sock *bpf_skc_to_tcp_request_sock(void ***sk*)**

Description
Dynamically cast a *sk* pointer to a *tcp_request_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

struct udp6_sock *bpf_skc_to_udp6_sock(void ***sk*)**

Description
Dynamically cast a *sk* pointer to a *udp6_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

long bpf_get_task_stack(struct task_struct ***task*, void ***buf***, u32** *size***, u64** *flags***)**

Description
Return a user or a kernel stack in bpf program provided buffer. Note: the user stack will only be populated if the *task* is the current task; all other tasks will return −EOPNOTSUPP. To achieve this, the helper needs *task*, which is a valid pointer to **struct task_struct**. To store the stacktrace, the bpf program provides *buf* with a nonnegative *size*.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set the following flags:

BPF_F_USER_STACK
Collect a user space stack instead of a kernel stack. The *task* must be the current task.

BPF_F_USER_BUILD_ID
Collect buildid+offset instead of ips for user stack, only valid if **BPF_F_USER_STACK** is also specified.

bpf_get_task_stack() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames, subject to sufficient large buffer size. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

```
# sysctl kernel.perf_event_max_stack=<new value>
```

Return The non−negative copied *buf* length equal to or less than *size* on success, or a negative error in case of failure.

long bpf_load_hdr_opt(struct bpf_sock_ops ***skops*, void ***searchby_res***, u32** *len***, u64** *flags***)**

Description
Load header option. Support reading a particular TCP header option for bpf program (**BPF_PROG_TYPE_SOCK_OPS**).

If *flags* is 0, it will search the option from the *skops*−>**skb_data**. The comment in **struct bpf_sock_ops** has details on what skb_data contains under different *skops*−>**op**.

The first byte of the *searchby_res* specifies the kind that it wants to search.

If the searching kind is an experimental kind (i.e. 253 or 254 according to RFC6994). It also needs to specify the "magic" which is either 2 bytes or 4 bytes. It then also needs to specify the size of the magic by using the 2nd byte which is "kind−length" of a TCP header option and the "kind−length" also includes the first 2 bytes "kind" and "kind−length" itself as a normal TCP header option also does.

For example, to search experimental kind 254 with 2 byte magic 0xeB9F, the searchby_res should be [254, 4, 0xeB, 0x9F, 0, 0, ... 0].

To search for the standard window scale option (3), the *searchby_res* should be [3, 0, 0, ... 0]. Note, kind−length must be 0 for regular option.

Searching for No−Op (0) and End−of−Option−List (1) are not supported.

len must be at least 2 bytes which is the minimal size of a header option.

Supported flags:

- **BPF_LOAD_HDR_OPT_TCP_SYN** to search from the saved_syn packet or the just−received syn packet.

Return > 0 when found, the header option is copied to *searchby_res*. The return value is the total length copied. On failure, a negative error code is returned:

−**EINVAL** if a parameter is invalid.

−**ENOMSG** if the option is not found.

−**ENOENT** if no syn packet is available when **BPF_LOAD_HDR_OPT_TCP_SYN** is used.

−**ENOSPC** if there is not enough space. Only *len* number of bytes are copied.

−**EFAULT** on failure to parse the header options in the packet.

−**EPERM** if the helper cannot be used under the current *skops*−>**op**.

long bpf_store_hdr_opt(struct bpf_sock_ops ***skops*, const void ***from***, u32** *len***, u64** *flags***)**

Description

Store header option. The data will be copied from buffer *from* with length *len* to the TCP header.

The buffer *from* should have the whole option that includes the kind, kind−length, and the actual option data. The *len* must be at least kind−length long. The kind−length does not have to be 4 byte aligned. The kernel will take care of the padding and setting the 4 bytes aligned value to th−>doff.

This helper will check for duplicated option by searching the same option in the outgoing skb.

This helper can only be called during **BPF_SOCK_OPS_WRITE_HDR_OPT_CB**.

Return 0 on success, or negative error in case of failure:

−**EINVAL** If param is invalid.

−**ENOSPC** if there is not enough space in the header. Nothing has been written

−**EEXIST** if the option already exists.

−**EFAULT** on failure to parse the existing header options.

−**EPERM** if the helper cannot be used under the current *skops*−>**op**.

long bpf_reserve_hdr_opt(struct bpf_sock_ops ***skops*, u32** *len***, u64** *flags***)**

Description

Reserve *len* bytes for the bpf header option. The space will be used by **bpf_store_hdr_opt**() later in **BPF_SOCK_OPS_WRITE_HDR_OPT_CB**.

If **bpf_reserve_hdr_opt**() is called multiple times, the total number of bytes will be reserved.

This helper can only be called during **BPF_SOCK_OPS_HDR_OPT_LEN_CB**.

Return 0 on success, or negative error in case of failure:

−**EINVAL** if a parameter is invalid.

−**ENOSPC** if there is not enough space in the header.

−**EPERM** if the helper cannot be used under the current *skops*−>**op**.

void *bpf_inode_storage_get(struct bpf_map ***map*, void *** *inode***, void *** *value***, u64** *flags***)**

Description

Get a bpf_local_storage from an *inode*.

Logically, it could be thought of as getting the value from a *map* with *inode* as the **key**. From this perspective, the usage is not much different from **bpf_map_lookup_elem**(*map*, **&***inode*) except this helper enforces the key must be an inode and the map must also be a **BPF_MAP_TYPE_INODE_STORAGE**.

Underneath, the value is stored locally at *inode* instead of the *map*. The *map* is used as the bpf−local−storage "type". The bpf−local−storage "type" (i.e. the *map*) is searched against all bpf_local_storage residing at *inode*.

An optional *flags* (**BPF_LOCAL_STORAGE_GET_F_CREATE**) can be used such that a new bpf_local_storage will be created if one does not exist. *value* can be used together with **BPF_LOCAL_STORAGE_GET_F_CREATE** to specify the initial value of a bpf_local_storage. If *value* is **NULL**, the new bpf_local_storage will be zero initialized.

Return A bpf_local_storage pointer is returned on success.

NULL if not found or there was an error in adding a new bpf_local_storage.

int bpf_inode_storage_delete(struct bpf_map ***map*, void *** *inode***)**

Description

Delete a bpf_local_storage from an *inode*.

Return 0 on success.

−**ENOENT** if the bpf_local_storage cannot be found.

long bpf_d_path(struct path ***path*, char ***buf***, u32** *sz***)**

Description

Return full path for given **struct path** object, which needs to be the kernel BTF *path* object. The path is returned in the provided buffer *buf* of size *sz* and is zero

terminated.

Return On success, the strictly positive length of the string, including the trailing NUL character. On error, a negative value.

long bpf_copy_from_user(void *_dst_, u32 _size_, const void *_user_ptr_)

Description

Read *size* bytes from user space address *user_ptr* and store the data in *dst*. This is a wrapper of **copy_from_user**().

Return 0 on success, or a negative error in case of failure.

long bpf_snprintf_btf(char *_str_, u32 _str_size_, struct btf_ptr *_ptr_, u32 _btf_ptr_size_, u64 _flags_)

Description

Use BTF to store a string representation of *ptr*->ptr in *str*, using *ptr*->type_id. This value should specify the type that *ptr*->ptr points to. LLVM __builtin_btf_type_id(type, 1) can be used to look up vmlinux BTF type ids. Traversing the data structure using BTF, the type information and values are stored in the first *str_size* - 1 bytes of *str*. Safe copy of the pointer data is carried out to avoid kernel crashes during operation. Smaller types can use string space on the stack; larger programs can use map data to store the string representation.

The string can be subsequently shared with userspace via bpf_perf_event_output() or ring buffer interfaces. bpf_trace_printk() is to be avoided as it places too small a limit on string size to be useful.

flags is a combination of

BTF_F_COMPACT
no formatting around type information

BTF_F_NONAME
no struct/union member names/types

BTF_F_PTR_RAW
show raw (unobfuscated) pointer values; equivalent to printk specifier %px.

BTF_F_ZERO
show zero-valued struct/union members; they are not displayed by default

Return The number of bytes that were written (or would have been written if output had to be truncated due to string size), or a negative error in cases of failure.

long bpf_seq_printf_btf(struct seq_file *_m_, struct btf_ptr *_ptr_, u32 _ptr_size_, u64 _flags_)

Description

Use BTF to write to seq_write a string representation of *ptr*->ptr, using *ptr*->type_id as per bpf_snprintf_btf(). *flags* are identical to those used for bpf_snprintf_btf.

Return 0 on success or a negative error in case of failure.

u64 bpf_skb_cgroup_classid(struct sk_buff *_skb_)

Description

See **bpf_get_cgroup_classid**() for the main description. This helper differs from **bpf_get_cgroup_classid**() in that the cgroup v1 net_cls class is retrieved only from the *skb*'s associated socket instead of the current process.

Return The id is returned or 0 in case the id could not be retrieved.

long bpf_redirect_neigh(u32 _ifindex_, struct bpf_redir_neigh *_params_, int _plen_, u64 _flags_)

Description

Redirect the packet to another net device of index *ifindex* and fill in L2 addresses from neighboring subsystem. This helper is somewhat similar to **bpf_redirect**(), except that it populates L2 addresses as well, meaning, internally, the helper relies on the neighbor lookup for the L2 address of the nexthop.

The helper will perform a FIB lookup based on the skb's networking header to get the address of the next hop, unless this is supplied by the caller in the *params* argument. The *plen* argument indicates the len of *params* and should be set to 0 if *params* is NULL.

The *flags* argument is reserved and must be 0. The helper is currently only supported for tc BPF program types, and enabled for IPv4 and IPv6 protocols.

Return The helper returns **TC_ACT_REDIRECT** on success or **TC_ACT_SHOT** on error.

void *bpf_per_cpu_ptr(const void ****percpu_ptr*, u32** *cpu***)**

Description

Take a pointer to a percpu ksym, *percpu_ptr*, and return a pointer to the percpu kernel variable on *cpu*. A ksym is an extern variable decorated with '__ksym'. For ksym, there is a global var (either static or global) defined of the same name in the kernel. The ksym is percpu if the global var is percpu. The returned pointer points to the global percpu var on *cpu*.

bpf_per_cpu_ptr() has the same semantic as per_cpu_ptr() in the kernel, except that bpf_per_cpu_ptr() may return NULL. This happens if *cpu* is larger than nr_cpu_ids. The caller of bpf_per_cpu_ptr() must check the returned value.

Return A pointer pointing to the kernel percpu variable on *cpu*, or NULL, if *cpu* is invalid.

void *bpf_this_cpu_ptr(const void ****percpu_ptr*)**

Description

Take a pointer to a percpu ksym, *percpu_ptr*, and return a pointer to the percpu kernel variable on this cpu. See the description of 'ksym' in **bpf_per_cpu_ptr**().

bpf_this_cpu_ptr() has the same semantic as this_cpu_ptr() in the kernel. Different from **bpf_per_cpu_ptr**(), it would never return NULL.

Return A pointer pointing to the kernel percpu variable on this cpu.

long bpf_redirect_peer(u32 *ifindex***, u64** *flags***)**

Description

Redirect the packet to another net device of index *ifindex*. This helper is somewhat similar to **bpf_redirect**(), except that the redirection happens to the *ifindex*' peer device and the netns switch takes place from ingress to ingress without going through the CPU's backlog queue.

The *flags* argument is reserved and must be 0. The helper is currently only supported for tc BPF program types at the ingress hook and for veth device types. The peer device must reside in a different network namespace.

Return The helper returns **TC_ACT_REDIRECT** on success or **TC_ACT_SHOT** on error.

void *bpf_task_storage_get(struct bpf_map ****map*, struct task_struct ****task***, void ****value***, u64** *flags***)**

Description

Get a bpf_local_storage from the *task*.

Logically, it could be thought of as getting the value from a *map* with *task* as the **key**. From this perspective, the usage is not much different from **bpf_map_lookup_elem**(*map*, **&***task*) except this helper enforces the key must be a task_struct and the map must also be a **BPF_MAP_TYPE_TASK_STORAGE**.

Underneath, the value is stored locally at *task* instead of the *map*. The *map* is used as the bpf−local−storage "type". The bpf−local−storage "type" (i.e. the *map*) is searched against all bpf_local_storage residing at *task*.

An optional *flags* (**BPF_LOCAL_STORAGE_GET_F_CREATE**) can be used such that a new bpf_local_storage will be created if one does not exist. *value* can be used

together with **BPF_LOCAL_STORAGE_GET_F_CREATE** to specify the initial value of a bpf_local_storage. If *value* is **NULL**, the new bpf_local_storage will be zero initialized.

Return A bpf_local_storage pointer is returned on success.

NULL if not found or there was an error in adding a new bpf_local_storage.

long bpf_task_storage_delete(struct bpf_map ****map*, struct task_struct ****task***)**

Description

Delete a bpf_local_storage from a *task*.

Return 0 on success.

−ENOENT if the bpf_local_storage cannot be found.

struct task_struct *bpf_get_current_task_btf(void)

Description

Return a BTF pointer to the "current" task. This pointer can also be used in helpers that accept an *ARG_PTR_TO_BTF_ID* of type *task_struct*.

Return Pointer to the current task.

long bpf_bprm_opts_set(struct linux_binprm ****bprm*, u64** *flags***)**

Description

Set or clear certain options on *bprm*:

BPF_F_BPRM_SECUREEXEC Set the secureexec bit which sets the **AT_SECURE** auxv for glibc. The bit is cleared if the flag is not specified.

Return **−EINVAL** if invalid *flags* are passed, zero otherwise.

u64 bpf_ktime_get_coarse_ns(void)

Description

Return a coarse−grained version of the time elapsed since system boot, in nanoseconds. Does not include time the system was suspended.

See: **clock_gettime**(**CLOCK_MONOTONIC_COARSE**)

Return Current *ktime*.

long bpf_ima_inode_hash(struct inode ****inode*, void ****dst***, u32** *size***)**

Description

Returns the stored IMA hash of the *inode* (if it's available). If the hash is larger than *size*, then only *size* bytes will be copied to *dst*

Return The **hash_algo** is returned on success, **−EOPNOTSUP** if IMA is disabled or **−EINVAL** if invalid arguments are passed.

struct socket *bpf_sock_from_file(struct file ****file*)**

Description

If the given file represents a socket, returns the associated socket.

Return A pointer to a struct socket on success or NULL if the file is not a socket.

long bpf_check_mtu(void ****ctx*, u32** *ifindex***, u32 ****mtu_len***, s32** *len_diff***, u64** *flags***)**

Description

Check packet size against exceeding MTU of net device (based on *ifindex*). This helper will likely be used in combination with helpers that adjust/change the packet size.

The argument *len_diff* can be used for querying with a planned size change. This allows to check MTU prior to changing packet ctx. Providing a *len_diff* adjustment that is larger than the actual packet size (resulting in negative packet size) will in principle

not exceed the MTU, which is why it is not considered a failure. Other BPF helpers are needed for performing the planned size change; therefore the responsibility for catching a negative packet size belongs in those helpers.

Specifying *ifindex* zero means the MTU check is performed against the current net device. This is practical if this isn't used prior to redirect.

On input *mtu_len* must be a valid pointer, else verifier will reject BPF program. If the value *mtu_len* is initialized to zero then the ctx packet size is use. When value *mtu_len* is provided as input this specify the L3 length that the MTU check is done against. Remember XDP and TC length operate at L2, but this value is L3 as this correlate to MTU and IP−header tot_len values which are L3 (similar behavior as bpf_fib_lookup).

The Linux kernel route table can configure MTUs on a more specific per route level, which is not provided by this helper. For route level MTU checks use the **bpf_fib_lookup**() helper.

ctx is either **struct xdp_md** for XDP programs or **struct sk_buff** for tc cls_act programs.

The *flags* argument can be a combination of one or more of the following values:

BPF_MTU_CHK_SEGS
: This flag will only works for *ctx* **struct sk_buff**. If packet context contains extra packet segment buffers (often knows as GSO skb), then MTU check is harder to check at this point, because in transmit path it is possible for the skb packet to get re−segmented (depending on net device features). This could still be a MTU violation, so this flag enables performing MTU check against segments, with a different violation return code to tell it apart. Check cannot use len_diff.

On return *mtu_len* pointer contains the MTU value of the net device. Remember the net device configured MTU is the L3 size, which is returned here and XDP and TC length operate at L2. Helper take this into account for you, but remember when using MTU value in your BPF−code.

Return

- 0 on success, and populate MTU value in *mtu_len* pointer.
- < 0 if any input argument is invalid (*mtu_len* not updated)

MTU violations return positive values, but also populate MTU value in *mtu_len* pointer, as this can be needed for implementing PMTU handing:

- **BPF_MTU_CHK_RET_FRAG_NEEDED**
- **BPF_MTU_CHK_RET_SEGS_TOOBIG**

long bpf_for_each_map_elem(struct bpf_map ****map*, void ****callback_fn***, void ****callback_ctx***, u64** ***flags*****)**

Description

For each element in **map**, call **callback_fn** function with **map**, **callback_ctx** and other map−specific parameters. The **callback_fn** should be a static function and the **callback_ctx** should be a pointer to the stack. The **flags** is used to control certain aspects of the helper. Currently, the **flags** must be 0.

The following are a list of supported map types and their respective expected callback signatures:

BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_PERCPU_HASH,

BPF_MAP_TYPE_LRU_HASH, BPF_MAP_TYPE_LRU_PERCPU_HASH, BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_PERCPU_ARRAY

long (*callback_fn)(struct bpf_map *map, const void *key, void *value, void *ctx);

For per_cpu maps, the map_value is the value on the cpu where the bpf_prog is running.

If **callback_fn** return 0, the helper will continue to the next element. If return value is 1, the helper will skip the rest of elements and return. Other return values are not used now.

Return The number of traversed map elements for success, **−EINVAL** for invalid **flags**.

long bpf_snprintf(char *_str_, u32 _str_size_, const char *_fmt_, u64 *_data_, u32 _data_len_)

Description

Outputs a string into the **str** buffer of size **str_size** based on a format string stored in a read−only map pointed by **fmt**.

Each format specifier in **fmt** corresponds to one u64 element in the **data** array. For strings and pointers where pointees are accessed, only the pointer values are stored in the *data* array. The *data_len* is the size of *data* in bytes − must be a multiple of 8.

Formats **%s** and **%p{i,I}{4,6}** require to read kernel memory. Reading kernel memory may fail due to either invalid address or valid address but requiring a major memory fault. If reading kernel memory fails, the string for **%s** will be an empty string, and the ip address for **%p{i,I}{4,6}** will be 0. Not returning error to bpf program is consistent with what **bpf_trace_printk**() does for now.

Return The strictly positive length of the formatted string, including the trailing zero character. If the return value is greater than **str_size**, **str** contains a truncated string, guaranteed to be zero−terminated except when **str_size** is 0.

Or **−EBUSY** if the per−CPU memory copy buffer is busy.

long bpf_sys_bpf(u32 _cmd_, void *_attr_, u32 _attr_size_)

Description

Execute bpf syscall with given arguments.

Return A syscall result.

long bpf_btf_find_by_name_kind(char *_name_, int _name_sz_, u32 _kind_, int _flags_)

Description

Find BTF type with given name and kind in vmlinux BTF or in module's BTFs.

Return Returns btf_id and btf_obj_fd in lower and upper 32 bits.

long bpf_sys_close(u32 _fd_)

Description

Execute close syscall for given FD.

Return A syscall result.

long bpf_timer_init(struct bpf_timer *_timer_, struct bpf_map *_map_, u64 _flags_)

Description

Initialize the timer. First 4 bits of *flags* specify clockid. Only CLOCK_MONOTONIC, CLOCK_REALTIME, CLOCK_BOOTTIME are allowed. All other bits of *flags* are reserved. The verifier will reject the program if *timer* is not from the same *map*.

Return 0 on success. **−EBUSY** if *timer* is already initialized. **−EINVAL** if invalid *flags* are passed. **−EPERM** if *timer* is in a map that doesn't have any user references. The user space should either hold a file descriptor to a map with timers or pin such map in

bpffs. When map is unpinned or file descriptor is closed all timers in the map will be cancelled and freed.

long bpf_timer_set_callback(struct bpf_timer ****timer*, void ****callback_fn***)**

Description

Configure the timer to call *callback_fn* static function.

Return 0 on success. **−EINVAL** if *timer* was not initialized with bpf_timer_init() earlier. **−EPERM** if *timer* is in a map that doesn't have any user references. The user space should either hold a file descriptor to a map with timers or pin such map in bpffs. When map is unpinned or file descriptor is closed all timers in the map will be cancelled and freed.

long bpf_timer_start(struct bpf_timer ****timer*, u64** *nsecs***, u64** *flags***)**

Description

Set timer expiration N nanoseconds from the current time. The configured callback will be invoked in soft irq context on some cpu and will not repeat unless another bpf_timer_start() is made. In such case the next invocation can migrate to a different cpu. Since struct bpf_timer is a field inside map element the map owns the timer. The bpf_timer_set_callback() will increment refcnt of BPF program to make sure that callback_fn code stays valid. When user space reference to a map reaches zero all timers in a map are cancelled and corresponding program's refcnts are decremented. This is done to make sure that Ctrl−C of a user process doesn't leave any timers running. If map is pinned in bpffs the callback_fn can re−arm itself indefinitely. bpf_map_update/delete_elem() helpers and user space sys_bpf commands cancel and free the timer in the given map element. The map can contain timers that invoke callback_fn−s from different programs. The same callback_fn can serve different timers from different maps if key/value layout matches across maps. Every bpf_timer_set_callback() can have different callback_fn.

flags can be one of:

BPF_F_TIMER_ABS
Start the timer in absolute expire value instead of the default relative one.

BPF_F_TIMER_CPU_PIN
Timer will be pinned to the CPU of the caller.

Return 0 on success. **−EINVAL** if *timer* was not initialized with bpf_timer_init() earlier or invalid *flags* are passed.

long bpf_timer_cancel(struct bpf_timer ****timer*)**

Description

Cancel the timer and wait for callback_fn to finish if it was running.

Return 0 if the timer was not active. 1 if the timer was active. **−EINVAL** if *timer* was not initialized with bpf_timer_init() earlier. **−EDEADLK** if callback_fn tried to call bpf_timer_cancel() on its own timer which would have led to a deadlock otherwise.

u64 bpf_get_func_ip(void ****ctx*)**

Description

Get address of the traced function (for tracing and kprobe programs).

When called for kprobe program attached as uprobe it returns probe address for both entry and return uprobe.

Return Address of the traced function for kprobe. 0 for kprobes placed within the function (not at the entry). Address of the probe for uprobe and return uprobe.

u64 bpf_get_attach_cookie(void ****ctx*)**

Description

Get bpf_cookie value provided (optionally) during the program attachment. It might be different for each individual attachment, even if BPF program itself is the same.

Expects BPF program context *ctx* as a first argument.

Supported for the following program types:

- kprobe/uprobe;
- tracepoint;
- perf_event.

Return Value specified by user at BPF link creation/attachment time or 0, if it was not specified.

long bpf_task_pt_regs(struct task_struct ****task*)**

Description

Get the struct pt_regs associated with **task**.

Return A pointer to struct pt_regs.

long bpf_get_branch_snapshot(void ****entries*, u32** *size***, u64** *flags***)**

Description

Get branch trace from hardware engines like Intel LBR. The hardware engine is stopped shortly after the helper is called. Therefore, the user need to filter branch entries based on the actual use case. To capture branch trace before the trigger point of the BPF program, the helper should be called at the beginning of the BPF program.

The data is stored as struct perf_branch_entry into output buffer *entries*. *size* is the size of *entries* in bytes. *flags* is reserved for now and must be zero.

Return On success, number of bytes written to *buf*. On error, a negative value.

−**EINVAL** if *flags* is not zero.

−**ENOENT** if architecture does not support branch records.

long bpf_trace_vprintk(const char ****fmt*, u32** *fmt_size***, const void ****data***, u32** *data_len***)**

Description

Behaves like **bpf_trace_printk**() helper, but takes an array of u64 to format and can handle more format args as a result.

Arguments are to be used as in **bpf_seq_printf**() helper.

Return The number of bytes written to the buffer, or a negative error in case of failure.

struct unix_sock *bpf_skc_to_unix_sock(void ****sk*)**

Description

Dynamically cast a *sk* pointer to a *unix_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

long bpf_kallsyms_lookup_name(const char ****name*, int** *name_sz***, int** *flags***, u64 ****res***)**

Description

Get the address of a kernel symbol, returned in *res*. *res* is set to 0 if the symbol is not found.

Return On success, zero. On error, a negative value.

−**EINVAL** if *flags* is not zero.

−**EINVAL** if string *name* is not the same size as *name_sz*.

−**ENOENT** if symbol is not found.

−**EPERM** if caller does not have permission to obtain kernel address.

long bpf_find_vma(struct task_struct ***task*, u64** *addr***, void (***callback_fn***)(struct task_struct ***task***, struct vm_area_struct ***vma***, void ***callback_ctx***), void ***callback_ctx***, u64** *flags***)**

Description

Find vma of *task* that contains *addr*, call *callback_fn* function with *task*, *vma*, and *callback_ctx*. The *callback_fn* should be a static function and the *callback_ctx* should be a pointer to the stack. The *flags* is used to control certain aspects of the helper. Currently, the *flags* must be 0.

The expected callback signature is

long (*callback_fn)(struct task_struct *task, struct vm_area_struct *vma, void *callback_ctx);

Return 0 on success. **−ENOENT** if *task−>mm* is NULL, or no vma contains *addr*. **−EBUSY** if failed to try lock mmap_lock. **−EINVAL** for invalid **flags**.

long bpf_loop(u32 *nr_loops***, void (***callback_fn***)(u32, void ***callback_ctx***), void ***callback_ctx***, u64** *flags***)**

Description

For **nr_loops**, call **callback_fn** function with **callback_ctx** as the context parameter. The **callback_fn** should be a static function and the **callback_ctx** should be a pointer to the stack. The **flags** is used to control certain aspects of the helper. Currently, the **flags** must be 0. Currently, nr_loops is limited to 1 << 23 (˜8 million) loops.

long (*callback_fn)(u32 index, void *ctx);

where **index** is the current index in the loop. The index is zero−indexed.

If **callback_fn** returns 0, the helper will continue to the next loop. If return value is 1, the helper will skip the rest of the loops and return. Other return values are not used now, and will be rejected by the verifier.

Return The number of loops performed, **−EINVAL** for invalid **flags**, **−E2BIG** if **nr_loops** exceeds the maximum number of loops.

long bpf_strncmp(const char ***s1*, u32** *s1_sz***, const char ***s2***)**

Description

Do strncmp() between **s1** and **s2**. **s1** doesn't need to be null−terminated and **s1_sz** is the maximum storage size of **s1**. **s2** must be a read−only string.

Return An integer less than, equal to, or greater than zero if the first **s1_sz** bytes of **s1** is found to be less than, to match, or be greater than **s2**.

long bpf_get_func_arg(void ***ctx*, u32** *n***, u64 ***value***)**

Description

Get **n**−th argument register (zero based) of the traced function (for tracing programs) returned in **value**.

Return 0 on success. **−EINVAL** if n >= argument register count of traced function.

long bpf_get_func_ret(void ***ctx*, u64 ***value***)**

Description

Get return value of the traced function (for tracing programs) in **value**.

Return 0 on success. **−EOPNOTSUPP** for tracing programs other than BPF_TRACE_FEXIT or BPF_MODIFY_RETURN.

long bpf_get_func_arg_cnt(void ***ctx*)**

Description

Get number of registers of the traced function (for tracing programs) where function arguments are stored in these registers.

Return The number of argument registers of the traced function.

int bpf_get_retval(void)

Description

Get the BPF program's return value that will be returned to the upper layers.

This helper is currently supported by cgroup programs and only by the hooks where BPF program's return value is returned to the userspace via errno.

Return The BPF program's return value.

int bpf_set_retval(int *retval***)**

Description

Set the BPF program's return value that will be returned to the upper layers.

This helper is currently supported by cgroup programs and only by the hooks where BPF program's return value is returned to the userspace via errno.

Note that there is the following corner case where the program exports an error via bpf_set_retval but signals success via 'return 1':

bpf_set_retval(−EPERM); return 1;

In this case, the BPF program's return value will use helper's −EPERM. This still holds true for cgroup/bind{4,6} which supports extra 'return 3' success case.

Return 0 on success, or a negative error in case of failure.

u64 bpf_xdp_get_buff_len(struct xdp_buff ***xdp_md*)**

Description

Get the total size of a given xdp buff (linear and paged area)

Return The total size of a given xdp buffer.

long bpf_xdp_load_bytes(struct xdp_buff ***xdp_md*, u32** *offset***, void ***buf***, u32** *len***)**

Description

This helper is provided as an easy way to load data from a xdp buffer. It can be used to load *len* bytes from *offset* from the frame associated to *xdp_md*, into the buffer pointed by *buf*.

Return 0 on success, or a negative error in case of failure.

long bpf_xdp_store_bytes(struct xdp_buff ***xdp_md*, u32** *offset***, void ***buf***, u32** *len***)**

Description

Store *len* bytes from buffer *buf* into the frame associated to *xdp_md*, at *offset*.

Return 0 on success, or a negative error in case of failure.

long bpf_copy_from_user_task(void ***dst*, u32** *size***, const void ***user_ptr***, struct task_struct ***tsk***, u64** *flags***)**

Description

Read *size* bytes from user space address *user_ptr* in *tsk*'s address space, and stores the data in *dst*. *flags* is not used yet and is provided for future extensibility. This helper can only be used by sleepable programs.

Return 0 on success, or a negative error in case of failure. On error *dst* buffer is zeroed out.

long bpf_skb_set_tstamp(struct sk_buff ***skb*, u64** *tstamp***, u32** *tstamp_type***)**

Description

Change the __sk_buff−>tstamp_type to *tstamp_type* and set *tstamp* to the __sk_buff−>tstamp together.

If there is no need to change the __sk_buff−>tstamp_type, the tstamp value can be directly written to __sk_buff−>tstamp instead.

BPF_SKB_TSTAMP_DELIVERY_MONO is the only tstamp that will be kept during bpf_redirect_*(). A non zero *tstamp* must be used with the BPF_SKB_TSTAMP_DELIVERY_MONO *tstamp_type*.

A BPF_SKB_TSTAMP_UNSPEC *tstamp_type* can only be used with a zero *tstamp*.

Only IPv4 and IPv6 skb−>protocol are supported.

This function is most useful when it needs to set a mono delivery time to __sk_buff−>tstamp and then bpf_redirect_*() to the egress of an iface. For example, changing the (rcv) timestamp in __sk_buff−>tstamp at ingress to a mono delivery time and then bpf_redirect_*() to *sch_fq@phy−dev*.

Return 0 on success. **−EINVAL** for invalid input **−EOPNOTSUPP** for unsupported protocol

long bpf_ima_file_hash(struct file *_file_, void *_dst_, u32 _size_)

Description

Returns a calculated IMA hash of the *file*. If the hash is larger than *size*, then only *size* bytes will be copied to *dst*

Return The **hash_algo** is returned on success, **−EOPNOTSUP** if the hash calculation failed or **−EINVAL** if invalid arguments are passed.

void *bpf_kptr_xchg(void *_map_value_, void *_ptr_)

Description

Exchange kptr at pointer *map_value* with *ptr*, and return the old value. *ptr* can be NULL, otherwise it must be a referenced pointer which will be released when this helper is called.

Return The old value of kptr (which can be NULL). The returned pointer if not NULL, is a reference which must be released using its corresponding release function, or moved into a BPF map before program exit.

void *bpf_map_lookup_percpu_elem(struct bpf_map *_map_, const void *_key_, u32 _cpu_)

Description

Perform a lookup in *percpu map* for an entry associated to *key* on *cpu*.

Return Map value associated to *key* on *cpu*, or **NULL** if no entry was found or *cpu* is invalid.

struct mptcp_sock *bpf_skc_to_mptcp_sock(void *_sk_)

Description

Dynamically cast a *sk* pointer to a *mptcp_sock* pointer.

Return *sk* if casting is valid, or **NULL** otherwise.

long bpf_dynptr_from_mem(void *_data_, u32 _size_, u64 _flags_, struct bpf_dynptr *_ptr_)

Description

Get a dynptr to local memory *data*.

data must be a ptr to a map value. The maximum *size* supported is DYNPTR_MAX_SIZE. *flags* is currently unused.

Return 0 on success, −E2BIG if the size exceeds DYNPTR_MAX_SIZE, −EINVAL if flags is not 0.

long bpf_ringbuf_reserve_dynptr(void *_ringbuf_, u32 _size_, u64 _flags_, struct bpf_dynptr *_ptr_)

Description

Reserve *size* bytes of payload in a ring buffer *ringbuf* through the dynptr interface. *flags* must be 0.

Please note that a corresponding bpf_ringbuf_submit_dynptr or bpf_ringbuf_discard_dynptr must be called on *ptr*, even if the reservation fails. This is enforced by

the verifier.

Return 0 on success, or a negative error in case of failure.

void bpf_ringbuf_submit_dynptr(struct bpf_dynptr *_ptr_, u64 _flags_)

Description

Submit reserved ring buffer sample, pointed to by *data*, through the dynptr interface. This is a no−op if the dynptr is invalid/null.

For more information on *flags*, please see 'bpf_ringbuf_submit'.

Return Nothing. Always succeeds.

void bpf_ringbuf_discard_dynptr(struct bpf_dynptr *_ptr_, u64 _flags_)

Description

Discard reserved ring buffer sample through the dynptr interface. This is a no−op if the dynptr is invalid/null.

For more information on *flags*, please see 'bpf_ringbuf_discard'.

Return Nothing. Always succeeds.

long bpf_dynptr_read(void *_dst_, u32 _len_, const struct bpf_dynptr *_src_, u32 _offset_, u64 _flags_)

Description

Read *len* bytes from *src* into *dst*, starting from *offset* into *src*. *flags* is currently unused.

Return 0 on success, −E2BIG if *offset* + *len* exceeds the length of *src*'s data, −EINVAL if *src* is an invalid dynptr or if *flags* is not 0.

long bpf_dynptr_write(const struct bpf_dynptr *_dst_, u32 _offset_, void *_src_, u32 _len_, u64 _flags_)

Description

Write *len* bytes from *src* into *dst*, starting from *offset* into *dst*.

flags must be 0 except for skb−type dynptrs.

For skb−type dynptrs:

- All data slices of the dynptr are automatically invalidated after **bpf_dynptr_write**(). This is because writing may pull the skb and change the underlying packet buffer.
- For *flags*, please see the flags accepted by **bpf_skb_store_bytes**().

Return 0 on success, −E2BIG if *offset* + *len* exceeds the length of *dst*'s data, −EINVAL if *dst* is an invalid dynptr or if *dst* is a read−only dynptr or if *flags* is not correct. For skb−type dynptrs, other errors correspond to errors returned by **bpf_skb_store_bytes**().

void *bpf_dynptr_data(const struct bpf_dynptr *_ptr_, u32 _offset_, u32 _len_)

Description

Get a pointer to the underlying dynptr data.

len must be a statically known value. The returned data slice is invalidated whenever the dynptr is invalidated.

skb and xdp type dynptrs may not use bpf_dynptr_data. They should instead use bpf_dynptr_slice and bpf_dynptr_slice_rdwr.

Return Pointer to the underlying dynptr data, NULL if the dynptr is read−only, if the dynptr is invalid, or if the offset and length is out of bounds.

s64 bpf_tcp_raw_gen_syncookie_ipv4(struct iphdr *_iph_, struct tcphdr *_th_, u32 _th_len_)

Description

Try to issue a SYN cookie for the packet with corresponding IPv4/TCP headers, *iph* and *th*, without depending on a listening socket.

iph points to the IPv4 header.

th points to the start of the TCP header, while *th_len* contains the length of the TCP header (at least **sizeof**(**struct tcphdr**)).

Return On success, lower 32 bits hold the generated SYN cookie in followed by 16 bits which hold the MSS value for that cookie, and the top 16 bits are unused.

On failure, the returned value is one of the following:

−EINVAL if *th_len* is invalid.

s64 bpf_tcp_raw_gen_syncookie_ipv6(struct ipv6hdr ***_iph_, struct tcphdr ***_th_**, u32** _th_len**)**

Description

Try to issue a SYN cookie for the packet with corresponding IPv6/TCP headers, *iph* and *th*, without depending on a listening socket.

iph points to the IPv6 header.

th points to the start of the TCP header, while *th_len* contains the length of the TCP header (at least **sizeof**(**struct tcphdr**)).

Return On success, lower 32 bits hold the generated SYN cookie in followed by 16 bits which hold the MSS value for that cookie, and the top 16 bits are unused.

On failure, the returned value is one of the following:

−EINVAL if *th_len* is invalid.

−EPROTONOSUPPORT if CONFIG_IPV6 is not builtin.

long bpf_tcp_raw_check_syncookie_ipv4(struct iphdr ***_iph_, struct tcphdr ***_th_**)**

Description

Check whether *iph* and *th* contain a valid SYN cookie ACK without depending on a listening socket.

iph points to the IPv4 header.

th points to the TCP header.

Return 0 if *iph* and *th* are a valid SYN cookie ACK.

On failure, the returned value is one of the following:

−EACCES if the SYN cookie is not valid.

long bpf_tcp_raw_check_syncookie_ipv6(struct ipv6hdr ***_iph_, struct tcphdr ***_th_**)**

Description

Check whether *iph* and *th* contain a valid SYN cookie ACK without depending on a listening socket.

iph points to the IPv6 header.

th points to the TCP header.

Return 0 if *iph* and *th* are a valid SYN cookie ACK.

On failure, the returned value is one of the following:

−EACCES if the SYN cookie is not valid.

−EPROTONOSUPPORT if CONFIG_IPV6 is not builtin.

u64 bpf_ktime_get_tai_ns(void)

Description

A nonsettable system−wide clock derived from wall−clock time but ignoring leap seconds. This clock does not experience discontinuities and backwards jumps caused by NTP inserting leap seconds as CLOCK_REALTIME does.

See: **clock_gettime**(**CLOCK_TAI**)

Return Current *ktime*.

long bpf_user_ringbuf_drain(struct bpf_map *_map_, void *_callback_fn_, void *_ctx_, u64 _flags_)

Description

Drain samples from the specified user ring buffer, and invoke the provided callback for each such sample:

long (*callback_fn)(const struct bpf_dynptr *dynptr, void *ctx);

If **callback_fn** returns 0, the helper will continue to try and drain the next sample, up to a maximum of BPF_MAX_USER_RINGBUF_SAMPLES samples. If the return value is 1, the helper will skip the rest of the samples and return. Other return values are not used now, and will be rejected by the verifier.

Return The number of drained samples if no error was encountered while draining samples, or 0 if no samples were present in the ring buffer. If a user−space producer was epoll−waiting on this map, and at least one sample was drained, they will receive an event notification notifying them of available space in the ring buffer. If the BPF_RB_NO_WAKEUP flag is passed to this function, no wakeup notification will be sent. If the BPF_RB_FORCE_WAKEUP flag is passed, a wakeup notification will be sent even if no sample was drained.

On failure, the returned value is one of the following:

−EBUSY if the ring buffer is contended, and another calling context was concurrently draining the ring buffer.

−EINVAL if user−space is not properly tracking the ring buffer due to the producer position not being aligned to 8 bytes, a sample not being aligned to 8 bytes, or the producer position not matching the advertised length of a sample.

−E2BIG if user−space has tried to publish a sample which is larger than the size of the ring buffer, or which cannot fit within a struct bpf_dynptr.

void *bpf_cgrp_storage_get(struct bpf_map *_map_, struct cgroup *_cgroup_, void *_value_, u64 _flags_)

Description

Get a bpf_local_storage from the *cgroup*.

Logically, it could be thought of as getting the value from a *map* with *cgroup* as the **key**. From this perspective, the usage is not much different from **bpf_map_lookup_elem**(*map*, **&***cgroup*) except this helper enforces the key must be a cgroup struct and the map must also be a **BPF_MAP_TYPE_CGRP_STORAGE**.

In reality, the local−storage value is embedded directly inside of the *cgroup* object itself, rather than being located in the **BPF_MAP_TYPE_CGRP_STORAGE** map. When the local−storage value is queried for some *map* on a *cgroup* object, the kernel will perform an O(n) iteration over all of the live local−storage values for that *cgroup* object until the local−storage value for the *map* is found.

An optional *flags* (**BPF_LOCAL_STORAGE_GET_F_CREATE**) can be used such that a new bpf_local_storage will be created if one does not exist. *value* can be used together with **BPF_LOCAL_STORAGE_GET_F_CREATE** to specify the initial value of a bpf_local_storage. If *value* is **NULL**, the new bpf_local_storage will be zero initialized.

Return A bpf_local_storage pointer is returned on success.

NULL if not found or there was an error in adding a new bpf_local_storage.

long bpf_cgrp_storage_delete(struct bpf_map ****map*, struct cgroup ****cgroup***)**

Description

Delete a bpf_local_storage from a *cgroup*.

Return 0 on success.

−ENOENT if the bpf_local_storage cannot be found.

EXAMPLES

Example usage for most of the eBPF helpers listed in this manual page are available within the Linux kernel sources, at the following locations:

- *samples/bpf/*
- *tools/testing/selftests/bpf/*

LICENSE

eBPF programs can have an associated license, passed along with the bytecode instructions to the kernel when the programs are loaded. The format for that string is identical to the one in use for kernel modules (Dual licenses, such as "Dual BSD/GPL", may be used). Some helper functions are only accessible to programs that are compatible with the GNU General Public License (GNU GPL).

In order to use such helpers, the eBPF program must be loaded with the correct license string passed (via **attr**) to the **bpf**() system call, and this generally translates into the C source code of the program containing a line similar to the following:

```
char ____license[] __attribute__((section("license"), used)) = "GPL";
```

IMPLEMENTATION

This manual page is an effort to document the existing eBPF helper functions. But as of this writing, the BPF sub−system is under heavy development. New eBPF program or map types are added, along with new helper functions. Some helpers are occasionally made available for additional program types. So in spite of the efforts of the community, this page might not be up−to−date. If you want to check by yourself what helper functions exist in your kernel, or what types of programs they can support, here are some files among the kernel tree that you may be interested in:

- *include/uapi/linux/bpf.h* is the main BPF header. It contains the full list of all helper functions, as well as many other BPF definitions including most of the flags, structs or constants used by the helpers.
- *net/core/filter.c* contains the definition of most network−related helper functions, and the list of program types from which they can be used.
- *kernel/trace/bpf_trace.c* is the equivalent for most tracing program−related helpers.
- *kernel/bpf/verifier.c* contains the functions used to check that valid types of eBPF maps are used with a given helper function.
- *kernel/bpf/* directory contains other files in which additional helpers are defined (for cgroups, sockmaps, etc.).
- The bpftool utility can be used to probe the availability of helper functions on the system (as well as supported program and map types, and a number of other parameters). To do so, run **bpftool feature probe** (see **bpftool−feature**(8) for details). Add the **unprivileged** keyword to list features available to unprivileged users.

Compatibility between helper functions and program types can generally be found in the files where

helper functions are defined. Look for the **struct bpf_func_proto** objects and for functions returning them: these functions contain a list of helpers that a given program type can call. Note that the **default:** label of the **switch ... case** used to filter helpers can call other functions, themselves allowing access to additional helpers. The requirement for GPL license is also in those **struct bpf_func_proto**.

Compatibility between helper functions and map types can be found in the **check_map_func_compatibility**() function in file *kernel/bpf/verifier.c*.

Helper functions that invalidate the checks on **data** and **data_end** pointers for network processing are listed in function **bpf_helper_changes_pkt_data**() in file *net/core/filter.c*.

SEE ALSO

bpf(2), **bpftool**(8), **cgroups**(7), **ip**(8), **perf_event_open**(2), **sendmsg**(2), **socket**(7), **tc-bpf**(8)

NAME

capabilities − overview of Linux capabilities

DESCRIPTION

For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: *privileged* processes (whose effective user ID is 0, referred to as superuser or root), and *unprivileged* processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with Linux 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as *capabilities*, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

Capabilities list

The following list shows the capabilities implemented on Linux, and the operations or behaviors that each capability permits:

CAP_AUDIT_CONTROL (since Linux 2.6.11)
Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.

CAP_AUDIT_READ (since Linux 3.16)
Allow reading the audit log via a multicast netlink socket.

CAP_AUDIT_WRITE (since Linux 2.6.11)
Write records to kernel auditing log.

CAP_BLOCK_SUSPEND (since Linux 3.5)
Employ features that can block system suspend (**epoll**(7) **EPOLLWAKEUP**, */proc/sys/wake_lock*).

CAP_BPF (since Linux 5.8)
Employ privileged BPF operations; see *bpf*(2) and *bpf-helpers*(7).

This capability was added in Linux 5.8 to separate out BPF functionality from the overloaded **CAP_SYS_ADMIN** capability.

CAP_CHECKPOINT_RESTORE (since Linux 5.9)
- Update */proc/sys/kernel/ns_last_pid* (see *pid_namespaces*(7));
- employ the *set_tid* feature of *clone3*(2);
- read the contents of the symbolic links in */proc/*pid*/map_files* for other processes.

This capability was added in Linux 5.9 to separate out checkpoint/restore functionality from the overloaded **CAP_SYS_ADMIN** capability.

CAP_CHOWN
Make arbitrary changes to file UIDs and GIDs (see *chown*(2)).

CAP_DAC_OVERRIDE
Bypass file read, write, and execute permission checks. (DAC is an abbreviation of "discretionary access control".)

CAP_DAC_READ_SEARCH
- Bypass file read permission checks and directory read and execute permission checks;
- invoke *open_by_handle_at*(2);
- use the *linkat*(2) **AT_EMPTY_PATH** flag to create a link to a file referred to by a file descriptor.

CAP_FOWNER
- Bypass permission checks on operations that normally require the filesystem UID of the process to match the UID of the file (e.g., *chmod*(2), *utime*(2)), excluding those operations covered by **CAP_DAC_OVERRIDE** and **CAP_DAC_READ_SEARCH**;
- set inode flags (see *ioctl_iflags*(2)) on arbitrary files;
- set Access Control Lists (ACLs) on arbitrary files;
- ignore directory sticky bit on file deletion;

- modify *user* extended attributes on sticky directory owned by any user;
- specify **O_NOATIME** for arbitrary files in *open*(2) and *fcntl*(2).

CAP_FSETID
- Don't clear set-user-ID and set-group-ID mode bits when a file is modified;
- set the set-group-ID bit for a file whose GID does not match the filesystem or any of the supplementary GIDs of the calling process.

CAP_IPC_LOCK
- Lock memory (**mlock**(2), *mlockall*(2), *mmap*(2), *shmctl*(2));
- Allocate memory using huge pages (**memfd_create**(2), *mmap*(2), *shmctl*(2)).

CAP_IPC_OWNER
Bypass permission checks for operations on System V IPC objects.

CAP_KILL
Bypass permission checks for sending signals (see *kill*(2)). This includes use of the *ioctl*(2) **KDSIGACCEPT** operation.

CAP_LEASE (since Linux 2.4)
Establish leases on arbitrary files (see *fcntl*(2)).

CAP_LINUX_IMMUTABLE
Set the **FS_APPEND_FL** and **FS_IMMUTABLE_FL** inode flags (see *ioctl_iflags*(2)).

CAP_MAC_ADMIN (since Linux 2.6.25)
Allow MAC configuration or state changes. Implemented for the Smack Linux Security Module (LSM).

CAP_MAC_OVERRIDE (since Linux 2.6.25)
Override Mandatory Access Control (MAC). Implemented for the Smack LSM.

CAP_MKNOD (since Linux 2.4)
Create special files using *mknod*(2).

CAP_NET_ADMIN
Perform various network-related operations:
- interface configuration;
- administration of IP firewall, masquerading, and accounting;
- modify routing tables;
- bind to any address for transparent proxying;
- set type-of-service (TOS);
- clear driver statistics;
- set promiscuous mode;
- enabling multicasting;
- use *setsockopt*(2) to set the following socket options: **SO_DEBUG**, **SO_MARK**, **SO_PRIORITY** (for a priority outside the range 0 to 6), **SO_RCVBUFFORCE**, and **SO_SNDBUFFORCE**.

CAP_NET_BIND_SERVICE
Bind a socket to Internet domain privileged ports (port numbers less than 1024).

CAP_NET_BROADCAST
(Unused) Make socket broadcasts, and listen to multicasts.

CAP_NET_RAW
- Use RAW and PACKET sockets;
- bind to any address for transparent proxying.

CAP_PERFMON (since Linux 5.8)
Employ various performance-monitoring mechanisms, including:

- call *perf_event_open*(2);
- employ various BPF operations that have performance implications.

This capability was added in Linux 5.8 to separate out performance monitoring functionality from the overloaded **CAP_SYS_ADMIN** capability. See also the kernel source file *Documentation/admin−guide/perf−security.rst*.

CAP_SETGID
- Make arbitrary manipulations of process GIDs and supplementary GID list;
- forge GID when passing socket credentials via UNIX domain sockets;
- write a group ID mapping in a user namespace (see *user_namespaces*(7)).

CAP_SETFCAP (since Linux 2.6.24)
Set arbitrary capabilities on a file.

Since Linux 5.12, this capability is also needed to map user ID 0 in a new user namespace; see *user_namespaces*(7) for details.

CAP_SETPCAP
If file capabilities are supported (i.e., since Linux 2.6.24): add any capability from the calling thread's bounding set to its inheritable set; drop capabilities from the bounding set (via *prctl*(2) **PR_CAPBSET_DROP**); make changes to the *securebits* flags.

If file capabilities are not supported (i.e., before Linux 2.6.24): grant or remove any capability in the caller's permitted capability set to or from any other process. (This property of **CAP_SETPCAP** is not available when the kernel is configured to support file capabilities, since **CAP_SETPCAP** has entirely different semantics for such kernels.)

CAP_SETUID
- Make arbitrary manipulations of process UIDs (**setuid**(2), *setreuid*(2), *setresuid*(2), *setfsuid*(2));
- forge UID when passing socket credentials via UNIX domain sockets;
- write a user ID mapping in a user namespace (see *user_namespaces*(7)).

CAP_SYS_ADMIN
Note: this capability is overloaded; see *Notes to kernel developers* below.

- Perform a range of system administration operations including: *quotactl*(2), *mount*(2), *umount*(2), *pivot_root*(2), *swapon*(2), *swapoff*(2), *sethostname*(2), and *setdomainname*(2);
- perform privileged *syslog*(2) operations (since Linux 2.6.37, **CAP_SYSLOG** should be used to permit such operations);
- perform **VM86_REQUEST_IRQ** *vm86*(2) command;
- access the same checkpoint/restore functionality that is governed by **CAP_CHECKPOINT_RESTORE** (but the latter, weaker capability is preferred for accessing that functionality).
- perform the same BPF operations as are governed by **CAP_BPF** (but the latter, weaker capability is preferred for accessing that functionality).
- employ the same performance monitoring mechanisms as are governed by **CAP_PERFMON** (but the latter, weaker capability is preferred for accessing that functionality).
- perform **IPC_SET** and **IPC_RMID** operations on arbitrary System V IPC objects;
- override **RLIMIT_NPROC** resource limit;
- perform operations on *trusted* and *security* extended attributes (see *xattr*(7));
- use *lookup_dcookie*(2);
- use *ioprio_set*(2) to assign **IOPRIO_CLASS_RT** and (before Linux 2.6.25) **IOPRIO_CLASS_IDLE** I/O scheduling classes;
- forge PID when passing socket credentials via UNIX domain sockets;
- exceed */proc/sys/fs/file−max*, the system-wide limit on the number of open files, in system calls that open files (e.g., *accept*(2), *execve*(2), *open*(2), *pipe*(2));
- employ **CLONE_*** flags that create new namespaces with *clone*(2) and *unshare*(2) (but, since Linux 3.8, creating user namespaces does not require any capability);
- access privileged *perf* event information;
- call *setns*(2) (requires **CAP_SYS_ADMIN** in the *target* namespace);
- call *fanotify_init*(2);
- perform privileged **KEYCTL_CHOWN** and **KEYCTL_SETPERM** *keyctl*(2) operations;
- perform *madvise*(2) **MADV_HWPOISON** operation;
- employ the **TIOCSTI** *ioctl*(2) to insert characters into the input queue of a terminal other than the caller's controlling terminal;
- employ the obsolete *nfsservctl*(2) system call;

- employ the obsolete *bdflush*(2) system call;
- perform various privileged block-device *ioctl*(2) operations;
- perform various privileged filesystem *ioctl*(2) operations;
- perform privileged *ioctl*(2) operations on the */dev/random* device (see *random*(4));
- install a *seccomp*(2) filter without first having to set the *no_new_privs* thread attribute;
- modify allow/deny rules for device control groups;
- employ the *ptrace*(2) **PTRACE_SECCOMP_GET_FILTER** operation to dump tracee's seccomp filters;
- employ the *ptrace*(2) **PTRACE_SETOPTIONS** operation to suspend the tracee's seccomp protections (i.e., the **PTRACE_O_SUSPEND_SECCOMP** flag);
- perform administrative operations on many device drivers;
- modify autogroup nice values by writing to */proc/*pid*/autogroup* (see *sched*(7)).

CAP_SYS_BOOT

Use *reboot*(2) and *kexec_load*(2).

CAP_SYS_CHROOT

- Use *chroot*(2);
- change mount namespaces using *setns*(2).

CAP_SYS_MODULE

- Load and unload kernel modules (see *init_module*(2) and *delete_module*(2));
- before Linux 2.6.25: drop capabilities from the system-wide capability bounding set.

CAP_SYS_NICE

- Lower the process nice value (**nice**(2), *setpriority*(2)) and change the nice value for arbitrary processes;
- set real-time scheduling policies for calling process, and set scheduling policies and priorities for arbitrary processes (**sched_setscheduler**(2), *sched_setparam*(2), *sched_setattr*(2));
- set CPU affinity for arbitrary processes (**sched_setaffinity**(2));
- set I/O scheduling class and priority for arbitrary processes (**ioprio_set**(2));
- apply *migrate_pages*(2) to arbitrary processes and allow processes to be migrated to arbitrary nodes;
- apply *move_pages*(2) to arbitrary processes;
- use the **MPOL_MF_MOVE_ALL** flag with *mbind*(2) and *move_pages*(2).

CAP_SYS_PACCT

Use *acct*(2).

CAP_SYS_PTRACE

- Trace arbitrary processes using *ptrace*(2);
- apply *get_robust_list*(2) to arbitrary processes;
- transfer data to or from the memory of arbitrary processes using *process_vm_readv*(2) and *process_vm_writev*(2);
- inspect processes using *kcmp*(2).

CAP_SYS_RAWIO

- Perform I/O port operations (**iopl**(2) and *ioperm*(2));
- access */proc/kcore*;
- employ the **FIBMAP** *ioctl*(2) operation;
- open devices for accessing x86 model-specific registers (MSRs, see *msr*(4));
- update */proc/sys/vm/mmap_min_addr*;
- create memory mappings at addresses below the value specified by */proc/sys/vm/mmap_min_addr*;
- map files in */proc/bus/pci*;
- open */dev/mem* and */dev/kmem*;
- perform various SCSI device commands;
- perform certain operations on *hpsa*(4) and *cciss*(4) devices;
- perform a range of device-specific operations on other devices.

CAP_SYS_RESOURCE

- Use reserved space on ext2 filesystems;
- make *ioctl*(2) calls controlling ext3 journaling;
- override disk quota limits;
- increase resource limits (see *setrlimit*(2));
- override **RLIMIT_NPROC** resource limit;
- override maximum number of consoles on console allocation;
- override maximum number of keymaps;
- allow more than 64hz interrupts from the real-time clock;
- raise *msg_qbytes* limit for a System V message queue above the limit in */proc/sys/kernel/msgmnb* (see *msgop*(2) and *msgctl*(2));
- allow the **RLIMIT_NOFILE** resource limit on the number of "in-flight" file descriptors to be bypassed when passing file descriptors to another process via a UNIX domain socket (see *unix*(7));
- override the */proc/sys/fs/pipe-size-max* limit when setting the capacity of a pipe using the **F_SETPIPE_SZ** *fcntl*(2) command;
- use **F_SETPIPE_SZ** to increase the capacity of a pipe above the limit specified by */proc/sys/fs/pipe-max-size*;
- override */proc/sys/fs/mqueue/queues_max*, */proc/sys/fs/mqueue/msg_max*, and */proc/sys/fs/mqueue/msgsize_max* limits when creating POSIX message queues (see *mq_overview*(7));
- employ the *prctl*(2) **PR_SET_MM** operation;
- set */proc/*pid*/oom_score_adj* to a value lower than the value last set by a process with **CAP_SYS_RESOURCE**.

CAP_SYS_TIME
Set system clock (**settimeofday**(2), *stime*(2), *adjtimex*(2)); set real-time (hardware) clock.

CAP_SYS_TTY_CONFIG
Use *vhangup*(2); employ various privileged *ioctl*(2) operations on virtual terminals.

CAP_SYSLOG (since Linux 2.6.37)
- Perform privileged *syslog*(2) operations. See *syslog*(2) for information on which operations require privilege.
- View kernel addresses exposed via */proc* and other interfaces when */proc/sys/kernel/kptr_restrict* has the value 1. (See the discussion of the *kptr_restrict* in *proc*(5).)

CAP_WAKE_ALARM (since Linux 3.0)
Trigger something that will wake up the system (set **CLOCK_REALTIME_ALARM** and **CLOCK_BOOTTIME_ALARM** timers).

Past and current implementation

A full implementation of capabilities requires that:

- For all privileged operations, the kernel must check whether the thread has the required capability in its effective set.
- The kernel must provide system calls allowing a thread's capability sets to be changed and retrieved.
- The filesystem must support attaching capabilities to an executable file, so that a process gains those capabilities when the file is executed.

Before Linux 2.6.24, only the first two of these requirements are met; since Linux 2.6.24, all three requirements are met.

Notes to kernel developers

When adding a new kernel feature that should be governed by a capability, consider the following points.

- The goal of capabilities is divide the power of superuser into pieces, such that if a program that has one or more capabilities is compromised, its power to do damage to the system would be less than the same program running with root privilege.
- You have the choice of either creating a new capability for your new feature, or associating the feature with one of the existing capabilities. In order to keep the set of capabilities to a manageable size, the latter option is preferable, unless there are compelling reasons to take the former option.

(There is also a technical limit: the size of capability sets is currently limited to 64 bits.)

- To determine which existing capability might best be associated with your new feature, review the list of capabilities above in order to find a "silo" into which your new feature best fits. One approach to take is to determine if there are other features requiring capabilities that will always be used along with the new feature. If the new feature is useless without these other features, you should use the same capability as the other features.
- *Don't* choose **CAP_SYS_ADMIN** if you can possibly avoid it! A vast proportion of existing capability checks are associated with this capability (see the partial list above). It can plausibly be called "the new root", since on the one hand, it confers a wide range of powers, and on the other hand, its broad scope means that this is the capability that is required by many privileged programs. Don't make the problem worse. The only new features that should be associated with **CAP_SYS_ADMIN** are ones that *closely* match existing uses in that silo.
- If you have determined that it really is necessary to create a new capability for your feature, don't make or name it as a "single-use" capability. Thus, for example, the addition of the highly specific **CAP_SYS_PACCT** was probably a mistake. Instead, try to identify and name your new capability as a broader silo into which other related future use cases might fit.

Thread capability sets

Each thread has the following capability sets containing zero or more of the above capabilities:

Permitted

This is a limiting superset for the effective capabilities that the thread may assume. It is also a limiting superset for the capabilities that may be added to the inheritable set by a thread that does not have the **CAP_SETPCAP** capability in its effective set.

If a thread drops a capability from its permitted set, it can never reacquire that capability (unless it *execve*(2)s either a set-user-ID-root program, or a program whose associated file capabilities grant that capability).

Inheritable

This is a set of capabilities preserved across an *execve*(2). Inheritable capabilities remain inheritable when executing any program, and inheritable capabilities are added to the permitted set when executing a program that has the corresponding bits set in the file inheritable set.

Because inheritable capabilities are not generally preserved across *execve*(2) when running as a non-root user, applications that wish to run helper programs with elevated capabilities should consider using ambient capabilities, described below.

Effective

This is the set of capabilities used by the kernel to perform permission checks for the thread.

Bounding (per-thread since Linux 2.6.25)

The capability bounding set is a mechanism that can be used to limit the capabilities that are gained during *execve*(2).

Since Linux 2.6.25, this is a per-thread capability set. In older kernels, the capability bounding set was a system wide attribute shared by all threads on the system.

For more details, see *Capability bounding set* below.

Ambient (since Linux 4.3)

This is a set of capabilities that are preserved across an *execve*(2) of a program that is not privileged. The ambient capability set obeys the invariant that no capability can ever be ambient if it is not both permitted and inheritable.

The ambient capability set can be directly modified using *prctl*(2). Ambient capabilities are automatically lowered if either of the corresponding permitted or inheritable capabilities is lowered.

Executing a program that changes UID or GID due to the set-user-ID or set-group-ID bits or executing a program that has any file capabilities set will clear the ambient set. Ambient capabilities are added to the permitted set and assigned to the effective set when *execve*(2) is called. If ambient capabilities cause a process's permitted and effective capabilities to increase during an *execve*(2), this does not trigger the secure-execution mode described in *ld.so*(8).

A child created via *fork*(2) inherits copies of its parent's capability sets. For details on how *execve*(2) affects capabilities, see *Transformation of capabilities during execve()* below.

Using *capset*(2), a thread may manipulate its own capability sets; see *Programmatically adjusting capability sets* below.

Since Linux 3.2, the file */proc/sys/kernel/cap_last_cap* exposes the numerical value of the highest capability supported by the running kernel; this can be used to determine the highest bit that may be set in a capability set.

File capabilities

Since Linux 2.6.24, the kernel supports associating capability sets with an executable file using *setcap*(8)The file capability sets are stored in an extended attribute (see *setxattr*(2) and *xattr*(7)) named *security.capability*. Writing to this extended attribute requires the **CAP_SETFCAP** capability. The file capability sets, in conjunction with the capability sets of the thread, determine the capabilities of a thread after an *execve*(2).

The three file capability sets are:

Permitted (formerly known as *forced*):
: These capabilities are automatically permitted to the thread, regardless of the thread's inheritable capabilities.

Inheritable (formerly known as *allowed*):
: This set is ANDed with the thread's inheritable set to determine which inheritable capabilities are enabled in the permitted set of the thread after the *execve*(2).

Effective:
: This is not a set, but rather just a single bit. If this bit is set, then during an *execve*(2) all of the new permitted capabilities for the thread are also raised in the effective set. If this bit is not set, then after an *execve*(2), none of the new permitted capabilities is in the new effective set.

  Enabling the file effective capability bit implies that any file permitted or inheritable capability that causes a thread to acquire the corresponding permitted capability during an *execve*(2) (see *Transformation of capabilities during execve()* below) will also acquire that capability in its effective set. Therefore, when assigning capabilities to a file (**setcap**(8), *cap_set_file*(3), *cap_set_fd*(3)), if we specify the effective flag as being enabled for any capability, then the effective flag must also be specified as enabled for all other capabilities for which the corresponding permitted or inheritable flag is enabled.

File capability extended attribute versioning

To allow extensibility, the kernel supports a scheme to encode a version number inside the *security.capability* extended attribute that is used to implement file capabilities. These version numbers are internal to the implementation, and not directly visible to user-space applications. To date, the following versions are supported:

VFS_CAP_REVISION_1
: This was the original file capability implementation, which supported 32-bit masks for file capabilities.

VFS_CAP_REVISION_2 (since Linux 2.6.25)
: This version allows for file capability masks that are 64 bits in size, and was necessary as the number of supported capabilities grew beyond 32. The kernel transparently continues to support the execution of files that have 32-bit version 1 capability masks, but when adding capabilities to files that did not previously have capabilities, or modifying the capabilities of existing files, it automatically uses the version 2 scheme (or possibly the version 3 scheme, as described below).

VFS_CAP_REVISION_3 (since Linux 4.14)
: Version 3 file capabilities are provided to support namespaced file capabilities (described below).

  As with version 2 file capabilities, version 3 capability masks are 64 bits in size. But in addition, the root user ID of namespace is encoded in the *security.capability* extended attribute. (A namespace's root user ID is the value that user ID 0 inside that namespace maps to in the initial user namespace.)

Version 3 file capabilities are designed to coexist with version 2 capabilities; that is, on a modern Linux system, there may be some files with version 2 capabilities while others have version 3 capabilities.

Before Linux 4.14, the only kind of file capability extended attribute that could be attached to a file was a **VFS_CAP_REVISION_2** attribute. Since Linux 4.14, the version of the *security.capability* extended attribute that is attached to a file depends on the circumstances in which the attribute was created.

Starting with Linux 4.14, a *security.capability* extended attribute is automatically created as (or converted to) a version 3 (**VFS_CAP_REVISION_3**) attribute if both of the following are true:

- The thread writing the attribute resides in a noninitial user namespace. (More precisely: the thread resides in a user namespace other than the one from which the underlying filesystem was mounted.)
- The thread has the **CAP_SETFCAP** capability over the file inode, meaning that (a) the thread has the **CAP_SETFCAP** capability in its own user namespace; and (b) the UID and GID of the file inode have mappings in the writer's user namespace.

When a **VFS_CAP_REVISION_3** *security.capability* extended attribute is created, the root user ID of the creating thread's user namespace is saved in the extended attribute.

By contrast, creating or modifying a *security.capability* extended attribute from a privileged (**CAP_SETFCAP**) thread that resides in the namespace where the underlying filesystem was mounted (this normally means the initial user namespace) automatically results in the creation of a version 2 (**VFS_CAP_REVISION_2**) attribute.

Note that the creation of a version 3 *security.capability* extended attribute is automatic. That is to say, when a user-space application writes (**setxattr**(2)) a *security.capability* attribute in the version 2 format, the kernel will automatically create a version 3 attribute if the attribute is created in the circumstances described above. Correspondingly, when a version 3 *security.capability* attribute is retrieved (**getxattr**(2)) by a process that resides inside a user namespace that was created by the root user ID (or a descendant of that user namespace), the returned attribute is (automatically) simplified to appear as a version 2 attribute (i.e., the returned value is the size of a version 2 attribute and does not include the root user ID). These automatic translations mean that no changes are required to user-space tools (e.g., *setcap*(1) and *getcap*(1)) in order for those tools to be used to create and retrieve version 3 *security.capability* attributes.

Note that a file can have either a version 2 or a version 3 *security.capability* extended attribute associated with it, but not both: creation or modification of the *security.capability* extended attribute will automatically modify the version according to the circumstances in which the extended attribute is created or modified.

Transformation of capabilities during execve()

During an *execve*(2), the kernel calculates the new capabilities of the process using the following algorithm:

```
P'(ambient)     = (file is privileged) ? 0 : P(ambient)

P'(permitted)   = (P(inheritable) & F(inheritable)) |
                  (F(permitted) & P(bounding)) | P'(ambient)

P'(effective)   = F(effective) ? P'(permitted) : P'(ambient)

P'(inheritable) = P(inheritable)    [i.e., unchanged]

P'(bounding)    = P(bounding)       [i.e., unchanged]
```

where:

P() denotes the value of a thread capability set before the *execve*(2)

P'() denotes the value of a thread capability set after the *execve*(2)

F() denotes a file capability set

Note the following details relating to the above capability transformation rules:

- The ambient capability set is present only since Linux 4.3. When determining the transformation of the ambient set during *execve*(2), a privileged file is one that has capabilities or has the set-user-ID or set-group-ID bit set.
- Prior to Linux 2.6.25, the bounding set was a system-wide attribute shared by all threads. That system-wide value was employed to calculate the new permitted set during *execve*(2) in the same manner as shown above for *P(bounding)*.

Note: during the capability transitions described above, file capabilities may be ignored (treated as empty) for the same reasons that the set-user-ID and set-group-ID bits are ignored; see *execve*(2). File capabilities are similarly ignored if the kernel was booted with the *no_file_caps* option.

Note: according to the rules above, if a process with nonzero user IDs performs an *execve*(2) then any capabilities that are present in its permitted and effective sets will be cleared. For the treatment of capabilities when a process with a user ID of zero performs an *execve*(2), see *Capabilities and execution of programs by root* below.

Safety checking for capability-dumb binaries

A capability-dumb binary is an application that has been marked to have file capabilities, but has not been converted to use the *libcap*(3) API to manipulate its capabilities. (In other words, this is a traditional set-user-ID-root program that has been switched to use file capabilities, but whose code has not been modified to understand capabilities.) For such applications, the effective capability bit is set on the file, so that the file permitted capabilities are automatically enabled in the process effective set when executing the file. The kernel recognizes a file which has the effective capability bit set as capability-dumb for the purpose of the check described here.

When executing a capability-dumb binary, the kernel checks if the process obtained all permitted capabilities that were specified in the file permitted set, after the capability transformations described above have been performed. (The typical reason why this might *not* occur is that the capability bounding set masked out some of the capabilities in the file permitted set.) If the process did not obtain the full set of file permitted capabilities, then *execve*(2) fails with the error **EPERM**. This prevents possible security risks that could arise when a capability-dumb application is executed with less privilege than it needs. Note that, by definition, the application could not itself recognize this problem, since it does not employ the *libcap*(3) API.

Capabilities and execution of programs by root

In order to mirror traditional UNIX semantics, the kernel performs special treatment of file capabilities when a process with UID 0 (root) executes a program and when a set-user-ID-root program is executed.

After having performed any changes to the process effective ID that were triggered by the set-user-ID mode bit of the binary—e.g., switching the effective user ID to 0 (root) because a set-user-ID-root program was executed—the kernel calculates the file capability sets as follows:

(1) If the real or effective user ID of the process is 0 (root), then the file inheritable and permitted sets are ignored; instead they are notionally considered to be all ones (i.e., all capabilities enabled). (There is one exception to this behavior, described in *Set-user-ID-root programs that have file capabilities* below.)

(2) If the effective user ID of the process is 0 (root) or the file effective bit is in fact enabled, then the file effective bit is notionally defined to be one (enabled).

These notional values for the file's capability sets are then used as described above to calculate the transformation of the process's capabilities during *execve*(2).

Thus, when a process with nonzero UIDs *execve*(2)s a set-user-ID-root program that does not have capabilities attached, or when a process whose real and effective UIDs are zero *execve*(2)s a program, the calculation of the process's new permitted capabilities simplifies to:

```
P'(permitted)   = P(inheritable) | P(bounding)

P'(effective)   = P'(permitted)
```

Consequently, the process gains all capabilities in its permitted and effective capability sets, except those masked out by the capability bounding set. (In the calculation of P'(permitted), the P'(ambient) term can be simplified away because it is by definition a proper subset of P(inheritable).)

The special treatments of user ID 0 (root) described in this subsection can be disabled using the

securebits mechanism described below.

Set-user-ID-root programs that have file capabilities

There is one exception to the behavior described in *Capabilities and execution of programs by root* above. If (a) the binary that is being executed has capabilities attached and (b) the real user ID of the process is *not* 0 (root) and (c) the effective user ID of the process *is* 0 (root), then the file capability bits are honored (i.e., they are not notionally considered to be all ones). The usual way in which this situation can arise is when executing a set-UID-root program that also has file capabilities. When such a program is executed, the process gains just the capabilities granted by the program (i.e., not all capabilities, as would occur when executing a set-user-ID-root program that does not have any associated file capabilities).

Note that one can assign empty capability sets to a program file, and thus it is possible to create a set-user-ID-root program that changes the effective and saved set-user-ID of the process that executes the program to 0, but confers no capabilities to that process.

Capability bounding set

The capability bounding set is a security mechanism that can be used to limit the capabilities that can be gained during an *execve*(2). The bounding set is used in the following ways:

- During an *execve*(2), the capability bounding set is ANDed with the file permitted capability set, and the result of this operation is assigned to the thread's permitted capability set. The capability bounding set thus places a limit on the permitted capabilities that may be granted by an executable file.
- (Since Linux 2.6.25) The capability bounding set acts as a limiting superset for the capabilities that a thread can add to its inheritable set using *capset*(2). This means that if a capability is not in the bounding set, then a thread can't add this capability to its inheritable set, even if it was in its permitted capabilities, and thereby cannot have this capability preserved in its permitted set when it *execve*(2)s a file that has the capability in its inheritable set.

Note that the bounding set masks the file permitted capabilities, but not the inheritable capabilities. If a thread maintains a capability in its inheritable set that is not in its bounding set, then it can still gain that capability in its permitted set by executing a file that has the capability in its inheritable set.

Depending on the kernel version, the capability bounding set is either a system-wide attribute, or a per-process attribute.

Capability bounding set from Linux 2.6.25 onward

From Linux 2.6.25, the *capability bounding set* is a per-thread attribute. (The system-wide capability bounding set described below no longer exists.)

The bounding set is inherited at *fork*(2) from the thread's parent, and is preserved across an *execve*(2).

A thread may remove capabilities from its capability bounding set using the *prctl*(2) **PR_CAPBSET_DROP** operation, provided it has the **CAP_SETPCAP** capability. Once a capability has been dropped from the bounding set, it cannot be restored to that set. A thread can determine if a capability is in its bounding set using the *prctl*(2) **PR_CAPBSET_READ** operation.

Removing capabilities from the bounding set is supported only if file capabilities are compiled into the kernel. Before Linux 2.6.33, file capabilities were an optional feature configurable via the **CONFIG_SECURITY_FILE_CAPABILITIES** option. Since Linux 2.6.33, the configuration option has been removed and file capabilities are always part of the kernel. When file capabilities are compiled into the kernel, the **init** process (the ancestor of all processes) begins with a full bounding set. If file capabilities are not compiled into the kernel, then **init** begins with a full bounding set minus **CAP_SETPCAP**, because this capability has a different meaning when there are no file capabilities.

Removing a capability from the bounding set does not remove it from the thread's inheritable set. However it does prevent the capability from being added back into the thread's inheritable set in the future.

Capability bounding set prior to Linux 2.6.25

Before Linux 2.6.25, the capability bounding set is a system-wide attribute that affects all threads on the system. The bounding set is accessible via the file */proc/sys/kernel/cap−bound*. (Confusingly, this bit mask parameter is expressed as a signed decimal number in */proc/sys/kernel/cap−bound*.)

Only the **init** process may set capabilities in the capability bounding set; other than that, the superuser (more precisely: a process with the **CAP_SYS_MODULE** capability) may only clear capabilities from this set.

On a standard system the capability bounding set always masks out the **CAP_SETPCAP** capability. To remove this restriction (dangerous!), modify the definition of **CAP_INIT_EFF_SET** in *include/linux/capability.h* and rebuild the kernel.

The system-wide capability bounding set feature was added to Linux 2.2.11.

Effect of user ID changes on capabilities

To preserve the traditional semantics for transitions between 0 and nonzero user IDs, the kernel makes the following changes to a thread's capability sets on changes to the thread's real, effective, saved set, and filesystem user IDs (using *setuid*(2), *setresuid*(2), or similar):

- If one or more of the real, effective, or saved set user IDs was previously 0, and as a result of the UID changes all of these IDs have a nonzero value, then all capabilities are cleared from the permitted, effective, and ambient capability sets.
- If the effective user ID is changed from 0 to nonzero, then all capabilities are cleared from the effective set.
- If the effective user ID is changed from nonzero to 0, then the permitted set is copied to the effective set.
- If the filesystem user ID is changed from 0 to nonzero (see *setfsuid*(2)), then the following capabilities are cleared from the effective set: **CAP_CHOWN**, **CAP_DAC_OVERRIDE**, **CAP_DAC_READ_SEARCH**, **CAP_FOWNER**, **CAP_FSETID**, **CAP_LINUX_IMMUTABLE** (since Linux 2.6.30), **CAP_MAC_OVERRIDE**, and **CAP_MKNOD** (since Linux 2.6.30). If the filesystem UID is changed from nonzero to 0, then any of these capabilities that are enabled in the permitted set are enabled in the effective set.

If a thread that has a 0 value for one or more of its user IDs wants to prevent its permitted capability set being cleared when it resets all of its user IDs to nonzero values, it can do so using the **SECBIT_KEEP_CAPS** securebits flag described below.

Programmatically adjusting capability sets

A thread can retrieve and change its permitted, effective, and inheritable capability sets using the *capget*(2) and *capset*(2) system calls. However, the use of *cap_get_proc*(3) and *cap_set_proc*(3), both provided in the *libcap* package, is preferred for this purpose. The following rules govern changes to the thread capability sets:

- If the caller does not have the **CAP_SETPCAP** capability, the new inheritable set must be a subset of the combination of the existing inheritable and permitted sets.
- (Since Linux 2.6.25) The new inheritable set must be a subset of the combination of the existing inheritable set and the capability bounding set.
- The new permitted set must be a subset of the existing permitted set (i.e., it is not possible to acquire permitted capabilities that the thread does not currently have).
- The new effective set must be a subset of the new permitted set.

The securebits flags: establishing a capabilities-only environment

Starting with Linux 2.6.26, and with a kernel in which file capabilities are enabled, Linux implements a set of per-thread *securebits* flags that can be used to disable special handling of capabilities for UID 0 (*root*). These flags are as follows:

SECBIT_KEEP_CAPS

Setting this flag allows a thread that has one or more 0 UIDs to retain capabilities in its permitted set when it switches all of its UIDs to nonzero values. If this flag is not set, then such a UID switch causes the thread to lose all permitted capabilities. This flag is always cleared on an *execve*(2).

Note that even with the **SECBIT_KEEP_CAPS** flag set, the effective capabilities of a thread are cleared when it switches its effective UID to a nonzero value. However, if the thread has set this flag and its effective UID is already nonzero, and the thread subsequently switches all other UIDs to nonzero values, then the effective capabilities will not be cleared.

The setting of the **SECBIT_KEEP_CAPS** flag is ignored if the **SECBIT_NO_SETUID_FIXUP** flag is set. (The latter flag provides a superset of the effect of the former flag.)

This flag provides the same functionality as the older *prctl*(2) **PR_SET_KEEPCAPS** operation.

SECBIT_NO_SETUID_FIXUP
Setting this flag stops the kernel from adjusting the process's permitted, effective, and ambient capability sets when the thread's effective and filesystem UIDs are switched between zero and nonzero values. See *Effect of user ID changes on capabilities* above.

SECBIT_NOROOT
If this bit is set, then the kernel does not grant capabilities when a set-user-ID-root program is executed, or when a process with an effective or real UID of 0 calls *execve*(2). (See *Capabilities and execution of programs by root* above.)

SECBIT_NO_CAP_AMBIENT_RAISE
Setting this flag disallows raising ambient capabilities via the *prctl*(2) **PR_CAP_AMBIENT_RAISE** operation.

Each of the above "base" flags has a companion "locked" flag. Setting any of the "locked" flags is irreversible, and has the effect of preventing further changes to the corresponding "base" flag. The locked flags are: **SECBIT_KEEP_CAPS_LOCKED**, **SECBIT_NO_SETUID_FIXUP_LOCKED**, **SECBIT_NOROOT_LOCKED**, and **SECBIT_NO_CAP_AMBIENT_RAISE_LOCKED**.

The *securebits* flags can be modified and retrieved using the *prctl*(2) **PR_SET_SECUREBITS** and **PR_GET_SECUREBITS** operations. The **CAP_SETPCAP** capability is required to modify the flags. Note that the **SECBIT_*** constants are available only after including the <*linux/securebits.h*> header file.

The *securebits* flags are inherited by child processes. During an *execve*(2), all of the flags are preserved, except **SECBIT_KEEP_CAPS** which is always cleared.

An application can use the following call to lock itself, and all of its descendants, into an environment where the only way of gaining capabilities is by executing a program with associated file capabilities:

```
prctl(PR_SET_SECUREBITS,
        /* SECBIT_KEEP_CAPS off */
        SECBIT_KEEP_CAPS_LOCKED |
        SECBIT_NO_SETUID_FIXUP |
        SECBIT_NO_SETUID_FIXUP_LOCKED |
        SECBIT_NOROOT |
        SECBIT_NOROOT_LOCKED);
        /* Setting/locking SECBIT_NO_CAP_AMBIENT_RAISE
           is not required */
```

Per-user-namespace "set-user-ID-root" programs
A set-user-ID program whose UID matches the UID that created a user namespace will confer capabilities in the process's permitted and effective sets when executed by any process inside that namespace or any descendant user namespace.

The rules about the transformation of the process's capabilities during the *execve*(2) are exactly as described in *Transformation of capabilities during execve()* and *Capabilities and execution of programs by root* above, with the difference that, in the latter subsection, "root" is the UID of the creator of the user namespace.

Namespaced file capabilities
Traditional (i.e., version 2) file capabilities associate only a set of capability masks with a binary executable file. When a process executes a binary with such capabilities, it gains the associated capabilities (within its user namespace) as per the rules described in *Transformation of capabilities during execve()* above.

Because version 2 file capabilities confer capabilities to the executing process regardless of which user namespace it resides in, only privileged processes are permitted to associate capabilities with a file. Here, "privileged" means a process that has the **CAP_SETFCAP** capability in the user namespace where the filesystem was mounted (normally the initial user namespace). This limitation renders file

capabilities useless for certain use cases. For example, in user-namespaced containers, it can be desirable to be able to create a binary that confers capabilities only to processes executed inside that container, but not to processes that are executed outside the container.

Linux 4.14 added so-called namespaced file capabilities to support such use cases. Namespaced file capabilities are recorded as version 3 (i.e., **VFS_CAP_REVISION_3**) *security.capability* extended attributes. Such an attribute is automatically created in the circumstances described in *File capability extended attribute versioning* above. When a version 3 *security.capability* extended attribute is created, the kernel records not just the capability masks in the extended attribute, but also the namespace root user ID.

As with a binary that has **VFS_CAP_REVISION_2** file capabilities, a binary with **VFS_CAP_REVISION_3** file capabilities confers capabilities to a process during **execve**(). However, capabilities are conferred only if the binary is executed by a process that resides in a user namespace whose UID 0 maps to the root user ID that is saved in the extended attribute, or when executed by a process that resides in a descendant of such a namespace.

Interaction with user namespaces

For further information on the interaction of capabilities and user namespaces, see *user_namespaces*(7).

STANDARDS

No standards govern capabilities, but the Linux capability implementation is based on the withdrawn POSIX.1e draft standard.

NOTES

When attempting to *strace*(1) binaries that have capabilities (or set-user-ID-root binaries), you may find the *−u <username>* option useful. Something like:

```
$ sudo strace -o trace.log -u ceci ./myprivprog
```

From Linux 2.5.27 to Linux 2.6.26, capabilities were an optional kernel component, and could be enabled/disabled via the **CONFIG_SECURITY_CAPABILITIES** kernel configuration option.

The */proc/*pid*/task/TID/status* file can be used to view the capability sets of a thread. The */proc/*pid*/status* file shows the capability sets of a process's main thread. Before Linux 3.8, nonexistent capabilities were shown as being enabled (1) in these sets. Since Linux 3.8, all nonexistent capabilities (above **CAP_LAST_CAP**) are shown as disabled (0).

The *libcap* package provides a suite of routines for setting and getting capabilities that is more comfortable and less likely to change than the interface provided by *capset*(2) and *capget*(2). This package also provides the *setcap*(8) and *getcap*(8) programs. It can be found at

.

Before Linux 2.6.24, and from Linux 2.6.24 to Linux 2.6.32 if file capabilities are not enabled, a thread with the **CAP_SETPCAP** capability can manipulate the capabilities of threads other than itself. However, this is only theoretically possible, since no thread ever has **CAP_SETPCAP** in either of these cases:

- In the pre-2.6.25 implementation the system-wide capability bounding set, */proc/sys/kernel/cap−bound*, always masks out the **CAP_SETPCAP** capability, and this can not be changed without modifying the kernel source and rebuilding the kernel.
- If file capabilities are disabled (i.e., the kernel **CONFIG_SECURITY_FILE_CAPABILITIES** option is disabled), then **init** starts out with the **CAP_SETPCAP** capability removed from its per-process bounding set, and that bounding set is inherited by all other processes created on the system.

SEE ALSO

capsh(1), *setpriv*(1), *prctl*(2), *setfsuid*(2), *cap_clear*(3), *cap_copy_ext*(3), *cap_from_text*(3), *cap_get_file*(3), *cap_get_proc*(3), *cap_init*(3), *capgetp*(3), *capsetp*(3), *libcap*(3), *proc*(5), *credentials*(7), *pthreads*(7), *user_namespaces*(7), *captest*(8), *filecap*(8), *getcap*(8), *getpcaps*(8), *netcap*(8), *pscap*(8), *setcap*(8)

include/linux/capability.h in the Linux kernel source tree

NAME

cgroup_namespaces − overview of Linux cgroup namespaces

DESCRIPTION

For an overview of namespaces, see *namespaces*(7).

Cgroup namespaces virtualize the view of a process's cgroups (see *cgroups*(7)) as seen via */proc/*pid*/cgroup* and */proc/*pid*/mountinfo*.

Each cgroup namespace has its own set of cgroup root directories. These root directories are the base points for the relative locations displayed in the corresponding records in the */proc/*pid*/cgroup* file. When a process creates a new cgroup namespace using *clone*(2) or *unshare*(2) with the **CLONE_NEWCGROUP** flag, its current cgroups directories become the cgroup root directories of the new namespace. (This applies both for the cgroups version 1 hierarchies and the cgroups version 2 unified hierarchy.)

When reading the cgroup memberships of a "target" process from */proc/*pid*/cgroup*, the pathname shown in the third field of each record will be relative to the reading process's root directory for the corresponding cgroup hierarchy. If the cgroup directory of the target process lies outside the root directory of the reading process's cgroup namespace, then the pathname will show ../ entries for each ancestor level in the cgroup hierarchy.

The following shell session demonstrates the effect of creating a new cgroup namespace.

First, (as superuser) in a shell in the initial cgroup namespace, we create a child cgroup in the *freezer* hierarchy, and place a process in that cgroup that we will use as part of the demonstration below:

```
# mkdir -p /sys/fs/cgroup/freezer/sub2
# sleep 10000 &     # Create a process that lives for a while
[1] 20124
# echo 20124 > /sys/fs/cgroup/freezer/sub2/cgroup.procs
```

We then create another child cgroup in the *freezer* hierarchy and put the shell into that cgroup:

```
# mkdir -p /sys/fs/cgroup/freezer/sub
# echo $$                     # Show PID of this shell
30655
# echo 30655 > /sys/fs/cgroup/freezer/sub/cgroup.procs
# cat /proc/self/cgroup | grep freezer
7:freezer:/sub
```

Next, we use *unshare*(1) to create a process running a new shell in new cgroup and mount namespaces:

```
# PS1="sh2# " unshare -Cm bash
```

From the new shell started by *unshare*(1), we then inspect the */proc/*pid*/cgroup* files of, respectively, the new shell, a process that is in the initial cgroup namespace (*init*, with PID 1), and the process in the sibling cgroup (*sub2*):

```
sh2# cat /proc/self/cgroup | grep freezer
7:freezer:/
sh2# cat /proc/1/cgroup | grep freezer
7:freezer:/..
sh2# cat /proc/20124/cgroup | grep freezer
7:freezer:/../sub2
```

From the output of the first command, we see that the freezer cgroup membership of the new shell (which is in the same cgroup as the initial shell) is shown defined relative to the freezer cgroup root directory that was established when the new cgroup namespace was created. (In absolute terms, the new shell is in the */sub* freezer cgroup, and the root directory of the freezer cgroup hierarchy in the new cgroup namespace is also */sub*. Thus, the new shell's cgroup membership is displayed as '/'.)

However, when we look in */proc/self/mountinfo* we see the following anomaly:

```
sh2# cat /proc/self/mountinfo | grep freezer
155 145 0:32 /.. /sys/fs/cgroup/freezer ...
```

The fourth field of this line (/..) should show the directory in the cgroup filesystem which forms the root of this mount. Since by the definition of cgroup namespaces, the process's current freezer cgroup

directory became its root freezer cgroup directory, we should see '/' in this field. The problem here is that we are seeing a mount entry for the cgroup filesystem corresponding to the initial cgroup namespace (whose cgroup filesystem is indeed rooted at the parent directory of *sub*). To fix this problem, we must remount the freezer cgroup filesystem from the new shell (i.e., perform the mount from a process that is in the new cgroup namespace), after which we see the expected results:

```
sh2# mount --make-rslave /     # Don't propagate mount events
                               # to other namespaces
sh2# umount /sys/fs/cgroup/freezer
sh2# mount -t cgroup -o freezer freezer /sys/fs/cgroup/freezer
sh2# cat /proc/self/mountinfo | grep freezer
155 145 0:32 / /sys/fs/cgroup/freezer rw,relatime ...
```

STANDARDS

Linux.

NOTES

Use of cgroup namespaces requires a kernel that is configured with the **CONFIG_CGROUPS** option.

The virtualization provided by cgroup namespaces serves a number of purposes:

- It prevents information leaks whereby cgroup directory paths outside of a container would otherwise be visible to processes in the container. Such leakages could, for example, reveal information about the container framework to containerized applications.
- It eases tasks such as container migration. The virtualization provided by cgroup namespaces allows containers to be isolated from knowledge of the pathnames of ancestor cgroups. Without such isolation, the full cgroup pathnames (displayed in */proc/self/cgroups*) would need to be replicated on the target system when migrating a container; those pathnames would also need to be unique, so that they don't conflict with other pathnames on the target system.
- It allows better confinement of containerized processes, because it is possible to mount the container's cgroup filesystems such that the container processes can't gain access to ancestor cgroup directories. Consider, for example, the following scenario:
  - We have a cgroup directory, */cg/1*, that is owned by user ID 9000.
  - We have a process, *X*, also owned by user ID 9000, that is namespaced under the cgroup */cg/1/2* (i.e., *X* was placed in a new cgroup namespace via *clone*(2) or *unshare*(2) with the **CLONE_NEWCGROUP** flag).

  In the absence of cgroup namespacing, because the cgroup directory */cg/1* is owned (and writable) by UID 9000 and process *X* is also owned by user ID 9000, process *X* would be able to modify the contents of cgroups files (i.e., change cgroup settings) not only in */cg/1/2* but also in the ancestor cgroup directory */cg/1*. Namespacing process *X* under the cgroup directory */cg/1/2*, in combination with suitable mount operations for the cgroup filesystem (as shown above), prevents it modifying files in */cg/1*, since it cannot even see the contents of that directory (or of further removed cgroup ancestor directories). Combined with correct enforcement of hierarchical limits, this prevents process *X* from escaping the limits imposed by ancestor cgroups.

SEE ALSO

unshare(1), *clone*(2), *setns*(2), *unshare*(2), *proc*(5), *cgroups*(7), *credentials*(7), *namespaces*(7), *user_namespaces*(7)

NAME

cgroups − Linux control groups

DESCRIPTION

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on).

Terminology

A *cgroup* is a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem.

A *subsystem* is a kernel component that modifies the behavior of the processes in a cgroup. Various subsystems have been implemented, making it possible to do things such as limiting the amount of CPU time and memory available to a cgroup, accounting for the CPU time used by a cgroup, and freezing and resuming execution of the processes in a cgroup. Subsystems are sometimes also known as *resource controllers* (or simply, controllers).

The cgroups for a controller are arranged in a *hierarchy*. This hierarchy is defined by creating, removing, and renaming subdirectories within the cgroup filesystem. At each level of the hierarchy, attributes (e.g., limits) can be defined. The limits, control, and accounting provided by cgroups generally have effect throughout the subhierarchy underneath the cgroup where the attributes are defined. Thus, for example, the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups.

Cgroups version 1 and version 2

The initial release of the cgroups implementation was in Linux 2.6.24. Over time, various cgroup controllers have been added to allow the management of various types of resources. However, the development of these controllers was largely uncoordinated, with the result that many inconsistencies arose between controllers and management of the cgroup hierarchies became rather complex. A longer description of these problems can be found in the kernel source file *Documentation/admin−guide/cgroup−v2.rst* (or *Documentation/cgroup−v2.txt* in Linux 4.17 and earlier).

Because of the problems with the initial cgroups implementation (cgroups version 1), starting in Linux 3.10, work began on a new, orthogonal implementation to remedy these problems. Initially marked experimental, and hidden behind the *−o __DEVEL__sane_behavior* mount option, the new version (cgroups version 2) was eventually made official with the release of Linux 4.5. Differences between the two versions are described in the text below. The file *cgroup.sane_behavior*, present in cgroups v1, is a relic of this mount option. The file always reports "0" and is only retained for backward compatibility.

Although cgroups v2 is intended as a replacement for cgroups v1, the older system continues to exist (and for compatibility reasons is unlikely to be removed). Currently, cgroups v2 implements only a subset of the controllers available in cgroups v1. The two systems are implemented so that both v1 controllers and v2 controllers can be mounted on the same system. Thus, for example, it is possible to use those controllers that are supported under version 2, while also using version 1 controllers where version 2 does not yet support those controllers. The only restriction here is that a controller can't be simultaneously employed in both a cgroups v1 hierarchy and in the cgroups v2 hierarchy.

CGROUPS VERSION 1

Under cgroups v1, each controller may be mounted against a separate cgroup filesystem that provides its own hierarchical organization of the processes on the system. It is also possible to comount multiple (or even all) cgroups v1 controllers against the same cgroup filesystem, meaning that the comounted controllers manage the same hierarchical organization of processes.

For each mounted hierarchy, the directory tree mirrors the control group hierarchy. Each control group is represented by a directory, with each of its child control cgroups represented as a child directory. For instance, */user/joe/1.session* represents control group *1.session*, which is a child of cgroup *joe*, which is a child of */user*. Under each cgroup directory is a set of files which can be read or written to, reflecting resource limits and a few general cgroup properties.

Tasks (threads) versus processes

In cgroups v1, a distinction is drawn between *processes* and *tasks*. In this view, a process can consist of multiple tasks (more commonly called threads, from a user-space perspective, and called such in the remainder of this man page). In cgroups v1, it is possible to independently manipulate the cgroup memberships of the threads in a process.

The cgroups v1 ability to split threads across different cgroups caused problems in some cases. For example, it made no sense for the *memory* controller, since all of the threads of a process share a single address space. Because of these problems, the ability to independently manipulate the cgroup memberships of the threads in a process was removed in the initial cgroups v2 implementation, and subsequently restored in a more limited form (see the discussion of "thread mode" below).

Mounting v1 controllers

The use of cgroups requires a kernel built with the **CONFIG_CGROUP** option. In addition, each of the v1 controllers has an associated configuration option that must be set in order to employ that controller.

In order to use a v1 controller, it must be mounted against a cgroup filesystem. The usual place for such mounts is under a *tmpfs*(5) filesystem mounted at */sys/fs/cgroup*. Thus, one might mount the *cpu* controller as follows:

```
mount -t cgroup -o cpu none /sys/fs/cgroup/cpu
```

It is possible to comount multiple controllers against the same hierarchy. For example, here the *cpu* and *cpuacct* controllers are comounted against a single hierarchy:

```
mount -t cgroup -o cpu,cpuacct none /sys/fs/cgroup/cpu,cpuacct
```

Comounting controllers has the effect that a process is in the same cgroup for all of the comounted controllers. Separately mounting controllers allows a process to be in cgroup */foo1* for one controller while being in */foo2/foo3* for another.

It is possible to comount all v1 controllers against the same hierarchy:

```
mount -t cgroup -o all cgroup /sys/fs/cgroup
```

(One can achieve the same result by omitting *-o all*, since it is the default if no controllers are explicitly specified.)

It is not possible to mount the same controller against multiple cgroup hierarchies. For example, it is not possible to mount both the *cpu* and *cpuacct* controllers against one hierarchy, and to mount the *cpu* controller alone against another hierarchy. It is possible to create multiple mount with exactly the same set of comounted controllers. However, in this case all that results is multiple mount points providing a view of the same hierarchy.

Note that on many systems, the v1 controllers are automatically mounted under */sys/fs/cgroup*; in particular, *systemd*(1) automatically creates such mounts.

Unmounting v1 controllers

A mounted cgroup filesystem can be unmounted using the *umount*(8) command, as in the following example:

```
umount /sys/fs/cgroup/pids
```

But note well: a cgroup filesystem is unmounted only if it is not busy, that is, it has no child cgroups. If this is not the case, then the only effect of the *umount*(8) is to make the mount invisible. Thus, to ensure that the mount is really removed, one must first remove all child cgroups, which in turn can be done only after all member processes have been moved from those cgroups to the root cgroup.

Cgroups version 1 controllers

Each of the cgroups version 1 controllers is governed by a kernel configuration option (listed below). Additionally, the availability of the cgroups feature is governed by the **CONFIG_CGROUPS** kernel configuration option.

cpu (since Linux 2.6.24; **CONFIG_CGROUP_SCHED**)

Cgroups can be guaranteed a minimum number of "CPU shares" when a system is busy. This does not limit a cgroup's CPU usage if the CPUs are not busy. For further information, see *Documentation/scheduler/sched-design-CFS.rst* (or *Documentation/scheduler/sched-design-CFS.txt* in Linux 5.2 and earlier).

In Linux 3.2, this controller was extended to provide CPU "bandwidth" control. If the kernel is configured with **CONFIG_CFS_BANDWIDTH**, then within each scheduling period (defined via a file in the cgroup directory), it is possible to define an upper limit on the CPU time allocated to the processes in a cgroup. This upper limit applies even if there is no other competition for the CPU. Further information can be found in the kernel source file *Documentation/scheduler/sched−bwc.rst* (or *Documentation/scheduler/sched−bwc.txt* in Linux 5.2 and earlier).

cpuacct (since Linux 2.6.24; **CONFIG_CGROUP_CPUACCT**)
This provides accounting for CPU usage by groups of processes.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/cpuacct.rst* (or *Documentation/cgroup−v1/cpuacct.txt* in Linux 5.2 and earlier).

cpuset (since Linux 2.6.24; **CONFIG_CPUSETS**)
This cgroup can be used to bind the processes in a cgroup to a specified set of CPUs and NUMA nodes.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/cpusets.rst* (or *Documentation/cgroup−v1/cpusets.txt* in Linux 5.2 and earlier).

memory (since Linux 2.6.25; **CONFIG_MEMCG**)
The memory controller supports reporting and limiting of process memory, kernel memory, and swap used by cgroups.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/memory.rst* (or *Documentation/cgroup−v1/memory.txt* in Linux 5.2 and earlier).

devices (since Linux 2.6.26; **CONFIG_CGROUP_DEVICE**)
This supports controlling which processes may create (mknod) devices as well as open them for reading or writing. The policies may be specified as allow-lists and deny-lists. Hierarchy is enforced, so new rules must not violate existing rules for the target or ancestor cgroups.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/devices.rst* (or *Documentation/cgroup−v1/devices.txt* in Linux 5.2 and earlier).

freezer (since Linux 2.6.28; **CONFIG_CGROUP_FREEZER**)
The *freezer* cgroup can suspend and restore (resume) all processes in a cgroup. Freezing a cgroup */A* also causes its children, for example, processes in */A/B*, to be frozen.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/freezer−subsystem.rst* (or *Documentation/cgroup−v1/freezer−subsystem.txt* in Linux 5.2 and earlier).

net_cls (since Linux 2.6.29; **CONFIG_CGROUP_NET_CLASSID**)
This places a classid, specified for the cgroup, on network packets created by a cgroup. These classids can then be used in firewall rules, as well as used to shape traffic using *tc*(8)This applies only to packets leaving the cgroup, not to traffic arriving at the cgroup.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/net_cls.rst* (or *Documentation/cgroup−v1/net_cls.txt* in Linux 5.2 and earlier).

blkio (since Linux 2.6.33; **CONFIG_BLK_CGROUP**)
The *blkio* cgroup controls and limits access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy.

Two policies are available. The first is a proportional-weight time-based division of disk implemented with CFQ. This is in effect for leaf nodes using CFQ. The second is a throttling policy which specifies upper I/O rate limits on a device.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/blkio−controller.rst* (or

Documentation/cgroup−v1/blkio−controller.txt in Linux 5.2 and earlier).

perf_event (since Linux 2.6.39; **CONFIG_CGROUP_PERF**)
This controller allows *perf* monitoring of the set of processes grouped in a cgroup.

Further information can be found in the kernel source files

net_prio (since Linux 3.3; **CONFIG_CGROUP_NET_PRIO**)
This allows priorities to be specified, per network interface, for cgroups.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/net_prio.rst* (or *Documentation/cgroup−v1/net_prio.txt* in Linux 5.2 and earlier).

hugetlb (since Linux 3.5; **CONFIG_CGROUP_HUGETLB**)
This supports limiting the use of huge pages by cgroups.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/hugetlb.rst* (or *Documentation/cgroup−v1/hugetlb.txt* in Linux 5.2 and earlier).

pids (since Linux 4.3; **CONFIG_CGROUP_PIDS**)
This controller permits limiting the number of process that may be created in a cgroup (and its descendants).

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/pids.rst* (or *Documentation/cgroup−v1/pids.txt* in Linux 5.2 and earlier).

rdma (since Linux 4.11; **CONFIG_CGROUP_RDMA**)
The RDMA controller permits limiting the use of RDMA/IB-specific resources per cgroup.

Further information can be found in the kernel source file *Documentation/admin−guide/cgroup−v1/rdma.rst* (or *Documentation/cgroup−v1/rdma.txt* in Linux 5.2 and earlier).

Creating cgroups and moving processes

A cgroup filesystem initially contains a single root cgroup, '/', which all processes belong to. A new cgroup is created by creating a directory in the cgroup filesystem:

```
mkdir /sys/fs/cgroup/cpu/cg1
```

This creates a new empty cgroup.

A process may be moved to this cgroup by writing its PID into the cgroup's *cgroup.procs* file:

```
echo $$ > /sys/fs/cgroup/cpu/cg1/cgroup.procs
```

Only one PID at a time should be written to this file.

Writing the value 0 to a *cgroup.procs* file causes the writing process to be moved to the corresponding cgroup.

When writing a PID into the *cgroup.procs*, all threads in the process are moved into the new cgroup at once.

Within a hierarchy, a process can be a member of exactly one cgroup. Writing a process's PID to a *cgroup.procs* file automatically removes it from the cgroup of which it was previously a member.

The *cgroup.procs* file can be read to obtain a list of the processes that are members of a cgroup. The returned list of PIDs is not guaranteed to be in order. Nor is it guaranteed to be free of duplicates. (For example, a PID may be recycled while reading from the list.)

In cgroups v1, an individual thread can be moved to another cgroup by writing its thread ID (i.e., the kernel thread ID returned by *clone*(2) and *gettid*(2)) to the *tasks* file in a cgroup directory. This file can be read to discover the set of threads that are members of the cgroup.

Removing cgroups

To remove a cgroup, it must first have no child cgroups and contain no (nonzombie) processes. So long as that is the case, one can simply remove the corresponding directory pathname. Note that files in a cgroup directory cannot and need not be removed.

Cgroups v1 release notification

Two files can be used to determine whether the kernel provides notifications when a cgroup becomes empty. A cgroup is considered to be empty when it contains no child cgroups and no member processes.

A special file in the root directory of each cgroup hierarchy, *release_agent*, can be used to register the pathname of a program that may be invoked when a cgroup in the hierarchy becomes empty. The pathname of the newly empty cgroup (relative to the cgroup mount point) is provided as the sole command-line argument when the *release_agent* program is invoked. The *release_agent* program might remove the cgroup directory, or perhaps repopulate it with a process.

The default value of the *release_agent* file is empty, meaning that no release agent is invoked.

The content of the *release_agent* file can also be specified via a mount option when the cgroup filesystem is mounted:

```
mount -o release_agent=pathname ...
```

Whether or not the *release_agent* program is invoked when a particular cgroup becomes empty is determined by the value in the *notify_on_release* file in the corresponding cgroup directory. If this file contains the value 0, then the *release_agent* program is not invoked. If it contains the value 1, the *release_agent* program is invoked. The default value for this file in the root cgroup is 0. At the time when a new cgroup is created, the value in this file is inherited from the corresponding file in the parent cgroup.

Cgroup v1 named hierarchies

In cgroups v1, it is possible to mount a cgroup hierarchy that has no attached controllers:

```
mount -t cgroup -o none,name=somename none /some/mount/point
```

Multiple instances of such hierarchies can be mounted; each hierarchy must have a unique name. The only purpose of such hierarchies is to track processes. (See the discussion of release notification below.) An example of this is the *name=systemd* cgroup hierarchy that is used by *systemd*(1) to track services and user sessions.

Since Linux 5.0, the *cgroup_no_v1* kernel boot option (described below) can be used to disable cgroup v1 named hierarchies, by specifying *cgroup_no_v1=named*.

CGROUPS VERSION 2

In cgroups v2, all mounted controllers reside in a single unified hierarchy. While (different) controllers may be simultaneously mounted under the v1 and v2 hierarchies, it is not possible to mount the same controller simultaneously under both the v1 and the v2 hierarchies.

The new behaviors in cgroups v2 are summarized here, and in some cases elaborated in the following subsections.

- Cgroups v2 provides a unified hierarchy against which all controllers are mounted.
- "Internal" processes are not permitted. With the exception of the root cgroup, processes may reside only in leaf nodes (cgroups that do not themselves contain child cgroups). The details are somewhat more subtle than this, and are described below.
- Active cgroups must be specified via the files *cgroup.controllers* and *cgroup.subtree_control*.
- The *tasks* file has been removed. In addition, the *cgroup.clone_children* file that is employed by the *cpuset* controller has been removed.
- An improved mechanism for notification of empty cgroups is provided by the *cgroup.events* file.

For more changes, see the *Documentation/admin-guide/cgroup-v2.rst* file in the kernel source (or *Documentation/cgroup-v2.txt* in Linux 4.17 and earlier).

Some of the new behaviors listed above saw subsequent modification with the addition in Linux 4.14 of "thread mode" (described below).

Cgroups v2 unified hierarchy

In cgroups v1, the ability to mount different controllers against different hierarchies was intended to allow great flexibility for application design. In practice, though, the flexibility turned out to be less useful than expected, and in many cases added complexity. Therefore, in cgroups v2, all available controllers are mounted against a single hierarchy. The available controllers are automatically mounted,

meaning that it is not necessary (or possible) to specify the controllers when mounting the cgroup v2 filesystem using a command such as the following:

```
mount -t cgroup2 none /mnt/cgroup2
```

A cgroup v2 controller is available only if it is not currently in use via a mount against a cgroup v1 hierarchy. Or, to put things another way, it is not possible to employ the same controller against both a v1 hierarchy and the unified v2 hierarchy. This means that it may be necessary first to unmount a v1 controller (as described above) before that controller is available in v2. Since *systemd*(1) makes heavy use of some v1 controllers by default, it can in some cases be simpler to boot the system with selected v1 controllers disabled. To do this, specify the *cgroup_no_v1=list* option on the kernel boot command line; *list* is a comma-separated list of the names of the controllers to disable, or the word *all* to disable all v1 controllers. (This situation is correctly handled by *systemd*(1), which falls back to operating without the specified controllers.)

Note that on many modern systems, *systemd*(1) automatically mounts the *cgroup2* filesystem at */sys/fs/cgroup/unified* during the boot process.

Cgroups v2 mount options

The following options (*mount -o*) can be specified when mounting the group v2 filesystem:

nsdelegate (since Linux 4.15)
: Treat cgroup namespaces as delegation boundaries. For details, see below.

memory_localevents (since Linux 5.2)
: The *memory.events* should show statistics only for the cgroup itself, and not for any descendant cgroups. This was the behavior before Linux 5.2. Starting in Linux 5.2, the default behavior is to include statistics for descendant cgroups in *memory.events*, and this mount option can be used to revert to the legacy behavior. This option is system wide and can be set on mount or modified through remount only from the initial mount namespace; it is silently ignored in noninitial namespaces.

Cgroups v2 controllers

The following controllers, documented in the kernel source file *Documentation/admin-guide/cgroup-v2.rst* (or *Documentation/cgroup-v2.txt* in Linux 4.17 and earlier), are supported in cgroups version 2:

cpu (since Linux 4.15)
: This is the successor to the version 1 *cpu* and *cpuacct* controllers.

cpuset (since Linux 5.0)
: This is the successor of the version 1 *cpuset* controller.

freezer (since Linux 5.2)
: This is the successor of the version 1 *freezer* controller.

hugetlb (since Linux 5.6)
: This is the successor of the version 1 *hugetlb* controller.

io (since Linux 4.5)
: This is the successor of the version 1 *blkio* controller.

memory (since Linux 4.5)
: This is the successor of the version 1 *memory* controller.

perf_event (since Linux 4.11)
: This is the same as the version 1 *perf_event* controller.

pids (since Linux 4.5)
: This is the same as the version 1 *pids* controller.

rdma (since Linux 4.11)
: This is the same as the version 1 *rdma* controller.

There is no direct equivalent of the *net_cls* and *net_prio* controllers from cgroups version 1. Instead, support has been added to *iptables*(8) to allow eBPF filters that hook on cgroup v2 pathnames to make decisions about network traffic on a per-cgroup basis.

The v2 *devices* controller provides no interface files; instead, device control is gated by attaching an

eBPF (**BPF_CGROUP_DEVICE**) program to a v2 cgroup.

Cgroups v2 subtree control

Each cgroup in the v2 hierarchy contains the following two files:

cgroup.controllers

This read-only file exposes a list of the controllers that are *available* in this cgroup. The contents of this file match the contents of the *cgroup.subtree_control* file in the parent cgroup.

cgroup.subtree_control

This is a list of controllers that are *active* (*enabled*) in the cgroup. The set of controllers in this file is a subset of the set in the *cgroup.controllers* of this cgroup. The set of active controllers is modified by writing strings to this file containing space-delimited controller names, each preceded by '+' (to enable a controller) or '−' (to disable a controller), as in the following example:

```
echo '+pids -memory' > x/y/cgroup.subtree_control
```

An attempt to enable a controller that is not present in *cgroup.controllers* leads to an **ENOENT** error when writing to the *cgroup.subtree_control* file.

Because the list of controllers in *cgroup.subtree_control* is a subset of those *cgroup.controllers*, a controller that has been disabled in one cgroup in the hierarchy can never be re-enabled in the subtree below that cgroup.

A cgroup's *cgroup.subtree_control* file determines the set of controllers that are exercised in the *child* cgroups. When a controller (e.g., *pids*) is present in the *cgroup.subtree_control* file of a parent cgroup, then the corresponding controller-interface files (e.g., *pids.max*) are automatically created in the children of that cgroup and can be used to exert resource control in the child cgroups.

Cgroups v2 "no internal processes" rule

Cgroups v2 enforces a so-called "no internal processes" rule. Roughly speaking, this rule means that, with the exception of the root cgroup, processes may reside only in leaf nodes (cgroups that do not themselves contain child cgroups). This avoids the need to decide how to partition resources between processes which are members of cgroup A and processes in child cgroups of A.

For instance, if cgroup */cg1/cg2* exists, then a process may reside in */cg1/cg2*, but not in */cg1*. This is to avoid an ambiguity in cgroups v1 with respect to the delegation of resources between processes in */cg1* and its child cgroups. The recommended approach in cgroups v2 is to create a subdirectory called *leaf* for any nonleaf cgroup which should contain processes, but no child cgroups. Thus, processes which previously would have gone into */cg1* would now go into */cg1/leaf*. This has the advantage of making explicit the relationship between processes in */cg1/leaf* and */cg1*'s other children.

The "no internal processes" rule is in fact more subtle than stated above. More precisely, the rule is that a (nonroot) cgroup can't both (1) have member processes, and (2) distribute resources into child cgroups—that is, have a nonempty *cgroup.subtree_control* file. Thus, it *is* possible for a cgroup to have both member processes and child cgroups, but before controllers can be enabled for that cgroup, the member processes must be moved out of the cgroup (e.g., perhaps into the child cgroups).

With the Linux 4.14 addition of "thread mode" (described below), the "no internal processes" rule has been relaxed in some cases.

Cgroups v2 cgroup.events file

Each nonroot cgroup in the v2 hierarchy contains a read-only file, *cgroup.events*, whose contents are key-value pairs (delimited by newline characters, with the key and value separated by spaces) providing state information about the cgroup:

```
$ cat mygrp/cgroup.events
populated 1
frozen 0
```

The following keys may appear in this file:

populated

The value of this key is either 1, if this cgroup or any of its descendants has member processes, or otherwise 0.

frozen (since Linux 5.2)
The value of this key is 1 if this cgroup is currently frozen, or 0 if it is not.

The *cgroup.events* file can be monitored, in order to receive notification when the value of one of its keys changes. Such monitoring can be done using *inotify*(7), which notifies changes as **IN_MODIFY** events, or *poll*(2), which notifies changes by returning the **POLLPRI** and **POLLERR** bits in the *revents* field.

Cgroup v2 release notification

Cgroups v2 provides a new mechanism for obtaining notification when a cgroup becomes empty. The cgroups v1 *release_agent* and *notify_on_release* files are removed, and replaced by the *populated* key in the *cgroup.events* file. This key either has the value 0, meaning that the cgroup (and its descendants) contain no (nonzombie) member processes, or 1, meaning that the cgroup (or one of its descendants) contains member processes.

The cgroups v2 release-notification mechanism offers the following advantages over the cgroups v1 *release_agent* mechanism:

- It allows for cheaper notification, since a single process can monitor multiple *cgroup.events* files (using the techniques described earlier). By contrast, the cgroups v1 mechanism requires the expense of creating a process for each notification.
- Notification for different cgroup subhierarchies can be delegated to different processes. By contrast, the cgroups v1 mechanism allows only one release agent for an entire hierarchy.

Cgroups v2 cgroup.stat file

Each cgroup in the v2 hierarchy contains a read-only *cgroup.stat* file (first introduced in Linux 4.14) that consists of lines containing key-value pairs. The following keys currently appear in this file:

nr_descendants
This is the total number of visible (i.e., living) descendant cgroups underneath this cgroup.

nr_dying_descendants
This is the total number of dying descendant cgroups underneath this cgroup. A cgroup enters the dying state after being deleted. It remains in that state for an undefined period (which will depend on system load) while resources are freed before the cgroup is destroyed. Note that the presence of some cgroups in the dying state is normal, and is not indicative of any problem.

A process can't be made a member of a dying cgroup, and a dying cgroup can't be brought back to life.

Limiting the number of descendant cgroups

Each cgroup in the v2 hierarchy contains the following files, which can be used to view and set limits on the number of descendant cgroups under that cgroup:

cgroup.max.depth (since Linux 4.14)
This file defines a limit on the depth of nesting of descendant cgroups. A value of 0 in this file means that no descendant cgroups can be created. An attempt to create a descendant whose nesting level exceeds the limit fails (*mkdir*(2) fails with the error **EAGAIN**).

Writing the string *"max"* to this file means that no limit is imposed. The default value in this file is *"max"*.

cgroup.max.descendants (since Linux 4.14)
This file defines a limit on the number of live descendant cgroups that this cgroup may have. An attempt to create more descendants than allowed by the limit fails (*mkdir*(2) fails with the error **EAGAIN**).

Writing the string *"max"* to this file means that no limit is imposed. The default value in this file is *"max"*.

CGROUPS DELEGATION: DELEGATING A HIERARCHY TO A LESS PRIVILEGED USER

In the context of cgroups, delegation means passing management of some subtree of the cgroup hierarchy to a nonprivileged user. Cgroups v1 provides support for delegation based on file permissions in the cgroup hierarchy but with less strict containment rules than v2 (as noted below). Cgroups v2 supports delegation with containment by explicit design. The focus of the discussion in this section is on

delegation in cgroups v2, with some differences for cgroups v1 noted along the way.

Some terminology is required in order to describe delegation. A *delegater* is a privileged user (i.e., root) who owns a parent cgroup. A *delegatee* is a nonprivileged user who will be granted the permissions needed to manage some subhierarchy under that parent cgroup, known as the *delegated subtree*.

To perform delegation, the delegater makes certain directories and files writable by the delegatee, typically by changing the ownership of the objects to be the user ID of the delegatee. Assuming that we want to delegate the hierarchy rooted at (say) */dlgt_grp* and that there are not yet any child cgroups under that cgroup, the ownership of the following is changed to the user ID of the delegatee:

/dlgt_grp
: Changing the ownership of the root of the subtree means that any new cgroups created under the subtree (and the files they contain) will also be owned by the delegatee.

/dlgt_grp/cgroup.procs
: Changing the ownership of this file means that the delegatee can move processes into the root of the delegated subtree.

/dlgt_grp/cgroup.subtree_control (cgroups v2 only)
: Changing the ownership of this file means that the delegatee can enable controllers (that are present in */dlgt_grp/cgroup.controllers*) in order to further redistribute resources at lower levels in the subtree. (As an alternative to changing the ownership of this file, the delegater might instead add selected controllers to this file.)

/dlgt_grp/cgroup.threads (cgroups v2 only)
: Changing the ownership of this file is necessary if a threaded subtree is being delegated (see the description of "thread mode", below). This permits the delegatee to write thread IDs to the file. (The ownership of this file can also be changed when delegating a domain subtree, but currently this serves no purpose, since, as described below, it is not possible to move a thread between domain cgroups by writing its thread ID to the *cgroup.threads* file.)

  In cgroups v1, the corresponding file that should instead be delegated is the *tasks* file.

The delegater should *not* change the ownership of any of the controller interfaces files (e.g., *pids.max*, *memory.high*) in *dlgt_grp*. Those files are used from the next level above the delegated subtree in order to distribute resources into the subtree, and the delegatee should not have permission to change the resources that are distributed into the delegated subtree.

See also the discussion of the */sys/kernel/cgroup/delegate* file in NOTES for information about further delegatable files in cgroups v2.

After the aforementioned steps have been performed, the delegatee can create child cgroups within the delegated subtree (the cgroup subdirectories and the files they contain will be owned by the delegatee) and move processes between cgroups in the subtree. If some controllers are present in *dlgt_grp/cgroup.subtree_control*, or the ownership of that file was passed to the delegatee, the delegatee can also control the further redistribution of the corresponding resources into the delegated subtree.

Cgroups v2 delegation: nsdelegate and cgroup namespaces

Starting with Linux 4.13, there is a second way to perform cgroup delegation in the cgroups v2 hierarchy. This is done by mounting or remounting the cgroup v2 filesystem with the *nsdelegate* mount option. For example, if the cgroup v2 filesystem has already been mounted, we can remount it with the *nsdelegate* option as follows:

```
mount -t cgroup2 -o remount,nsdelegate \
                 none /sys/fs/cgroup/unified
```

The effect of this mount option is to cause cgroup namespaces to automatically become delegation boundaries. More specifically, the following restrictions apply for processes inside the cgroup namespace:

- Writes to controller interface files in the root directory of the namespace will fail with the error **EPERM**. Processes inside the cgroup namespace can still write to delegatable files in the root directory of the cgroup namespace such as *cgroup.procs* and *cgroup.subtree_control*, and can create subhierarchy underneath the root directory.

- Attempts to migrate processes across the namespace boundary are denied (with the error **ENOENT**). Processes inside the cgroup namespace can still (subject to the containment rules described below) move processes between cgroups *within* the subhierarchy under the namespace root.

The ability to define cgroup namespaces as delegation boundaries makes cgroup namespaces more useful. To understand why, suppose that we already have one cgroup hierarchy that has been delegated to a nonprivileged user, *cecilia*, using the older delegation technique described above. Suppose further that *cecilia* wanted to further delegate a subhierarchy under the existing delegated hierarchy. (For example, the delegated hierarchy might be associated with an unprivileged container run by *cecilia*.) Even if a cgroup namespace was employed, because both hierarchies are owned by the unprivileged user *cecilia*, the following illegitimate actions could be performed:

- A process in the inferior hierarchy could change the resource controller settings in the root directory of that hierarchy. (These resource controller settings are intended to allow control to be exercised from the *parent* cgroup; a process inside the child cgroup should not be allowed to modify them.)
- A process inside the inferior hierarchy could move processes into and out of the inferior hierarchy if the cgroups in the superior hierarchy were somehow visible.

Employing the *nsdelegate* mount option prevents both of these possibilities.

The *nsdelegate* mount option only has an effect when performed in the initial mount namespace; in other mount namespaces, the option is silently ignored.

Note: On some systems, *systemd*(1) automatically mounts the cgroup v2 filesystem. In order to experiment with the *nsdelegate* operation, it may be useful to boot the kernel with the following command-line options:

```
cgroup_no_v1=all systemd.legacy_systemd_cgroup_controller
```

These options cause the kernel to boot with the cgroups v1 controllers disabled (meaning that the controllers are available in the v2 hierarchy), and tells *systemd*(1) not to mount and use the cgroup v2 hierarchy, so that the v2 hierarchy can be manually mounted with the desired options after boot-up.

Cgroup delegation containment rules

Some delegation *containment rules* ensure that the delegatee can move processes between cgroups within the delegated subtree, but can't move processes from outside the delegated subtree into the subtree or vice versa. A nonprivileged process (i.e., the delegatee) can write the PID of a "target" process into a *cgroup.procs* file only if all of the following are true:

- The writer has write permission on the *cgroup.procs* file in the destination cgroup.
- The writer has write permission on the *cgroup.procs* file in the nearest common ancestor of the source and destination cgroups. Note that in some cases, the nearest common ancestor may be the source or destination cgroup itself. This requirement is not enforced for cgroups v1 hierarchies, with the consequence that containment in v1 is less strict than in v2. (For example, in cgroups v1 the user that owns two distinct delegated subhierarchies can move a process between the hierarchies.)
- If the cgroup v2 filesystem was mounted with the *nsdelegate* option, the writer must be able to see the source and destination cgroups from its cgroup namespace.
- In cgroups v1: the effective UID of the writer (i.e., the delegatee) matches the real user ID or the saved set-user-ID of the target process. Before Linux 4.11, this requirement also applied in cgroups v2 (This was a historical requirement inherited from cgroups v1 that was later deemed unnecessary, since the other rules suffice for containment in cgroups v2.)

Note: one consequence of these delegation containment rules is that the unprivileged delegatee can't place the first process into the delegated subtree; instead, the delegater must place the first process (a process owned by the delegatee) into the delegated subtree.

CGROUPS VERSION 2 THREAD MODE

Among the restrictions imposed by cgroups v2 that were not present in cgroups v1 are the following:

- *No thread-granularity control*: all of the threads of a process must be in the same cgroup.

- *No internal processes*: a cgroup can't both have member processes and exercise controllers on child cgroups.

Both of these restrictions were added because the lack of these restrictions had caused problems in cgroups v1. In particular, the cgroups v1 ability to allow thread-level granularity for cgroup membership made no sense for some controllers. (A notable example was the *memory* controller: since threads share an address space, it made no sense to split threads across different *memory* cgroups.)

Notwithstanding the initial design decision in cgroups v2, there were use cases for certain controllers, notably the *cpu* controller, for which thread-level granularity of control was meaningful and useful. To accommodate such use cases, Linux 4.14 added *thread mode* for cgroups v2.

Thread mode allows the following:

- The creation of *threaded subtrees* in which the threads of a process may be spread across cgroups inside the tree. (A threaded subtree may contain multiple multithreaded processes.)
- The concept of *threaded controllers*, which can distribute resources across the cgroups in a threaded subtree.
- A relaxation of the "no internal processes rule", so that, within a threaded subtree, a cgroup can both contain member threads and exercise resource control over child cgroups.

With the addition of thread mode, each nonroot cgroup now contains a new file, *cgroup.type*, that exposes, and in some circumstances can be used to change, the "type" of a cgroup. This file contains one of the following type values:

domain This is a normal v2 cgroup that provides process-granularity control. If a process is a member of this cgroup, then all threads of the process are (by definition) in the same cgroup. This is the default cgroup type, and provides the same behavior that was provided for cgroups in the initial cgroups v2 implementation.

threaded
This cgroup is a member of a threaded subtree. Threads can be added to this cgroup, and controllers can be enabled for the cgroup.

domain threaded
This is a domain cgroup that serves as the root of a threaded subtree. This cgroup type is also known as "threaded root".

domain invalid
This is a cgroup inside a threaded subtree that is in an "invalid" state. Processes can't be added to the cgroup, and controllers can't be enabled for the cgroup. The only thing that can be done with this cgroup (other than deleting it) is to convert it to a *threaded* cgroup by writing the string *"threaded"* to the *cgroup.type* file.

The rationale for the existence of this "interim" type during the creation of a threaded subtree (rather than the kernel simply immediately converting all cgroups under the threaded root to the type *threaded*) is to allow for possible future extensions to the thread mode model

Threaded versus domain controllers

With the addition of threads mode, cgroups v2 now distinguishes two types of resource controllers:

- *Threaded* controllers: these controllers support thread-granularity for resource control and can be enabled inside threaded subtrees, with the result that the corresponding controller-interface files appear inside the cgroups in the threaded subtree. As at Linux 4.19, the following controllers are threaded: *cpu*, *perf_event*, and *pids*.
- *Domain* controllers: these controllers support only process granularity for resource control. From the perspective of a domain controller, all threads of a process are always in the same cgroup. Domain controllers can't be enabled inside a threaded subtree.

Creating a threaded subtree

There are two pathways that lead to the creation of a threaded subtree. The first pathway proceeds as follows:

(1) We write the string *"threaded"* to the *cgroup.type* file of a cgroup *y/z* that currently has the type *domain*. This has the following effects:

- The type of the cgroup *y/z* becomes *threaded*.
- The type of the parent cgroup, *y*, becomes *domain threaded*. The parent cgroup is the root of a threaded subtree (also known as the "threaded root").
- All other cgroups under *y* that were not already of type *threaded* (because they were inside already existing threaded subtrees under the new threaded root) are converted to type *domain invalid*. Any subsequently created cgroups under *y* will also have the type *domain invalid*.

(2) We write the string *"threaded"* to each of the *domain invalid* cgroups under *y*, in order to convert them to the type *threaded*. As a consequence of this step, all threads under the threaded root now have the type *threaded* and the threaded subtree is now fully usable. The requirement to write *"threaded"* to each of these cgroups is somewhat cumbersome, but allows for possible future extensions to the thread-mode model.

The second way of creating a threaded subtree is as follows:

(1) In an existing cgroup, *z*, that currently has the type *domain*, we (1.1) enable one or more threaded controllers and (1.2) make a process a member of *z*. (These two steps can be done in either order.) This has the following consequences:

- The type of *z* becomes *domain threaded*.
- All of the descendant cgroups of *z* that were not already of type *threaded* are converted to type *domain invalid*.

(2) As before, we make the threaded subtree usable by writing the string *"threaded"* to each of the *domain invalid* cgroups under *z*, in order to convert them to the type *threaded*.

One of the consequences of the above pathways to creating a threaded subtree is that the threaded root cgroup can be a parent only to *threaded* (and *domain invalid*) cgroups. The threaded root cgroup can't be a parent of a *domain* cgroups, and a *threaded* cgroup can't have a sibling that is a *domain* cgroup.

Using a threaded subtree

Within a threaded subtree, threaded controllers can be enabled in each subgroup whose type has been changed to *threaded*; upon doing so, the corresponding controller interface files appear in the children of that cgroup.

A process can be moved into a threaded subtree by writing its PID to the *cgroup.procs* file in one of the cgroups inside the tree. This has the effect of making all of the threads in the process members of the corresponding cgroup and makes the process a member of the threaded subtree. The threads of the process can then be spread across the threaded subtree by writing their thread IDs (see *gettid*(2)) to the *cgroup.threads* files in different cgroups inside the subtree. The threads of a process must all reside in the same threaded subtree.

As with writing to *cgroup.procs*, some containment rules apply when writing to the *cgroup.threads* file:

- The writer must have write permission on the cgroup.threads file in the destination cgroup.
- The writer must have write permission on the *cgroup.procs* file in the common ancestor of the source and destination cgroups. (In some cases, the common ancestor may be the source or destination cgroup itself.)
- The source and destination cgroups must be in the same threaded subtree. (Outside a threaded subtree, an attempt to move a thread by writing its thread ID to the *cgroup.threads* file in a different *domain* cgroup fails with the error **EOPNOTSUPP**.)

The *cgroup.threads* file is present in each cgroup (including *domain* cgroups) and can be read in order to discover the set of threads that is present in the cgroup. The set of thread IDs obtained when reading this file is not guaranteed to be ordered or free of duplicates.

The *cgroup.procs* file in the threaded root shows the PIDs of all processes that are members of the threaded subtree. The *cgroup.procs* files in the other cgroups in the subtree are not readable.

Domain controllers can't be enabled in a threaded subtree; no controller-interface files appear inside the cgroups underneath the threaded root. From the point of view of a domain controller, threaded subtrees are invisible: a multithreaded process inside a threaded subtree appears to a domain controller as a process that resides in the threaded root cgroup.

Within a threaded subtree, the "no internal processes" rule does not apply: a cgroup can both contain

member processes (or thread) and exercise controllers on child cgroups.

Rules for writing to cgroup.type and creating threaded subtrees

A number of rules apply when writing to the *cgroup.type* file:

- Only the string *"threaded"* may be written. In other words, the only explicit transition that is possible is to convert a *domain* cgroup to type *threaded*.
- The effect of writing *"threaded"* depends on the current value in *cgroup.type*, as follows:
  - *domain* or *domain threaded*: start the creation of a threaded subtree (whose root is the parent of this cgroup) via the first of the pathways described above;
  - *domain invalid*: convert this cgroup (which is inside a threaded subtree) to a usable (i.e., *threaded*) state;
  - *threaded*: no effect (a "no-op").
- We can't write to a *cgroup.type* file if the parent's type is *domain invalid*. In other words, the cgroups of a threaded subtree must be converted to the *threaded* state in a top-down manner.

There are also some constraints that must be satisfied in order to create a threaded subtree rooted at the cgroup *x*:

- There can be no member processes in the descendant cgroups of *x*. (The cgroup *x* can itself have member processes.)
- No domain controllers may be enabled in *x*'s *cgroup.subtree_control* file.

If any of the above constraints is violated, then an attempt to write *"threaded"* to a *cgroup.type* file fails with the error **ENOTSUP**.

The "domain threaded" cgroup type

According to the pathways described above, the type of a cgroup can change to *domain threaded* in either of the following cases:

- The string *"threaded"* is written to a child cgroup.
- A threaded controller is enabled inside the cgroup and a process is made a member of the cgroup.

A *domain threaded* cgroup, *x*, can revert to the type *domain* if the above conditions no longer hold true—that is, if all *threaded* child cgroups of *x* are removed and either *x* no longer has threaded controllers enabled or no longer has member processes.

When a *domain threaded* cgroup *x* reverts to the type *domain*:

- All *domain invalid* descendants of *x* that are not in lower-level threaded subtrees revert to the type *domain*.
- The root cgroups in any lower-level threaded subtrees revert to the type *domain threaded*.

Exceptions for the root cgroup

The root cgroup of the v2 hierarchy is treated exceptionally: it can be the parent of both *domain* and *threaded* cgroups. If the string *"threaded"* is written to the *cgroup.type* file of one of the children of the root cgroup, then

- The type of that cgroup becomes *threaded*.
- The type of any descendants of that cgroup that are not part of lower-level threaded subtrees changes to *domain invalid*.

Note that in this case, there is no cgroup whose type becomes *domain threaded*. (Notionally, the root cgroup can be considered as the threaded root for the cgroup whose type was changed to *threaded*.)

The aim of this exceptional treatment for the root cgroup is to allow a threaded cgroup that employs the *cpu* controller to be placed as high as possible in the hierarchy, so as to minimize the (small) cost of traversing the cgroup hierarchy.

The cgroups v2 "cpu" controller and realtime threads

As at Linux 4.19, the cgroups v2 *cpu* controller does not support control of realtime threads (specifically threads scheduled under any of the policies **SCHED_FIFO**, **SCHED_RR**, described **SCHED_DEADLINE**; see *sched*(7)). Therefore, the *cpu* controller can be enabled in the root cgroup only if all realtime threads are in the root cgroup. (If there are realtime threads in nonroot cgroups,

then a *write*(2) of the string *"+cpu"* to the *cgroup.subtree_control* file fails with the error **EINVAL**.)

On some systems, *systemd*(1) places certain realtime threads in nonroot cgroups in the v2 hierarchy. On such systems, these threads must first be moved to the root cgroup before the *cpu* controller can be enabled.

ERRORS

The following errors can occur for *mount*(2):

EBUSY
An attempt to mount a cgroup version 1 filesystem specified neither the *name*= option (to mount a named hierarchy) nor a controller name (or *all*).

NOTES

A child process created via *fork*(2) inherits its parent's cgroup memberships. A process's cgroup memberships are preserved across *execve*(2).

The *clone3*(2) **CLONE_INTO_CGROUP** flag can be used to create a child process that begins its life in a different version 2 cgroup from the parent process.

/proc files

/proc/cgroups (since Linux 2.6.24)
This file contains information about the controllers that are compiled into the kernel. An example of the contents of this file (reformatted for readability) is the following:

```
#subsys_name    hierarchy       num_cgroups     enabled
cpuset          4               1               1
cpu             8               1               1
cpuacct         8               1               1
blkio           6               1               1
memory          3               1               1
devices         10              84              1
freezer         7               1               1
net_cls         9               1               1
perf_event      5               1               1
net_prio        9               1               1
hugetlb         0               1               0
pids            2               1               1
```

The fields in this file are, from left to right:

[1] The name of the controller.

[2] The unique ID of the cgroup hierarchy on which this controller is mounted. If multiple cgroups v1 controllers are bound to the same hierarchy, then each will show the same hierarchy ID in this field. The value in this field will be 0 if:

- the controller is not mounted on a cgroups v1 hierarchy;
- the controller is bound to the cgroups v2 single unified hierarchy; or
- the controller is disabled (see below).

[3] The number of control groups in this hierarchy using this controller.

[4] This field contains the value 1 if this controller is enabled, or 0 if it has been disabled (via the *cgroup_disable* kernel command-line boot parameter).

*/proc/*pid*/cgroup* (since Linux 2.6.24)
This file describes control groups to which the process with the corresponding PID belongs. The displayed information differs for cgroups version 1 and version 2 hierarchies.

For each cgroup hierarchy of which the process is a member, there is one entry containing three colon-separated fields:

```
hierarchy-ID:controller-list:cgroup-path
```

For example:

```
5:cpuacct,cpu,cpuset:/daemons
```

The colon-separated fields are, from left to right:

[1] For cgroups version 1 hierarchies, this field contains a unique hierarchy ID number that can be matched to a hierarchy ID in */proc/cgroups*. For the cgroups version 2 hierarchy, this field contains the value 0.

[2] For cgroups version 1 hierarchies, this field contains a comma-separated list of the controllers bound to the hierarchy. For the cgroups version 2 hierarchy, this field is empty.

[3] This field contains the pathname of the control group in the hierarchy to which the process belongs. This pathname is relative to the mount point of the hierarchy.

/sys/kernel/cgroup files

/sys/kernel/cgroup/delegate (since Linux 4.15)

This file exports a list of the cgroups v2 files (one per line) that are delegatable (i.e., whose ownership should be changed to the user ID of the delegatee). In the future, the set of delegatable files may change or grow, and this file provides a way for the kernel to inform user-space applications of which files must be delegated. As at Linux 4.15, one sees the following when inspecting this file:

```
$ cat /sys/kernel/cgroup/delegate
cgroup.procs
cgroup.subtree_control
cgroup.threads
```

/sys/kernel/cgroup/features (since Linux 4.15)

Over time, the set of cgroups v2 features that are provided by the kernel may change or grow, or some features may not be enabled by default. This file provides a way for user-space applications to discover what features the running kernel supports and has enabled. Features are listed one per line:

```
$ cat /sys/kernel/cgroup/features
nsdelegate
memory_localevents
```

The entries that can appear in this file are:

memory_localevents (since Linux 5.2)
The kernel supports the *memory_localevents* mount option.

nsdelegate (since Linux 4.15)
The kernel supports the *nsdelegate* mount option.

memory_recursiveprot (since Linux 5.7)
The kernel supports the *memory_recursiveprot* mount option.

SEE ALSO

prlimit(1), *systemd*(1), *systemd-cgls*(1), *systemd-cgtop*(1), *clone*(2), *ioprio_set*(2), *perf_event_open*(2), *setrlimit*(2), *cgroup_namespaces*(7), *cpuset*(7), *namespaces*(7), *sched*(7), *user_namespaces*(7)

The kernel source file *Documentation/admin-guide/cgroup-v2.rst*.

NAME

charsets − character set standards and internationalization

DESCRIPTION

This manual page gives an overview on different character set standards and how they were used on Linux before Unicode became ubiquitous. Some of this information is still helpful for people working with legacy systems and documents.

Standards discussed include such as ASCII, GB 2312, ISO/IEC 8859, JIS, KOI8-R, KS, and Unicode.

The primary emphasis is on character sets that were actually used by locale character sets, not the myriad others that could be found in data from other systems.

ASCII

ASCII (American Standard Code For Information Interchange) is the original 7-bit character set, originally designed for American English. Also known as US-ASCII. It is currently described by the ISO/IEC 646:1991 IRV (International Reference Version) standard.

Various ASCII variants replacing the dollar sign with other currency symbols and replacing punctuation with non-English alphabetic characters to cover German, French, Spanish, and others in 7 bits emerged. All are deprecated; glibc does not support locales whose character sets are not true supersets of ASCII.

As Unicode, when using UTF-8, is ASCII-compatible, plain ASCII text still renders properly on modern UTF-8 using systems.

ISO/IEC 8859

ISO/IEC 8859 is a series of 15 8-bit character sets, all of which have ASCII in their low (7-bit) half, invisible control characters in positions 128 to 159, and 96 fixed-width graphics in positions 160–255.

Of these, the most important is ISO/IEC 8859-1 ("Latin Alphabet No. 1" / Latin-1). It was widely adopted and supported by different systems, and is gradually being replaced with Unicode. The ISO/IEC 8859-1 characters are also the first 256 characters of Unicode.

Console support for the other ISO/IEC 8859 character sets is available under Linux through user-mode utilities (such as *setfont*(8)) that modify keyboard bindings and the EGA graphics table and employ the "user mapping" font table in the console driver.

Here are brief descriptions of each character set:

ISO/IEC 8859-1 (Latin-1)
: Latin-1 covers many European languages such as Albanian, Basque, Danish, English, Faroese, Galician, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, and Swedish. The lack of the ligatures Dutch IJ/ij, French œ, and „German“ quotation marks was considered tolerable.

ISO/IEC 8859-2 (Latin-2)
: Latin-2 supports many Latin-written Central and East European languages such as Bosnian, Croatian, Czech, German, Hungarian, Polish, Slovak, and Slovene. Replacing Romanian / with ş/ţ was considered tolerable.

ISO/IEC 8859-3 (Latin-3)
: Latin-3 was designed to cover of Esperanto, Maltese, and Turkish, but ISO/IEC 8859-9 later superseded it for Turkish.

ISO/IEC 8859-4 (Latin-4)
: Latin-4 introduced letters for North European languages such as Estonian, Latvian, and Lithuanian, but was superseded by ISO/IEC 8859-10 and ISO/IEC 8859-13.

ISO/IEC 8859-5
: Cyrillic letters supporting Bulgarian, Byelorussian, Macedonian, Russian, Serbian, and (almost completely) Ukrainian. It was never widely used, see the discussion of KOI8-R/KOI8-U below.

ISO/IEC 8859-6
: Was created for Arabic. The ISO/IEC 8859-6 glyph table is a fixed font of separate letter forms, but a proper display engine should combine these using the proper initial, medial, and final forms.

ISO/IEC 8859-7
: Was created for Modern Greek in 1987, updated in 2003.

ISO/IEC 8859-8
: Supports Modern Hebrew without niqud (punctuation signs). Niqud and full-fledged Biblical Hebrew were outside the scope of this character set.

ISO/IEC 8859-9 (Latin-5)
: This is a variant of Latin-1 that replaces Icelandic letters with Turkish ones.

ISO/IEC 8859-10 (Latin-6)
: Latin-6 added the Inuit (Greenlandic) and Sami (Lappish) letters that were missing in Latin-4 to cover the entire Nordic area.

ISO/IEC 8859-11
: Supports the Thai alphabet and is nearly identical to the TIS-620 standard.

ISO/IEC 8859-12
: This character set does not exist.

ISO/IEC 8859-13 (Latin-7)
: Supports the Baltic Rim languages; in particular, it includes Latvian characters not found in Latin-4.

ISO/IEC 8859-14 (Latin-8)
: This is the Celtic character set, covering Old Irish, Manx, Gaelic, Welsh, Cornish, and Breton.

ISO/IEC 8859-15 (Latin-9)
: Latin-9 is similar to the widely used Latin-1 but replaces some less common symbols with the Euro sign and French and Finnish letters that were missing in Latin-1.

ISO/IEC 8859-16 (Latin-10)
: This character set covers many Southeast European languages, and most importantly supports Romanian more completely than Latin-2.

KOI8-R / KOI8-U
: KOI8-R is a non-ISO character set popular in Russia before Unicode. The lower half is ASCII; the upper is a Cyrillic character set somewhat better designed than ISO/IEC 8859-5. KOI8-U, based on KOI8-R, has better support for Ukrainian. Neither of these sets are ISO/IEC 2022 compatible, unlike the ISO/IEC 8859 series.

  Console support for KOI8-R is available under Linux through user-mode utilities that modify keyboard bindings and the EGA graphics table, and employ the "user mapping" font table in the console driver.

GB 2312
: GB 2312 is a mainland Chinese national standard character set used to express simplified Chinese. Just like JIS X 0208, characters are mapped into a 94x94 two-byte matrix used to construct EUC-CN. EUC-CN is the most important encoding for Linux and includes ASCII and GB 2312. Note that EUC-CN is often called as GB, GB 2312, or CN-GB.

Big5
: Big5 was a popular character set in Taiwan to express traditional Chinese. (Big5 is both a character set and an encoding.) It is a superset of ASCII. Non-ASCII characters are expressed in two bytes. Bytes 0xa1–0xfe are used as leading bytes for two-byte characters. Big5 and its extension were widely used in Taiwan and Hong Kong. It is not ISO/IEC 2022 compliant.

JIS X 0208
: JIS X 0208 is a Japanese national standard character set. Though there are some more Japanese national standard character sets (like JIS X 0201, JIS X 0212, and JIS X 0213), this is the most important one. Characters are mapped into a 94x94 two-byte matrix, whose each byte is in the range 0x21–0x7e. Note that JIS X 0208 is a character set, not an encoding. This means that JIS X 0208 itself is not used for expressing text data. JIS X 0208 is used as a component to construct encodings such as EUC-JP, Shift_JIS, and ISO/IEC 2022-JP. EUC-JP is the most important encoding for Linux and includes ASCII and JIS X 0208. In EUC-JP, JIS X 0208 characters are expressed in two bytes, each of which is the JIS X 0208 code plus 0x80.

KS X 1001

KS X 1001 is a Korean national standard character set. Just as JIS X 0208, characters are mapped into a 94x94 two-byte matrix. KS X 1001 is used like JIS X 0208, as a component to construct encodings such as EUC-KR, Johab, and ISO/IEC 2022-KR. EUC-KR is the most important encoding for Linux and includes ASCII and KS X 1001. KS C 5601 is an older name for KS X 1001.

ISO/IEC 2022 and ISO/IEC 4873

The ISO/IEC 2022 and ISO/IEC 4873 standards describe a font-control model based on VT100 practice. This model is (partially) supported by the Linux kernel and by *xterm*(1)Several ISO/IEC 2022-based character encodings have been defined, especially for Japanese.

There are 4 graphic character sets, called G0, G1, G2, and G3, and one of them is the current character set for codes with high bit zero (initially G0), and one of them is the current character set for codes with high bit one (initially G1). Each graphic character set has 94 or 96 characters, and is essentially a 7-bit character set. It uses codes either 040–0177 (041–0176) or 0240–0377 (0241–0376). G0 always has size 94 and uses codes 041–0176.

Switching between character sets is done using the shift functions **^N** (SO or LS1), **^O** (SI or LS0), ESC n (LS2), ESC o (LS3), ESC N (SS2), ESC O (SS3), ESC ~ (LS1R), ESC } (LS2R), ESC | (LS3R). The function LS*n* makes character set G*n* the current one for codes with high bit zero. The function LS*n*R makes character set G*n* the current one for codes with high bit one. The function SS*n* makes character set G*n* (*n*=2 or 3) the current one for the next character only (regardless of the value of its high order bit).

A 94-character set is designated as G*n* character set by an escape sequence ESC (xx (for G0), ESC) xx (for G1), ESC * xx (for G2), ESC + xx (for G3), where xx is a symbol or a pair of symbols found in the ISO/IEC 2375 International Register of Coded Character Sets. For example, ESC (@ selects the ISO/IEC 646 character set as G0, ESC (A selects the UK standard character set (with pound instead of number sign), ESC (B selects ASCII (with dollar instead of currency sign), ESC (M selects a character set for African languages, ESC (! A selects the Cuban character set, and so on.

A 96-character set is designated as G*n* character set by an escape sequence ESC − xx (for G1), ESC . xx (for G2) or ESC / xx (for G3). For example, ESC − G selects the Hebrew alphabet as G1.

A multibyte character set is designated as G*n* character set by an escape sequence ESC $ xx or ESC $ (xx (for G0), ESC $) xx (for G1), ESC $ * xx (for G2), ESC $ + xx (for G3). For example, ESC $ (C selects the Korean character set for G0. The Japanese character set selected by ESC $ B has a more recent version selected by ESC & @ ESC $ B.

ISO/IEC 4873 stipulates a narrower use of character sets, where G0 is fixed (always ASCII), so that G1, G2, and G3 can be invoked only for codes with the high order bit set. In particular, **^N** and **^O** are not used anymore, ESC (xx can be used only with xx=B, and ESC) xx, ESC * xx, ESC + xx are equivalent to ESC − xx, ESC . xx, ESC / xx, respectively.

TIS-620

TIS-620 is a Thai national standard character set and a superset of ASCII. In the same fashion as the ISO/IEC 8859 series, Thai characters are mapped into 0xa1–0xfe.

Unicode

Unicode (ISO/IEC 10646) is a standard which aims to unambiguously represent every character in every human language. Unicode's structure permits 20.1 bits to encode every character. Since most computers don't include 20.1-bit integers, Unicode is usually encoded as 32-bit integers internally and either a series of 16-bit integers (UTF-16) (needing two 16-bit integers only when encoding certain rare characters) or a series of 8-bit bytes (UTF-8).

Linux represents Unicode using the 8-bit Unicode Transformation Format (UTF-8). UTF-8 is a variable length encoding of Unicode. It uses 1 byte to code 7 bits, 2 bytes for 11 bits, 3 bytes for 16 bits, 4 bytes for 21 bits, 5 bytes for 26 bits, 6 bytes for 31 bits.

Let 0,1,x stand for a zero, one, or arbitrary bit. A byte 0xxxxxxx stands for the Unicode 00000000 0xxxxxxx which codes the same symbol as the ASCII 0xxxxxxx. Thus, ASCII goes unchanged into UTF-8, and people using only ASCII do not notice any change: not in code, and not in file size.

A byte 110xxxxx is the start of a 2-byte code, and 110xxxxx 10yyyyyy is assembled into 00000xxx xxyyyyyy. A byte 1110xxxx is the start of a 3-byte code, and 1110xxxx 10yyyyyy 10zzzzzz is assembled into xxxxyyyy yyzzzzzz. (When UTF-8 is used to code the 31-bit ISO/IEC 10646 then this

progression continues up to 6-byte codes.)

For most texts in ISO/IEC 8859 character sets, this means that the characters outside of ASCII are now coded with two bytes. This tends to expand ordinary text files by only one or two percent. For Russian or Greek texts, this expands ordinary text files by 100%, since text in those languages is mostly outside of ASCII. For Japanese users this means that the 16-bit codes now in common use will take three bytes. While there are algorithmic conversions from some character sets (especially ISO/IEC 8859-1) to Unicode, general conversion requires carrying around conversion tables, which can be quite large for 16-bit codes.

Note that UTF-8 is self-synchronizing: 10xxxxxx is a tail, any other byte is the head of a code. Note that the only way ASCII bytes occur in a UTF-8 stream, is as themselves. In particular, there are no embedded NULs ('\0') or '/'s that form part of some larger code.

Since ASCII, and, in particular, NUL and '/', are unchanged, the kernel does not notice that UTF-8 is being used. It does not care at all what the bytes it is handling stand for.

Rendering of Unicode data streams is typically handled through "subfont" tables which map a subset of Unicode to glyphs. Internally the kernel uses Unicode to describe the subfont loaded in video RAM. This means that in the Linux console in UTF-8 mode, one can use a character set with 512 different symbols. This is not enough for Japanese, Chinese, and Korean, but it is enough for most other purposes.

SEE ALSO

iconv(1), *ascii*(7), *iso_8859-1*(7), *unicode*(7), *utf-8*(7)

NAME

complex − basics of complex mathematics

LIBRARY

Math library (*libm*, *−lm*)

SYNOPSIS

#include <complex.h>

DESCRIPTION

Complex numbers are numbers of the form z = a+b*i, where a and b are real numbers and i = sqrt(−1), so that i*i = −1.

There are other ways to represent that number. The pair (a,b) of real numbers may be viewed as a point in the plane, given by X- and Y-coordinates. This same point may also be described by giving the pair of real numbers (r,phi), where r is the distance to the origin O, and phi the angle between the X-axis and the line Oz. Now z = r*exp(i*phi) = r*(cos(phi)+i*sin(phi)).

The basic operations are defined on z = a+b*i and w = c+d*i as:

addition: z+w = (a+c) + (b+d)*i

multiplication: z*w = (a*c − b*d) + (a*d + b*c)*i

division: z/w = ((a*c + b*d)/(c*c + d*d)) + ((b*c − a*d)/(c*c + d*d))*i

Nearly all math function have a complex counterpart but there are some complex-only functions.

EXAMPLES

Your C-compiler can work with complex numbers if it supports the C99 standard. The imaginary unit is represented by I.

```
/* check that exp(i * pi) == −1 */
#include <math.h>        /* for atan */
#include <stdio.h>
#include <complex.h>

int
main(void)
{
    double pi = 4 * atan(1.0);
    double complex z = cexp(I * pi);
    printf("%f + %f * i\n", creal(z), cimag(z));
}
```

SEE ALSO

cabs(3), *cacos*(3), *cacosh*(3), *carg*(3), *casin*(3), *casinh*(3), *catan*(3), *catanh*(3), *ccos*(3), *ccosh*(3), *cerf*(3), *cexp*(3), *cexp2*(3), *cimag*(3), *clog*(3), *clog10*(3), *clog2*(3), *conj*(3), *cpow*(3), *cproj*(3), *creal*(3), *csin*(3), *csinh*(3), *csqrt*(3), *ctan*(3), *ctanh*(3)

NAME

cp1251 − CP 1251 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The Windows Code Pages include several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). CP 1251 encodes the characters used in Cyrillic scripts.

CP 1251 characters

The following table displays the characters in CP 1251 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
200	128	80	Ђ	CYRILLIC CAPITAL LETTER DJE
201	129	81	Ѓ	CYRILLIC CAPITAL LETTER GJE
202	130	82	‚	SINGLE LOW-9 QUOTATION MARK
203	131	83	ѓ	CYRILLIC SMALL LETTER GJE
204	132	84	„	DOUBLE LOW-9 QUOTATION MARK
205	133	85	…	HORIZONTAL ELLIPSIS
206	134	86	†	DAGGER
207	135	87	‡	DOUBLE DAGGER
210	136	88	€	EURO SIGN
211	137	89	‰	PER MILLE SIGN
212	138	8A	Љ	CYRILLIC CAPITAL LETTER LJE
213	139	8B	‹	SINGLE LEFT-POINTING ANGLE QUOTATION MARK
214	140	8C	Њ	CYRILLIC CAPITAL LETTER NJE
215	141	8D	Ќ	CYRILLIC CAPITAL LETTER KJE
216	142	8E	Ћ	CYRILLIC CAPITAL LETTER TSHE
217	143	8F	Џ	CYRILLIC CAPITAL LETTER DZHE
220	144	90	ђ	CYRILLIC SMALL LETTER DJE
221	145	91	‘	LEFT SINGLE QUOTATION MARK
222	146	92	’	RIGHT SINGLE QUOTATION MARK
223	147	93	“	LEFT DOUBLE QUOTATION MARK
224	148	94	”	RIGHT DOUBLE QUOTATION MARK
225	149	95	•	BULLET
226	150	96	–	EN DASH
227	151	97	—	EM DASH
230	152	98		UNDEFINED
231	153	99	™	TRADE MARK SIGN
232	154	9A	љ	CYRILLIC SMALL LETTER LJE
233	155	9B	›	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
234	156	9C	њ	CYRILLIC SMALL LETTER NJE
235	157	9D	ќ	CYRILLIC SMALL LETTER KJE
236	158	9E	ћ	CYRILLIC SMALL LETTER TSHE
237	159	9F	џ	CYRILLIC SMALL LETTER DZHE
240	160	A0		NO-BREAK SPACE
241	161	A1	Ў	CYRILLIC CAPITAL LETTER SHORT U
242	162	A2	ў	CYRILLIC SMALL LETTER SHORT U
243	163	A3	Ј	CYRILLIC CAPITAL LETTER JE
244	164	A4	¤	CURRENCY SIGN
245	165	A5	Ґ	CYRILLIC CAPITAL LETTER GHE WITH UPTURN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	Ё	CYRILLIC CAPITAL LETTER IO
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	Є	CYRILLIC CAPITAL LETTER UKRAINIAN IE
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	Ї	CYRILLIC CAPITAL LETTER YI

260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	І	CYRILLIC CAPITAL LETTER BYELORUSSIAN-UKRAINIAN I
263	179	B3	і	CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I
264	180	B4	ґ	CYRILLIC SMALL LETTER GHE WITH UPTURN
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	ё	CYRILLIC SMALL LETTER IO
271	185	B9	№	NUMERO SIGN
272	186	BA	є	CYRILLIC SMALL LETTER UKRAINIAN IE
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	ј	CYRILLIC SMALL LETTER JE
275	189	BD	Ѕ	CYRILLIC CAPITAL LETTER DZE
276	190	BE	ѕ	CYRILLIC SMALL LETTER DZE
277	191	BF	ї	CYRILLIC SMALL LETTER YI
300	192	C0	А	CYRILLIC CAPITAL LETTER A
301	193	C1	Б	CYRILLIC CAPITAL LETTER BE
302	194	C2	В	CYRILLIC CAPITAL LETTER VE
303	195	C3	Г	CYRILLIC CAPITAL LETTER GHE
304	196	C4	Д	CYRILLIC CAPITAL LETTER DE
305	197	C5	Е	CYRILLIC CAPITAL LETTER IE
306	198	C6	Ж	CYRILLIC CAPITAL LETTER ZHE
307	199	C7	З	CYRILLIC CAPITAL LETTER ZE
310	200	C8	И	CYRILLIC CAPITAL LETTER I
311	201	C9	Й	CYRILLIC CAPITAL LETTER SHORT I
312	202	CA	К	CYRILLIC CAPITAL LETTER KA
313	203	CB	Л	CYRILLIC CAPITAL LETTER EL
314	204	CC	М	CYRILLIC CAPITAL LETTER EM
315	205	CD	Н	CYRILLIC CAPITAL LETTER EN
316	206	CE	О	CYRILLIC CAPITAL LETTER O
317	207	CF	П	CYRILLIC CAPITAL LETTER PE
320	208	D0	Р	CYRILLIC CAPITAL LETTER ER
321	209	D1	С	CYRILLIC CAPITAL LETTER ES
322	210	D2	Т	CYRILLIC CAPITAL LETTER TE
323	211	D3	У	CYRILLIC CAPITAL LETTER U
324	212	D4	Ф	CYRILLIC CAPITAL LETTER EF
325	213	D5	Х	CYRILLIC CAPITAL LETTER HA
326	214	D6	Ц	CYRILLIC CAPITAL LETTER TSE
327	215	D7	Ч	CYRILLIC CAPITAL LETTER CHE
330	216	D8	Ш	CYRILLIC CAPITAL LETTER SHA
331	217	D9	Щ	CYRILLIC CAPITAL LETTER SHCHA
332	218	DA	Ъ	CYRILLIC CAPITAL LETTER HARD SIGN
333	219	DB	Ы	CYRILLIC CAPITAL LETTER YERU
334	220	DC	Ь	CYRILLIC CAPITAL LETTER SOFT SIGN
335	221	DD	Э	CYRILLIC CAPITAL LETTER E
336	222	DE	Ю	CYRILLIC CAPITAL LETTER YU
337	223	DF	Я	CYRILLIC CAPITAL LETTER YA
340	224	E0	а	CYRILLIC SMALL LETTER A
341	225	E1	б	CYRILLIC SMALL LETTER BE
342	226	E2	в	CYRILLIC SMALL LETTER VE
343	227	E3	г	CYRILLIC SMALL LETTER GHE
344	228	E4	д	CYRILLIC SMALL LETTER DE
345	229	E5	е	CYRILLIC SMALL LETTER IE
346	230	E6	ж	CYRILLIC SMALL LETTER ZHE
347	231	E7	з	CYRILLIC SMALL LETTER ZE
350	232	E8	и	CYRILLIC SMALL LETTER I

351	233	E9	й	CYRILLIC SMALL LETTER SHORT I
352	234	EA	к	CYRILLIC SMALL LETTER KA
353	235	EB	л	CYRILLIC SMALL LETTER EL
354	236	EC	м	CYRILLIC SMALL LETTER EM
355	237	ED	н	CYRILLIC SMALL LETTER EN
356	238	EE	о	CYRILLIC SMALL LETTER O
357	239	EF	п	CYRILLIC SMALL LETTER PE
360	240	F0	р	CYRILLIC SMALL LETTER ER
361	241	F1	с	CYRILLIC SMALL LETTER ES
362	242	F2	т	CYRILLIC SMALL LETTER TE
363	243	F3	у	CYRILLIC SMALL LETTER U
364	244	F4	ф	CYRILLIC SMALL LETTER EF
365	245	F5	х	CYRILLIC SMALL LETTER HA
366	246	F6	ц	CYRILLIC SMALL LETTER TSE
367	247	F7	ч	CYRILLIC SMALL LETTER CHE
370	248	F8	ш	CYRILLIC SMALL LETTER SHA
371	249	F9	щ	CYRILLIC SMALL LETTER SHCHA
372	250	FA	ъ	CYRILLIC SMALL LETTER HARD SIGN
373	251	FB	ы	CYRILLIC SMALL LETTER YERU
374	252	FC	ь	CYRILLIC SMALL LETTER SOFT SIGN
375	253	FD	э	CYRILLIC SMALL LETTER E
376	254	FE	ю	CYRILLIC SMALL LETTER YU
377	255	FF	я	CYRILLIC SMALL LETTER YA

NOTES

CP 1251 is also known as Windows Cyrillic.

SEE ALSO

ascii(7), *charsets*(7), *cp1252*(7), *iso_8859-5*(7), *koi8-r*(7), *koi8-u*(7), *utf-8*(7)

NAME

cp1252 − CP 1252 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The Windows Code Pages include several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). CP 1252 encodes the characters used in many West European languages.

CP 1252 characters

The following table displays the characters in CP 1252 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
200	128	80	€	EURO SIGN
202	130	82	‚	SINGLE LOW-9 QUOTATION MARK
203	131	83	*ƒ*	LATIN SMALL LETTER F WITH HOOK
204	132	84	„	DOUBLE LOW-9 QUOTATION MARK
205	133	85	…	HORIZONTAL ELLIPSIS
206	134	86	†	DAGGER
207	135	87	‡	DOUBLE DAGGER
210	136	88	ˆ	MODIFIER LETTER CIRCUMFLEX ACCENT
211	137	89	‰	PER MILLE SIGN
212	138	8A	Š	LATIN CAPITAL LETTER S WITH CARON
213	139	8B	‹	SINGLE LEFT-POINTING ANGLE QUOTATION MARK
214	140	8C	Œ	LATIN CAPITAL LIGATURE OE
216	142	8E	Ž	LATIN CAPITAL LETTER Z WITH CARON
221	145	91	‘	LEFT SINGLE QUOTATION MARK
222	146	92	’	RIGHT SINGLE QUOTATION MARK
223	147	93	“	LEFT DOUBLE QUOTATION MARK
224	148	94	”	RIGHT DOUBLE QUOTATION MARK
225	149	95	•	BULLET
226	150	96	–	EN DASH
227	151	97	—	EM DASH
230	152	98	˜	SMALL TILDE
231	153	99	™	TRADE MARK SIGN
232	154	9A	š	LATIN SMALL LETTER S WITH CARON
233	155	9B	›	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
234	156	9C	œ	LATIN SMALL LIGATURE OE
236	158	9E	ž	LATIN SMALL LETTER Z WITH CARON
237	159	9F	Ÿ	LATIN CAPITAL LETTER Y WITH DIAERESIS
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT

265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF
276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ð	LATIN CAPITAL LETTER ETH
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX

357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN
370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

CP 1252 is also known as Windows-1252.

SEE ALSO

ascii(7), *charsets*(7), *cp1251*(7), *iso_8859-1*(7), *iso_8859-15*(7), *utf-8*(7)

NAME

cpuset − confine processes to processor and memory node subsets

DESCRIPTION

The cpuset filesystem is a pseudo-filesystem interface to the kernel cpuset mechanism, which is used to control the processor placement and memory placement of processes. It is commonly mounted at */dev/cpuset*.

On systems with kernels compiled with built in support for cpusets, all processes are attached to a cpuset, and cpusets are always present. If a system supports cpusets, then it will have the entry **nodev cpuset** in the file */proc/filesystems*. By mounting the cpuset filesystem (see the **EXAMPLES** section below), the administrator can configure the cpusets on a system to control the processor and memory placement of processes on that system. By default, if the cpuset configuration on a system is not modified or if the cpuset filesystem is not even mounted, then the cpuset mechanism, though present, has no effect on the system's behavior.

A cpuset defines a list of CPUs and memory nodes.

The CPUs of a system include all the logical processing units on which a process can execute, including, if present, multiple processor cores within a package and Hyper-Threads within a processor core. Memory nodes include all distinct banks of main memory; small and SMP systems typically have just one memory node that contains all the system's main memory, while NUMA (non-uniform memory access) systems have multiple memory nodes.

Cpusets are represented as directories in a hierarchical pseudo-filesystem, where the top directory in the hierarchy (*/dev/cpuset*) represents the entire system (all online CPUs and memory nodes) and any cpuset that is the child (descendant) of another parent cpuset contains a subset of that parent's CPUs and memory nodes. The directories and files representing cpusets have normal filesystem permissions.

Every process in the system belongs to exactly one cpuset. A process is confined to run only on the CPUs in the cpuset it belongs to, and to allocate memory only on the memory nodes in that cpuset. When a process *fork*(2)s, the child process is placed in the same cpuset as its parent. With sufficient privilege, a process may be moved from one cpuset to another and the allowed CPUs and memory nodes of an existing cpuset may be changed.

When the system begins booting, a single cpuset is defined that includes all CPUs and memory nodes on the system, and all processes are in that cpuset. During the boot process, or later during normal system operation, other cpusets may be created, as subdirectories of this top cpuset, under the control of the system administrator, and processes may be placed in these other cpusets.

Cpusets are integrated with the *sched_setaffinity*(2) scheduling affinity mechanism and the *mbind*(2) and *set_mempolicy*(2) memory-placement mechanisms in the kernel. Neither of these mechanisms let a process make use of a CPU or memory node that is not allowed by that process's cpuset. If changes to a process's cpuset placement conflict with these other mechanisms, then cpuset placement is enforced even if it means overriding these other mechanisms. The kernel accomplishes this overriding by silently restricting the CPUs and memory nodes requested by these other mechanisms to those allowed by the invoking process's cpuset. This can result in these other calls returning an error, if for example, such a call ends up requesting an empty set of CPUs or memory nodes, after that request is restricted to the invoking process's cpuset.

Typically, a cpuset is used to manage the CPU and memory-node confinement for a set of cooperating processes such as a batch scheduler job, and these other mechanisms are used to manage the placement of individual processes or memory regions within that set or job.

FILES

Each directory below */dev/cpuset* represents a cpuset and contains a fixed set of pseudo-files describing the state of that cpuset.

New cpusets are created using the *mkdir*(2) system call or the *mkdir*(1) command. The properties of a cpuset, such as its flags, allowed CPUs and memory nodes, and attached processes, are queried and modified by reading or writing to the appropriate file in that cpuset's directory, as listed below.

The pseudo-files in each cpuset directory are automatically created when the cpuset is created, as a result of the *mkdir*(2) invocation. It is not possible to directly add or remove these pseudo-files.

A cpuset directory that contains no child cpuset directories, and has no attached processes, can be

removed using *rmdir*(2) or *rmdir*(1)It is not necessary, or possible, to remove the pseudo-files inside the directory before removing it.

The pseudo-files in each cpuset directory are small text files that may be read and written using traditional shell utilities such as *cat*(1), and *echo*(1), or from a program by using file I/O library functions or system calls, such as *open*(2), *read*(2), *write*(2), and *close*(2).

The pseudo-files in a cpuset directory represent internal kernel state and do not have any persistent image on disk. Each of these per-cpuset files is listed and described below.

tasks List of the process IDs (PIDs) of the processes in that cpuset. The list is formatted as a series of ASCII decimal numbers, each followed by a newline. A process may be added to a cpuset (automatically removing it from the cpuset that previously contained it) by writing its PID to that cpuset's *tasks* file (with or without a trailing newline).

Warning: only one PID may be written to the *tasks* file at a time. If a string is written that contains more than one PID, only the first one will be used.

notify_on_release
Flag (0 or 1). If set (1), that cpuset will receive special handling after it is released, that is, after all processes cease using it (i.e., terminate or are moved to a different cpuset) and all child cpuset directories have been removed. See the **Notify On Release** section, below.

cpuset.cpus
List of the physical numbers of the CPUs on which processes in that cpuset are allowed to execute. See **List Format** below for a description of the format of *cpus*.

The CPUs allowed to a cpuset may be changed by writing a new list to its *cpus* file.

cpuset.cpu_exclusive
Flag (0 or 1). If set (1), the cpuset has exclusive use of its CPUs (no sibling or cousin cpuset may overlap CPUs). By default, this is off (0). Newly created cpusets also initially default this to off (0).

Two cpusets are *sibling* cpusets if they share the same parent cpuset in the */dev/cpuset* hierarchy. Two cpusets are *cousin* cpusets if neither is the ancestor of the other. Regardless of the *cpu_exclusive* setting, if one cpuset is the ancestor of another, and if both of these cpusets have nonempty *cpus*, then their *cpus* must overlap, because the *cpus* of any cpuset are always a subset of the *cpus* of its parent cpuset.

cpuset.mems
List of memory nodes on which processes in this cpuset are allowed to allocate memory. See **List Format** below for a description of the format of *mems*.

cpuset.mem_exclusive
Flag (0 or 1). If set (1), the cpuset has exclusive use of its memory nodes (no sibling or cousin may overlap). Also if set (1), the cpuset is a **Hardwall** cpuset (see below). By default, this is off (0). Newly created cpusets also initially default this to off (0).

Regardless of the *mem_exclusive* setting, if one cpuset is the ancestor of another, then their memory nodes must overlap, because the memory nodes of any cpuset are always a subset of the memory nodes of that cpuset's parent cpuset.

cpuset.mem_hardwall (since Linux 2.6.26)
Flag (0 or 1). If set (1), the cpuset is a **Hardwall** cpuset (see below). Unlike **mem_exclusive**, there is no constraint on whether cpusets marked **mem_hardwall** may have overlapping memory nodes with sibling or cousin cpusets. By default, this is off (0). Newly created cpusets also initially default this to off (0).

cpuset.memory_migrate (since Linux 2.6.16)
Flag (0 or 1). If set (1), then memory migration is enabled. By default, this is off (0). See the **Memory Migration** section, below.

cpuset.memory_pressure (since Linux 2.6.16)
A measure of how much memory pressure the processes in this cpuset are causing. See the **Memory Pressure** section, below. Unless *memory_pressure_enabled* is enabled, always has value zero (0). This file is read-only. See the **WARNINGS** section, below.

cpuset.memory_pressure_enabled (since Linux 2.6.16)
Flag (0 or 1). This file is present only in the root cpuset, normally */dev/cpuset*. If set (1), the *memory_pressure* calculations are enabled for all cpusets in the system. By default, this is off (0). See the **Memory Pressure** section, below.

cpuset.memory_spread_page (since Linux 2.6.17)
Flag (0 or 1). If set (1), pages in the kernel page cache (filesystem buffers) are uniformly spread across the cpuset. By default, this is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the **Memory Spread** section, below.

cpuset.memory_spread_slab (since Linux 2.6.17)
Flag (0 or 1). If set (1), the kernel slab caches for file I/O (directory and inode structures) are uniformly spread across the cpuset. By default, is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the **Memory Spread** section, below.

cpuset.sched_load_balance (since Linux 2.6.24)
Flag (0 or 1). If set (1, the default) the kernel will automatically load balance processes in that cpuset over the allowed CPUs in that cpuset. If cleared (0) the kernel will avoid load balancing processes in this cpuset, *unless* some other cpuset with overlapping CPUs has its *sched_load_balance* flag set. See **Scheduler Load Balancing**, below, for further details.

cpuset.sched_relax_domain_level (since Linux 2.6.26)
Integer, between −1 and a small positive value. The *sched_relax_domain_level* controls the width of the range of CPUs over which the kernel scheduler performs immediate rebalancing of runnable tasks across CPUs. If *sched_load_balance* is disabled, then the setting of *sched_relax_domain_level* does not matter, as no such load balancing is done. If *sched_load_balance* is enabled, then the higher the value of the *sched_relax_domain_level*, the wider the range of CPUs over which immediate load balancing is attempted. See **Scheduler Relax Domain Level**, below, for further details.

In addition to the above pseudo-files in each directory below */dev/cpuset*, each process has a pseudo-file, */proc/*pid*/cpuset*, that displays the path of the process's cpuset directory relative to the root of the cpuset filesystem.

Also the */proc/*pid*/status* file for each process has four added lines, displaying the process's *Cpus_allowed* (on which CPUs it may be scheduled) and *Mems_allowed* (on which memory nodes it may obtain memory), in the two formats **Mask Format** and **List Format** (see below) as shown in the following example:

```
Cpus_allowed:   ffffffff,ffffffff,ffffffff,ffffffff
Cpus_allowed_list:     0-127
Mems_allowed:   ffffffff,ffffffff
Mems_allowed_list:     0-63
```

The "allowed" fields were added in Linux 2.6.24; the "allowed_list" fields were added in Linux 2.6.26.

EXTENDED CAPABILITIES

In addition to controlling which *cpus* and *mems* a process is allowed to use, cpusets provide the following extended capabilities.

Exclusive cpusets

If a cpuset is marked *cpu_exclusive* or *mem_exclusive*, no other cpuset, other than a direct ancestor or descendant, may share any of the same CPUs or memory nodes.

A cpuset that is *mem_exclusive* restricts kernel allocations for buffer cache pages and other internal kernel data pages commonly shared by the kernel across multiple users. All cpusets, whether *mem_exclusive* or not, restrict allocations of memory for user space. This enables configuring a system so that several independent jobs can share common kernel data, while isolating each job's user allocation in its own cpuset. To do this, construct a large *mem_exclusive* cpuset to hold all the jobs, and construct child, non-*mem_exclusive* cpusets for each individual job. Only a small amount of kernel memory, such as requests from interrupt handlers, is allowed to be placed on memory nodes outside even a *mem_exclusive* cpuset.

Hardwall

A cpuset that has *mem_exclusive* or *mem_hardwall* set is a *hardwall* cpuset. A *hardwall* cpuset restricts kernel allocations for page, buffer, and other data commonly shared by the kernel across multiple

users. All cpusets, whether *hardwall* or not, restrict allocations of memory for user space.

This enables configuring a system so that several independent jobs can share common kernel data, such as filesystem pages, while isolating each job's user allocation in its own cpuset. To do this, construct a large *hardwall* cpuset to hold all the jobs, and construct child cpusets for each individual job which are not *hardwall* cpusets.

Only a small amount of kernel memory, such as requests from interrupt handlers, is allowed to be taken outside even a *hardwall* cpuset.

Notify on release

If the *notify_on_release* flag is enabled (1) in a cpuset, then whenever the last process in the cpuset leaves (exits or attaches to some other cpuset) and the last child cpuset of that cpuset is removed, the kernel will run the command */sbin/cpuset_release_agent*, supplying the pathname (relative to the mount point of the cpuset filesystem) of the abandoned cpuset. This enables automatic removal of abandoned cpusets.

The default value of *notify_on_release* in the root cpuset at system boot is disabled (0). The default value of other cpusets at creation is the current value of their parent's *notify_on_release* setting.

The command */sbin/cpuset_release_agent* is invoked, with the name (*/dev/cpuset* relative path) of the to-be-released cpuset in *argv[1]*.

The usual contents of the command */sbin/cpuset_release_agent* is simply the shell script:

```
#!/bin/sh
rmdir /dev/cpuset/$1
```

As with other flag values below, this flag can be changed by writing an ASCII number 0 or 1 (with optional trailing newline) into the file, to clear or set the flag, respectively.

Memory pressure

The *memory_pressure* of a cpuset provides a simple per-cpuset running average of the rate that the processes in a cpuset are attempting to free up in-use memory on the nodes of the cpuset to satisfy additional memory requests.

This enables batch managers that are monitoring jobs running in dedicated cpusets to efficiently detect what level of memory pressure that job is causing.

This is useful both on tightly managed systems running a wide mix of submitted jobs, which may choose to terminate or reprioritize jobs that are trying to use more memory than allowed on the nodes assigned them, and with tightly coupled, long-running, massively parallel scientific computing jobs that will dramatically fail to meet required performance goals if they start to use more memory than allowed to them.

This mechanism provides a very economical way for the batch manager to monitor a cpuset for signs of memory pressure. It's up to the batch manager or other user code to decide what action to take if it detects signs of memory pressure.

Unless memory pressure calculation is enabled by setting the pseudo-file */dev/cpuset/cpuset.memory_pressure_enabled*, it is not computed for any cpuset, and reads from any *memory_pressure* always return zero, as represented by the ASCII string "0\n". See the **WARNINGS** section, below.

A per-cpuset, running average is employed for the following reasons:

- Because this meter is per-cpuset rather than per-process or per virtual memory region, the system load imposed by a batch scheduler monitoring this metric is sharply reduced on large systems, because a scan of the tasklist can be avoided on each set of queries.
- Because this meter is a running average rather than an accumulating counter, a batch scheduler can detect memory pressure with a single read, instead of having to read and accumulate results for a period of time.
- Because this meter is per-cpuset rather than per-process, the batch scheduler can obtain the key information—memory pressure in a cpuset—with a single read, rather than having to query and accumulate results over all the (dynamically changing) set of processes in the cpuset.

The *memory_pressure* of a cpuset is calculated using a per-cpuset simple digital filter that is kept within the kernel. For each cpuset, this filter tracks the recent rate at which processes attached to that

cpuset enter the kernel direct reclaim code.

The kernel direct reclaim code is entered whenever a process has to satisfy a memory page request by first finding some other page to repurpose, due to lack of any readily available already free pages. Dirty filesystem pages are repurposed by first writing them to disk. Unmodified filesystem buffer pages are repurposed by simply dropping them, though if that page is needed again, it will have to be reread from disk.

The *cpuset.memory_pressure* file provides an integer number representing the recent (half-life of 10 seconds) rate of entries to the direct reclaim code caused by any process in the cpuset, in units of reclaims attempted per second, times 1000.

Memory spread

There are two Boolean flag files per cpuset that control where the kernel allocates pages for the filesystem buffers and related in-kernel data structures. They are called *cpuset.memory_spread_page* and *cpuset.memory_spread_slab*.

If the per-cpuset Boolean flag file *cpuset.memory_spread_page* is set, then the kernel will spread the filesystem buffers (page cache) evenly over all the nodes that the faulting process is allowed to use, instead of preferring to put those pages on the node where the process is running.

If the per-cpuset Boolean flag file *cpuset.memory_spread_slab* is set, then the kernel will spread some filesystem-related slab caches, such as those for inodes and directory entries, evenly over all the nodes that the faulting process is allowed to use, instead of preferring to put those pages on the node where the process is running.

The setting of these flags does not affect the data segment (see *brk*(2)) or stack segment pages of a process.

By default, both kinds of memory spreading are off and the kernel prefers to allocate memory pages on the node local to where the requesting process is running. If that node is not allowed by the process's NUMA memory policy or cpuset configuration or if there are insufficient free memory pages on that node, then the kernel looks for the nearest node that is allowed and has sufficient free memory.

When new cpusets are created, they inherit the memory spread settings of their parent.

Setting memory spreading causes allocations for the affected page or slab caches to ignore the process's NUMA memory policy and be spread instead. However, the effect of these changes in memory placement caused by cpuset-specified memory spreading is hidden from the *mbind*(2) or *set_mempolicy*(2) calls. These two NUMA memory policy calls always appear to behave as if no cpuset-specified memory spreading is in effect, even if it is. If cpuset memory spreading is subsequently turned off, the NUMA memory policy most recently specified by these calls is automatically reapplied.

Both *cpuset.memory_spread_page* and *cpuset.memory_spread_slab* are Boolean flag files. By default, they contain "0", meaning that the feature is off for that cpuset. If a "1" is written to that file, that turns the named feature on.

Cpuset-specified memory spreading behaves similarly to what is known (in other contexts) as round-robin or interleave memory placement.

Cpuset-specified memory spreading can provide substantial performance improvements for jobs that:

- need to place thread-local data on memory nodes close to the CPUs which are running the threads that most frequently access that data; but also
- need to access large filesystem data sets that must to be spread across the several nodes in the job's cpuset in order to fit.

Without this policy, the memory allocation across the nodes in the job's cpuset can become very uneven, especially for jobs that might have just a single thread initializing or reading in the data set.

Memory migration

Normally, under the default setting (disabled) of *cpuset.memory_migrate*, once a page is allocated (given a physical page of main memory), then that page stays on whatever node it was allocated, so long as it remains allocated, even if the cpuset's memory-placement policy *mems* subsequently changes.

When memory migration is enabled in a cpuset, if the *mems* setting of the cpuset is changed, then any memory page in use by any process in the cpuset that is on a memory node that is no longer allowed

will be migrated to a memory node that is allowed.

Furthermore, if a process is moved into a cpuset with *memory_migrate* enabled, any memory pages it uses that were on memory nodes allowed in its previous cpuset, but which are not allowed in its new cpuset, will be migrated to a memory node allowed in the new cpuset.

The relative placement of a migrated page within the cpuset is preserved during these migration operations if possible. For example, if the page was on the second valid node of the prior cpuset, then the page will be placed on the second valid node of the new cpuset, if possible.

Scheduler load balancing

The kernel scheduler automatically load balances processes. If one CPU is underutilized, the kernel will look for processes on other more overloaded CPUs and move those processes to the underutilized CPU, within the constraints of such placement mechanisms as cpusets and *sched_setaffinity*(2).

The algorithmic cost of load balancing and its impact on key shared kernel data structures such as the process list increases more than linearly with the number of CPUs being balanced. For example, it costs more to load balance across one large set of CPUs than it does to balance across two smaller sets of CPUs, each of half the size of the larger set. (The precise relationship between the number of CPUs being balanced and the cost of load balancing depends on implementation details of the kernel process scheduler, which is subject to change over time, as improved kernel scheduler algorithms are implemented.)

The per-cpuset flag *sched_load_balance* provides a mechanism to suppress this automatic scheduler load balancing in cases where it is not needed and suppressing it would have worthwhile performance benefits.

By default, load balancing is done across all CPUs, except those marked isolated using the kernel boot time "isolcpus=" argument. (See **Scheduler Relax Domain Level**, below, to change this default.)

This default load balancing across all CPUs is not well suited to the following two situations:

- On large systems, load balancing across many CPUs is expensive. If the system is managed using cpusets to place independent jobs on separate sets of CPUs, full load balancing is unnecessary.
- Systems supporting real-time on some CPUs need to minimize system overhead on those CPUs, including avoiding process load balancing if that is not needed.

When the per-cpuset flag *sched_load_balance* is enabled (the default setting), it requests load balancing across all the CPUs in that cpuset's allowed CPUs, ensuring that load balancing can move a process (not otherwise pinned, as by *sched_setaffinity*(2)) from any CPU in that cpuset to any other.

When the per-cpuset flag *sched_load_balance* is disabled, then the scheduler will avoid load balancing across the CPUs in that cpuset, *except* in so far as is necessary because some overlapping cpuset has *sched_load_balance* enabled.

So, for example, if the top cpuset has the flag *sched_load_balance* enabled, then the scheduler will load balance across all CPUs, and the setting of the *sched_load_balance* flag in other cpusets has no effect, as we're already fully load balancing.

Therefore in the above two situations, the flag *sched_load_balance* should be disabled in the top cpuset, and only some of the smaller, child cpusets would have this flag enabled.

When doing this, you don't usually want to leave any unpinned processes in the top cpuset that might use nontrivial amounts of CPU, as such processes may be artificially constrained to some subset of CPUs, depending on the particulars of this flag setting in descendant cpusets. Even if such a process could use spare CPU cycles in some other CPUs, the kernel scheduler might not consider the possibility of load balancing that process to the underused CPU.

Of course, processes pinned to a particular CPU can be left in a cpuset that disables *sched_load_balance* as those processes aren't going anywhere else anyway.

Scheduler relax domain level

The kernel scheduler performs immediate load balancing whenever a CPU becomes free or another task becomes runnable. This load balancing works to ensure that as many CPUs as possible are usefully employed running tasks. The kernel also performs periodic load balancing off the software clock described in *time*(7). The setting of *sched_relax_domain_level* applies only to immediate load balancing. Regardless of the *sched_relax_domain_level* setting, periodic load balancing is attempted over all

CPUs (unless disabled by turning off *sched_load_balance*.) In any case, of course, tasks will be scheduled to run only on CPUs allowed by their cpuset, as modified by *sched_setaffinity*(2) system calls.

On small systems, such as those with just a few CPUs, immediate load balancing is useful to improve system interactivity and to minimize wasteful idle CPU cycles. But on large systems, attempting immediate load balancing across a large number of CPUs can be more costly than it is worth, depending on the particular performance characteristics of the job mix and the hardware.

The exact meaning of the small integer values of *sched_relax_domain_level* will depend on internal implementation details of the kernel scheduler code and on the non-uniform architecture of the hardware. Both of these will evolve over time and vary by system architecture and kernel version.

As of this writing, when this capability was introduced in Linux 2.6.26, on certain popular architectures, the positive values of *sched_relax_domain_level* have the following meanings.

1 Perform immediate load balancing across Hyper-Thread siblings on the same core.
2 Perform immediate load balancing across other cores in the same package.
3 Perform immediate load balancing across other CPUs on the same node or blade.
4 Perform immediate load balancing across over several (implementation detail) nodes [On NUMA systems].
5 Perform immediate load balancing across over all CPUs in system [On NUMA systems].

The *sched_relax_domain_level* value of zero (0) always means don't perform immediate load balancing, hence that load balancing is done only periodically, not immediately when a CPU becomes available or another task becomes runnable.

The *sched_relax_domain_level* value of minus one (−1) always means use the system default value. The system default value can vary by architecture and kernel version. This system default value can be changed by kernel boot-time "relax_domain_level=" argument.

In the case of multiple overlapping cpusets which have conflicting *sched_relax_domain_level* values, then the highest such value applies to all CPUs in any of the overlapping cpusets. In such cases, **−1** is the lowest value, overridden by any other value, and **0** is the next lowest value.

FORMATS

The following formats are used to represent sets of CPUs and memory nodes.

Mask format

The **Mask Format** is used to represent CPU and memory-node bit masks in the */proc/*pid*/status* file.

This format displays each 32-bit word in hexadecimal (using ASCII characters "0" - "9" and "a" - "f"); words are filled with leading zeros, if required. For masks longer than one word, a comma separator is used between words. Words are displayed in big-endian order, which has the most significant bit first. The hex digits within a word are also in big-endian order.

The number of 32-bit words displayed is the minimum number needed to display all bits of the bit mask, based on the size of the bit mask.

Examples of the **Mask Format**:

```
00000001                        # just bit 0 set
40000000,00000000,00000000      # just bit 94 set
00000001,00000000,00000000      # just bit 64 set
000000ff,00000000               # bits 32-39 set
00000000,000e3862               # 1,5,6,11-13,17-19 set
```

A mask with bits 0, 1, 2, 4, 8, 16, 32, and 64 set displays as:

```
00000001,00000001,00010117
```

The first "1" is for bit 64, the second for bit 32, the third for bit 16, the fourth for bit 8, the fifth for bit 4, and the "7" is for bits 2, 1, and 0.

List format

The **List Format** for *cpus* and *mems* is a comma-separated list of CPU or memory-node numbers and ranges of numbers, in ASCII decimal.

Examples of the **List Format**:

```
0-4,9           # bits 0, 1, 2, 3, 4, and 9 set
```

```
0-2,7,12-14     # bits 0, 1, 2, 7, 12, 13, and 14 set
```

RULES

The following rules apply to each cpuset:

- Its CPUs and memory nodes must be a (possibly equal) subset of its parent's.
- It can be marked *cpu_exclusive* only if its parent is.
- It can be marked *mem_exclusive* only if its parent is.
- If it is *cpu_exclusive*, its CPUs may not overlap any sibling.
- If it is *mem_exclusive*, its memory nodes may not overlap any sibling.

PERMISSIONS

The permissions of a cpuset are determined by the permissions of the directories and pseudo-files in the cpuset filesystem, normally mounted at */dev/cpuset*.

For instance, a process can put itself in some other cpuset (than its current one) if it can write the *tasks* file for that cpuset. This requires execute permission on the encompassing directories and write permission on the *tasks* file.

An additional constraint is applied to requests to place some other process in a cpuset. One process may not attach another to a cpuset unless it would have permission to send that process a signal (see *kill*(2)).

A process may create a child cpuset if it can access and write the parent cpuset directory. It can modify the CPUs or memory nodes in a cpuset if it can access that cpuset's directory (execute permissions on the each of the parent directories) and write the corresponding *cpus* or *mems* file.

There is one minor difference between the manner in which these permissions are evaluated and the manner in which normal filesystem operation permissions are evaluated. The kernel interprets relative pathnames starting at a process's current working directory. Even if one is operating on a cpuset file, relative pathnames are interpreted relative to the process's current working directory, not relative to the process's current cpuset. The only ways that cpuset paths relative to a process's current cpuset can be used are if either the process's current working directory is its cpuset (it first did a **cd** or *chdir*(2) to its cpuset directory beneath */dev/cpuset*, which is a bit unusual) or if some user code converts the relative cpuset path to a full filesystem path.

In theory, this means that user code should specify cpusets using absolute pathnames, which requires knowing the mount point of the cpuset filesystem (usually, but not necessarily, */dev/cpuset*). In practice, all user level code that this author is aware of simply assumes that if the cpuset filesystem is mounted, then it is mounted at */dev/cpuset*. Furthermore, it is common practice for carefully written user code to verify the presence of the pseudo-file */dev/cpuset/tasks* in order to verify that the cpuset pseudo-filesystem is currently mounted.

WARNINGS

Enabling memory_pressure

By default, the per-cpuset file *cpuset.memory_pressure* always contains zero (0). Unless this feature is enabled by writing "1" to the pseudo-file */dev/cpuset/cpuset.memory_pressure_enabled*, the kernel does not compute per-cpuset *memory_pressure*.

Using the echo command

When using the **echo** command at the shell prompt to change the values of cpuset files, beware that the built-in **echo** command in some shells does not display an error message if the *write*(2) system call fails. For example, if the command:

```
echo 19 > cpuset.mems
```

failed because memory node 19 was not allowed (perhaps the current system does not have a memory node 19), then the **echo** command might not display any error. It is better to use the **/bin/echo** external command to change cpuset file settings, as this command will display *write*(2) errors, as in the example:

```
/bin/echo 19 > cpuset.mems
/bin/echo: write error: Invalid argument
```

EXCEPTIONS

Memory placement

Not all allocations of system memory are constrained by cpusets, for the following reasons.

If hot-plug functionality is used to remove all the CPUs that are currently assigned to a cpuset, then the kernel will automatically update the *cpus_allowed* of all processes attached to CPUs in that cpuset to allow all CPUs. When memory hot-plug functionality for removing memory nodes is available, a similar exception is expected to apply there as well. In general, the kernel prefers to violate cpuset placement, rather than starving a process that has had all its allowed CPUs or memory nodes taken offline. User code should reconfigure cpusets to refer only to online CPUs and memory nodes when using hot-plug to add or remove such resources.

A few kernel-critical, internal memory-allocation requests, marked GFP_ATOMIC, must be satisfied immediately. The kernel may drop some request or malfunction if one of these allocations fail. If such a request cannot be satisfied within the current process's cpuset, then we relax the cpuset, and look for memory anywhere we can find it. It's better to violate the cpuset than stress the kernel.

Allocations of memory requested by kernel drivers while processing an interrupt lack any relevant process context, and are not confined by cpusets.

Renaming cpusets

You can use the *rename*(2) system call to rename cpusets. Only simple renaming is supported; that is, changing the name of a cpuset directory is permitted, but moving a directory into a different directory is not permitted.

ERRORS

The Linux kernel implementation of cpusets sets *errno* to specify the reason for a failed system call affecting cpusets.

The possible *errno* settings and their meaning when set on a failed cpuset call are as listed below.

E2BIG Attempted a *write*(2) on a special cpuset file with a length larger than some kernel-determined upper limit on the length of such writes.

EACCES
Attempted to *write*(2) the process ID (PID) of a process to a cpuset *tasks* file when one lacks permission to move that process.

EACCES
Attempted to add, using *write*(2), a CPU or memory node to a cpuset, when that CPU or memory node was not already in its parent.

EACCES
Attempted to set, using *write*(2), *cpuset.cpu_exclusive* or *cpuset.mem_exclusive* on a cpuset whose parent lacks the same setting.

EACCES
Attempted to *write*(2) a *cpuset.memory_pressure* file.

EACCES
Attempted to create a file in a cpuset directory.

EBUSY
Attempted to remove, using *rmdir*(2), a cpuset with attached processes.

EBUSY
Attempted to remove, using *rmdir*(2), a cpuset with child cpusets.

EBUSY
Attempted to remove a CPU or memory node from a cpuset that is also in a child of that cpuset.

EEXIST
Attempted to create, using *mkdir*(2), a cpuset that already exists.

EEXIST
Attempted to *rename*(2) a cpuset to a name that already exists.

EFAULT
Attempted to *read*(2) or *write*(2) a cpuset file using a buffer that is outside the writing processes accessible address space.

EINVAL
Attempted to change a cpuset, using *write*(2), in a way that would violate a *cpu_exclusive* or *mem_exclusive* attribute of that cpuset or any of its siblings.

EINVAL
Attempted to *write*(2) an empty *cpuset.cpus* or *cpuset.mems* list to a cpuset which has attached processes or child cpusets.

EINVAL
Attempted to *write*(2) a *cpuset.cpus* or *cpuset.mems* list which included a range with the second number smaller than the first number.

EINVAL
Attempted to *write*(2) a *cpuset.cpus* or *cpuset.mems* list which included an invalid character in the string.

EINVAL
Attempted to *write*(2) a list to a *cpuset.cpus* file that did not include any online CPUs.

EINVAL
Attempted to *write*(2) a list to a *cpuset.mems* file that did not include any online memory nodes.

EINVAL
Attempted to *write*(2) a list to a *cpuset.mems* file that included a node that held no memory.

EIO Attempted to *write*(2) a string to a cpuset *tasks* file that does not begin with an ASCII decimal integer.

EIO Attempted to *rename*(2) a cpuset into a different directory.

ENAMETOOLONG
Attempted to *read*(2) a */proc/*pid*/cpuset* file for a cpuset path that is longer than the kernel page size.

ENAMETOOLONG
Attempted to create, using *mkdir*(2), a cpuset whose base directory name is longer than 255 characters.

ENAMETOOLONG
Attempted to create, using *mkdir*(2), a cpuset whose full pathname, including the mount point (typically "/dev/cpuset/") prefix, is longer than 4095 characters.

ENODEV
The cpuset was removed by another process at the same time as a *write*(2) was attempted on one of the pseudo-files in the cpuset directory.

ENOENT
Attempted to create, using *mkdir*(2), a cpuset in a parent cpuset that doesn't exist.

ENOENT
Attempted to *access*(2) or *open*(2) a nonexistent file in a cpuset directory.

ENOMEM
Insufficient memory is available within the kernel; can occur on a variety of system calls affecting cpusets, but only if the system is extremely short of memory.

ENOSPC
Attempted to *write*(2) the process ID (PID) of a process to a cpuset *tasks* file when the cpuset had an empty *cpuset.cpus* or empty *cpuset.mems* setting.

ENOSPC
Attempted to *write*(2) an empty *cpuset.cpus* or *cpuset.mems* setting to a cpuset that has tasks attached.

ENOTDIR
Attempted to *rename*(2) a nonexistent cpuset.

EPERM
Attempted to remove a file from a cpuset directory.

ERANGE
Specified a *cpuset.cpus* or *cpuset.mems* list to the kernel which included a number too large for the kernel to set in its bit masks.

ESRCH
Attempted to *write*(2) the process ID (PID) of a nonexistent process to a cpuset *tasks* file.

VERSIONS

Cpusets appeared in Linux 2.6.12.

NOTES

Despite its name, the *pid* parameter is actually a thread ID, and each thread in a threaded group can be attached to a different cpuset. The value returned from a call to *gettid*(2) can be passed in the argument *pid*.

BUGS

cpuset.memory_pressure cpuset files can be opened for writing, creation, or truncation, but then the *write*(2) fails with *errno* set to **EACCES**, and the creation and truncation options on *open*(2) have no effect.

EXAMPLES

The following examples demonstrate querying and setting cpuset options using shell commands.

Creating and attaching to a cpuset.

To create a new cpuset and attach the current command shell to it, the steps are:

(1) mkdir /dev/cpuset (if not already done)
(2) mount −t cpuset none /dev/cpuset (if not already done)
(3) Create the new cpuset using *mkdir*(1)
(4) Assign CPUs and memory nodes to the new cpuset.
(5) Attach the shell to the new cpuset.

For example, the following sequence of commands will set up a cpuset named "Charlie", containing just CPUs 2 and 3, and memory node 1, and then attach the current shell to that cpuset.

```
$ mkdir /dev/cpuset
$ mount -t cpuset cpuset /dev/cpuset
$ cd /dev/cpuset
$ mkdir Charlie
$ cd Charlie
$ /bin/echo 2-3 > cpuset.cpus
$ /bin/echo 1 > cpuset.mems
$ /bin/echo $$ > tasks
# The current shell is now running in cpuset Charlie
# The next line should display '/Charlie'
$ cat /proc/self/cpuset
```

Migrating a job to different memory nodes.

To migrate a job (the set of processes attached to a cpuset) to different CPUs and memory nodes in the system, including moving the memory pages currently allocated to that job, perform the following steps.

(1) Let's say we want to move the job in cpuset *alpha* (CPUs 4–7 and memory nodes 2–3) to a new cpuset *beta* (CPUs 16–19 and memory nodes 8–9).
(2) First create the new cpuset *beta*.
(3) Then allow CPUs 16–19 and memory nodes 8–9 in *beta*.
(4) Then enable *memory_migration* in *beta*.
(5) Then move each process from *alpha* to *beta*.

The following sequence of commands accomplishes this.

```
$ cd /dev/cpuset
```

```
$ mkdir beta
$ cd beta
$ /bin/echo 16-19 > cpuset.cpus
$ /bin/echo 8-9 > cpuset.mems
$ /bin/echo 1 > cpuset.memory_migrate
$ while read i; do /bin/echo $i; done < ../alpha/tasks > tasks
```

The above should move any processes in *alpha* to *beta*, and any memory held by these processes on memory nodes 2–3 to memory nodes 8–9, respectively.

Notice that the last step of the above sequence did not do:

```
$ cp ../alpha/tasks tasks
```

The *while* loop, rather than the seemingly easier use of the *cp*(1) command, was necessary because only one process PID at a time may be written to the *tasks* file.

The same effect (writing one PID at a time) as the *while* loop can be accomplished more efficiently, in fewer keystrokes and in syntax that works on any shell, but alas more obscurely, by using the **-u** (unbuffered) option of *sed*(1):

```
$ sed -un p < ../alpha/tasks > tasks
```

SEE ALSO

taskset(1), *get_mempolicy*(2), *getcpu*(2), *mbind*(2), *sched_getaffinity*(2), *sched_setaffinity*(2), *sched_setscheduler*(2), *set_mempolicy*(2), *CPU_SET*(3), *proc*(5), *cgroups*(7), *numa*(7), *sched*(7), *migratepages*(8), *numactl*(8)

Documentation/admin-guide/cgroup-v1/cpusets.rst in the Linux kernel source tree (or *Documentation/cgroup-v1/cpusets.txt* before Linux 4.18, and *Documentation/cpusets.txt* before Linux 2.6.29)

NAME

credentials − process identifiers

DESCRIPTION

Process ID (PID)

Each process has a unique nonnegative integer identifier that is assigned when the process is created using *fork*(2). A process can obtain its PID using *getpid*(2). A PID is represented using the type *pid_t* (defined in <*sys/types.h*>).

PIDs are used in a range of system calls to identify the process affected by the call, for example: *kill*(2), *ptrace*(2), *setpriority*(2), *setpgid*(2), *setsid*(2), *sigqueue*(3), and *waitpid*(2).

A process's PID is preserved across an *execve*(2).

Parent process ID (PPID)

A process's parent process ID identifies the process that created this process using *fork*(2). A process can obtain its PPID using *getppid*(2). A PPID is represented using the type *pid_t*.

A process's PPID is preserved across an *execve*(2).

Process group ID and session ID

Each process has a session ID and a process group ID, both represented using the type *pid_t*. A process can obtain its session ID using *getsid*(2), and its process group ID using *getpgrp*(2).

A child created by *fork*(2) inherits its parent's session ID and process group ID. A process's session ID and process group ID are preserved across an *execve*(2).

Sessions and process groups are abstractions devised to support shell job control. A process group (sometimes called a "job") is a collection of processes that share the same process group ID; the shell creates a new process group for the process(es) used to execute single command or pipeline (e.g., the two processes created to execute the command "ls | wc" are placed in the same process group). A process's group membership can be set using *setpgid*(2). The process whose process ID is the same as its process group ID is the *process group leader* for that group.

A session is a collection of processes that share the same session ID. All of the members of a process group also have the same session ID (i.e., all of the members of a process group always belong to the same session, so that sessions and process groups form a strict two-level hierarchy of processes.) A new session is created when a process calls *setsid*(2), which creates a new session whose session ID is the same as the PID of the process that called *setsid*(2). The creator of the session is called the *session leader*.

All of the processes in a session share a *controlling terminal*. The controlling terminal is established when the session leader first opens a terminal (unless the **O_NOCTTY** flag is specified when calling *open*(2)). A terminal may be the controlling terminal of at most one session.

At most one of the jobs in a session may be the *foreground job*; other jobs in the session are *background jobs*. Only the foreground job may read from the terminal; when a process in the background attempts to read from the terminal, its process group is sent a **SIGTTIN** signal, which suspends the job. If the **TOSTOP** flag has been set for the terminal (see *termios*(3)), then only the foreground job may write to the terminal; writes from background jobs cause a **SIGTTOU** signal to be generated, which suspends the job. When terminal keys that generate a signal (such as the *interrupt* key, normally control-C) are pressed, the signal is sent to the processes in the foreground job.

Various system calls and library functions may operate on all members of a process group, including *kill*(2), *killpg*(3), *getpriority*(2), *setpriority*(2), *ioprio_get*(2), *ioprio_set*(2), *waitid*(2), and *waitpid*(2). See also the discussion of the **F_GETOWN**, **F_GETOWN_EX**, **F_SETOWN**, and **F_SETOWN_EX** operations in *fcntl*(2).

User and group identifiers

Each process has various associated user and group IDs. These IDs are integers, respectively represented using the types *uid_t* and *gid_t* (defined in <*sys/types.h*>).

On Linux, each process has the following user and group identifiers:

- Real user ID and real group ID. These IDs determine who owns the process. A process can obtain its real user (group) ID using *getuid*(2) (**getgid**(2)).

- Effective user ID and effective group ID. These IDs are used by the kernel to determine the permissions that the process will have when accessing shared resources such as message queues, shared memory, and semaphores. On most UNIX systems, these IDs also determine the permissions when accessing files. However, Linux uses the filesystem IDs described below for this task. A process can obtain its effective user (group) ID using *geteuid*(2) (**getegid**(2)).
- Saved set-user-ID and saved set-group-ID. These IDs are used in set-user-ID and set-group-ID programs to save a copy of the corresponding effective IDs that were set when the program was executed (see *execve*(2)). A set-user-ID program can assume and drop privileges by switching its effective user ID back and forth between the values in its real user ID and saved set-user-ID. This switching is done via calls to *seteuid*(2), *setreuid*(2), or *setresuid*(2). A set-group-ID program performs the analogous tasks using *setegid*(2), *setregid*(2), or *setresgid*(2). A process can obtain its saved set-user-ID (set-group-ID) using *getresuid*(2) (**getresgid**(2)).
- Filesystem user ID and filesystem group ID (Linux-specific). These IDs, in conjunction with the supplementary group IDs described below, are used to determine permissions for accessing files; see *path_resolution*(7) for details. Whenever a process's effective user (group) ID is changed, the kernel also automatically changes the filesystem user (group) ID to the same value. Consequently, the filesystem IDs normally have the same values as the corresponding effective ID, and the semantics for file-permission checks are thus the same on Linux as on other UNIX systems. The filesystem IDs can be made to differ from the effective IDs by calling *setfsuid*(2) and *setfsgid*(2).
- Supplementary group IDs. This is a set of additional group IDs that are used for permission checks when accessing files and other shared resources. Before Linux 2.6.4, a process can be a member of up to 32 supplementary groups; since Linux 2.6.4, a process can be a member of up to 65536 supplementary groups. The call *sysconf(_SC_NGROUPS_MAX)* can be used to determine the number of supplementary groups of which a process may be a member. A process can obtain its set of supplementary group IDs using *getgroups*(2).

A child process created by *fork*(2) inherits copies of its parent's user and groups IDs. During an *execve*(2), a process's real user and group ID and supplementary group IDs are preserved; the effective and saved set IDs may be changed, as described in *execve*(2).

Aside from the purposes noted above, a process's user IDs are also employed in a number of other contexts:

- when determining the permissions for sending signals (see *kill*(2));
- when determining the permissions for setting process-scheduling parameters (nice value, real time scheduling policy and priority, CPU affinity, I/O priority) using *setpriority*(2), *sched_setaffinity*(2), *sched_setscheduler*(2), *sched_setparam*(2), *sched_setattr*(2), and *ioprio_set*(2);
- when checking resource limits (see *getrlimit*(2));
- when checking the limit on the number of inotify instances that the process may create (see *inotify*(7)).

Modifying process user and group IDs

Subject to rules described in the relevant manual pages, a process can use the following APIs to modify its user and group IDs:

setuid(2) (*setgid*(2))
: Modify the process's real (and possibly effective and saved-set) user (group) IDs.

seteuid(2) (*setegid*(2))
: Modify the process's effective user (group) ID.

setfsuid(2) (*setfsgid*(2))
: Modify the process's filesystem user (group) ID.

setreuid(2) (*setregid*(2))
: Modify the process's real and effective (and possibly saved-set) user (group) IDs.

setresuid(2) (*setresgid*(2))
: Modify the process's real, effective, and saved-set user (group) IDs.

setgroups(2)
: Modify the process's supplementary group list.

Any changes to a process's effective user (group) ID are automatically carried over to the process's filesystem user (group) ID. Changes to a process's effective user or group ID can also affect the process "dumpable" attribute, as described in *prctl*(2).

Changes to process user and group IDs can affect the capabilities of the process, as described in *capabilities*(7).

STANDARDS

Process IDs, parent process IDs, process group IDs, and session IDs are specified in POSIX.1. The real, effective, and saved set user and groups IDs, and the supplementary group IDs, are specified in POSIX.1.

The filesystem user and group IDs are a Linux extension.

NOTES

Various fields in the */proc/*pid*/status* file show the process credentials described above. See *proc*(5) for further information.

The POSIX threads specification requires that credentials are shared by all of the threads in a process. However, at the kernel level, Linux maintains separate user and group credentials for each thread. The NPTL threading implementation does some work to ensure that any change to user or group credentials (e.g., calls to *setuid*(2), *setresuid*(2)) is carried through to all of the POSIX threads in a process. See *nptl*(7) for further details.

SEE ALSO

bash(1), *csh*(1), *groups*(1), *id*(1), *newgrp*(1), *ps*(1), *runuser*(1), *setpriv*(1), *sg*(1), *su*(1), *access*(2), *execve*(2), *faccessat*(2), *fork*(2), *getgroups*(2), *getpgrp*(2), *getpid*(2), *getppid*(2), *getsid*(2), *kill*(2), *setegid*(2), *seteuid*(2), *setfsgid*(2), *setfsuid*(2), *setgid*(2), *setgroups*(2), *setpgid*(2), *setresgid*(2), *setresuid*(2), *setsid*(2), *setuid*(2), *waitpid*(2), *euidaccess*(3), *initgroups*(3), *killpg*(3), *tcgetpgrp*(3), *tcgetsid*(3), *tcsetpgrp*(3), *group*(5), *passwd*(5), *shadow*(5), *capabilities*(7), *namespaces*(7), *path_resolution*(7), *pid_namespaces*(7), *pthreads*(7), *signal*(7), *system_data_types*(7), *unix*(7), *user_namespaces*(7), *sudo*(8)

NAME

ddp − Linux AppleTalk protocol implementation

SYNOPSIS

#include <sys/socket.h>
#include <netatalk/at.h>

ddp_socket = **socket(AF_APPLETALK, SOCK_DGRAM, 0);**
raw_socket = **socket(AF_APPLETALK, SOCK_RAW,** *protocol***);**

DESCRIPTION

Linux implements the AppleTalk protocols described in *Inside AppleTalk*. Only the DDP layer and AARP are present in the kernel. They are designed to be used via the **netatalk** protocol libraries. This page documents the interface for those who wish or need to use the DDP layer directly.

The communication between AppleTalk and the user program works using a BSD-compatible socket interface. For more information on sockets, see *socket*(7).

An AppleTalk socket is created by calling the *socket*(2) function with a **AF_APPLETALK** socket family argument. Valid socket types are **SOCK_DGRAM** to open a **ddp** socket or **SOCK_RAW** to open a **raw** socket. *protocol* is the AppleTalk protocol to be received or sent. For **SOCK_RAW** you must specify **ATPROTO_DDP**.

Raw sockets may be opened only by a process with effective user ID 0 or when the process has the **CAP_NET_RAW** capability.

Address format

An AppleTalk socket address is defined as a combination of a network number, a node number, and a port number.

```
struct at_addr {
    unsigned short s_net;
    unsigned char  s_node;
};

struct sockaddr_atalk {
    sa_family_t    sat_family;    /* address family */
    unsigned char  sat_port;      /* port */
    struct at_addr sat_addr;      /* net/node */
};
```

sat_family is always set to **AF_APPLETALK**. *sat_port* contains the port. The port numbers below 129 are known as *reserved ports*. Only processes with the effective user ID 0 or the **CAP_NET_BIND_SERVICE** capability may *bind*(2) to these sockets. *sat_addr* is the host address. The *net* member of *struct at_addr* contains the host network in network byte order. The value of **AT_ANYNET** is a wildcard and also implies "this network." The *node* member of *struct at_addr* contains the host node number. The value of **AT_ANYNODE** is a wildcard and also implies "this node." The value of **ATADDR_BCAST** is a link local broadcast address.

Socket options

No protocol-specific socket options are supported.

/proc interfaces

IP supports a set of */proc* interfaces to configure some global AppleTalk parameters. The parameters can be accessed by reading or writing files in the directory */proc/sys/net/atalk/* .

aarp−expiry−time
The time interval (in seconds) before an AARP cache entry expires.

aarp−resolve−time
The time interval (in seconds) before an AARP cache entry is resolved.

aarp−retransmit−limit
The number of retransmissions of an AARP query before the node is declared dead.

aarp−tick−time
The timer rate (in seconds) for the timer driving AARP.

The default values match the specification and should never need to be changed.

Ioctls

All ioctls described in *socket*(7) apply to DDP.

ERRORS

EACCES
The user tried to execute an operation without the necessary permissions. These include sending to a broadcast address without having the broadcast flag set, and trying to bind to a reserved port without effective user ID 0 or **CAP_NET_BIND_SERVICE**.

EADDRINUSE
Tried to bind to an address already in use.

EADDRNOTAVAIL
A nonexistent interface was requested or the requested source address was not local.

EAGAIN
Operation on a nonblocking socket would block.

EALREADY
A connection operation on a nonblocking socket is already in progress.

ECONNABORTED
A connection was closed during an *accept*(2).

EHOSTUNREACH
No routing table entry matches the destination address.

EINVAL
Invalid argument passed.

EISCONN
connect(2) was called on an already connected socket.

EMSGSIZE
Datagram is bigger than the DDP MTU.

ENODEV
Network device not available or not capable of sending IP.

ENOENT
SIOCGSTAMP was called on a socket where no packet arrived.

ENOMEM and **ENOBUFS**
Not enough memory available.

ENOPKG
A kernel subsystem was not configured.

ENOPROTOOPT and **EOPNOTSUPP**
Invalid socket option passed.

ENOTCONN
The operation is defined only on a connected socket, but the socket wasn't connected.

EPERM
User doesn't have permission to set high priority, make a configuration change, or send signals to the requested process or group.

EPIPE The connection was unexpectedly closed or shut down by the other end.

ESOCKTNOSUPPORT
The socket was unconfigured, or an unknown socket type was requested.

VERSIONS

AppleTalk is supported by Linux 2.0 or higher. The */proc* interfaces exist since Linux 2.2.

NOTES

Be very careful with the **SO_BROADCAST** option; it is not privileged in Linux. It is easy to overload the network with careless sending to broadcast addresses.

Compatibility

The basic AppleTalk socket interface is compatible with **netatalk** on BSD-derived systems. Many BSD systems fail to check **SO_BROADCAST** when sending broadcast frames; this can lead to compatibility problems.

The raw socket mode is unique to Linux and exists to support the alternative CAP package and AppleTalk monitoring tools more easily.

BUGS

There are too many inconsistent error values.

The ioctls used to configure routing tables, devices, AARP tables, and other devices are not yet described.

SEE ALSO

recvmsg(2), *sendmsg*(2), *capabilities*(7), *socket*(7)

NAME

environ − user environment

SYNOPSIS

extern char *****environ*;**

DESCRIPTION

The variable *environ* points to an array of pointers to strings called the "environment". The last pointer in this array has the value NULL. This array of strings is made available to the process by the *execve*(2) call when a new program is started. When a child process is created via *fork*(2), it inherits a *copy* of its parent's environment.

By convention, the strings in *environ* have the form "*name*=*value*". The name is case-sensitive and may not contain the character "=". The value can be anything that can be represented as a string. The name and the value may not contain an embedded null byte ('\0'), since this is assumed to terminate the string.

Environment variables may be placed in the shell's environment by the *export* command in *sh*(1), or by the *setenv* command if you use *csh*(1)

The initial environment of the shell is populated in various ways, such as definitions from */etc/environment* that are processed by *pam_env*(8) for all users at login time (on systems that employ *pam*(8)). In addition, various shell initialization scripts, such as the system-wide */etc/profile* script and per-user initializations script may include commands that add variables to the shell's environment; see the manual page of your preferred shell for details.

Bourne-style shells support the syntax

```
NAME=value command
```

to create an environment variable definition only in the scope of the process that executes *command*. Multiple variable definitions, separated by white space, may precede *command*.

Arguments may also be placed in the environment at the point of an *exec*(3). A C program can manipulate its environment using the functions *getenv*(3), *putenv*(3), *setenv*(3), and *unsetenv*(3).

What follows is a list of environment variables typically seen on a system. This list is incomplete and includes only common variables seen by average users in their day-to-day routine. Environment variables specific to a particular program or library function are documented in the ENVIRONMENT section of the appropriate manual page.

USER The name of the logged-in user (used by some BSD-derived programs). Set at login time, see section NOTES below.

LOGNAME
The name of the logged-in user (used by some System-V derived programs). Set at login time, see section NOTES below.

HOME
A user's login directory. Set at login time, see section NOTES below.

LANG The name of a locale to use for locale categories when not overridden by **LC_ALL** or more specific environment variables such as **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** (see *locale*(7) for further details of the **LC_*** environment variables).

PATH The sequence of directory prefixes that *sh*(1) and many other programs employ when searching for an executable file that is specified as a simple filename (i.a., a pathname that contains no slashes). The prefixes are separated by colons (**:**). The list of prefixes is searched from beginning to end, by checking the pathname formed by concatenating a prefix, a slash, and the filename, until a file with execute permission is found.

As a legacy feature, a zero-length prefix (specified as two adjacent colons, or an initial or terminating colon) is interpreted to mean the current working directory. However, use of this feature is deprecated, and POSIX notes that a conforming application shall use an explicit pathname (e.g., .) to specify the current working directory.

Analogously to **PATH**, one has **CDPATH** used by some shells to find the target of a change directory command, **MANPATH** used by *man*(1) to find manual pages, and so on.

PWD Absolute path to the current working directory; required to be partially canonical (no . or .. components).

SHELL
The absolute pathname of the user's login shell. Set at login time, see section NOTES below.

TERM The terminal type for which output is to be prepared.

PAGER
The user's preferred utility to display text files. Any string acceptable as a command-string operand to the *sh* −*c* command shall be valid. If **PAGER** is null or is not set, then applications that launch a pager will default to a program such as *less*(1) or *more*(1)

EDITOR/VISUAL
The user's preferred utility to edit text files. Any string acceptable as a command_string operand to the *sh* −*c* command shall be valid.

Note that the behavior of many programs and library routines is influenced by the presence or value of certain environment variables. Examples include the following:

- The variables **LANG**, **LANGUAGE**, **NLSPATH**, **LOCPATH**, **LC_ALL**, **LC_MESSAGES**, and so on influence locale handling; see *catopen*(3), *gettext*(3), and *locale*(7).
- **TMPDIR** influences the path prefix of names created by *tempnam*(3) and other routines, and the temporary directory used by *sort*(1) and other programs.
- **LD_LIBRARY_PATH**, **LD_PRELOAD**, and other **LD_*** variables influence the behavior of the dynamic loader/linker. See also *ld.so*(8).
- **POSIXLY_CORRECT** makes certain programs and library routines follow the prescriptions of POSIX.
- The behavior of *malloc*(3) is influenced by **MALLOC_*** variables.
- The variable **HOSTALIASES** gives the name of a file containing aliases to be used with *gethostbyname*(3).
- **TZ** and **TZDIR** give timezone information used by *tzset*(3) and through that by functions like *ctime*(3), *localtime*(3), *mktime*(3), *strftime*(3). See also *tzselect*(8).
- **TERMCAP** gives information on how to address a given terminal (or gives the name of a file containing such information).
- **COLUMNS** and **LINES** tell applications about the window size, possibly overriding the actual size.
- **PRINTER** or **LPDEST** may specify the desired printer to use. See *lpr*(1)

NOTES

Historically and by standard, *environ* must be declared in the user program. However, as a (nonstandard) programmer convenience, *environ* is declared in the header file <*unistd.h*> if the **_GNU_SOURCE** feature test macro is defined (see *feature_test_macros*(7)).

The *prctl*(2) **PR_SET_MM_ENV_START** and **PR_SET_MM_ENV_END** operations can be used to control the location of the process's environment.

The **HOME**, **LOGNAME**, **SHELL**, and **USER** variables are set when the user is changed via a session management interface, typically by a program such as *login*(1) from a user database (such as *passwd*(5)). (Switching to the root user using *su*(1) may result in a mixed environment where **LOGNAME** and **USER** are retained from old user; see the *su*(1) manual page.)

BUGS

Clearly there is a security risk here. Many a system command has been tricked into mischief by a user who specified unusual values for **IFS** or **LD_LIBRARY_PATH**.

There is also the risk of name space pollution. Programs like *make* and *autoconf* allow overriding of default utility names from the environment with similarly named variables in all caps. Thus one uses **CC** to select the desired C compiler (and similarly **MAKE**, **AR**, **AS**, **FC**, **LD**, **LEX**, **RM**, **YACC**, etc.). However, in some traditional uses such an environment variable gives options for the program instead of a pathname. Thus, one has **MORE** and **LESS**. Such usage is considered mistaken, and to be avoided in new programs.

SEE ALSO

bash(1), *csh*(1), *env*(1), *login*(1), *printenv*(1), *sh*(1), *su*(1), *tcsh*(1), *execve*(2), *clearenv*(3), *exec*(3), *getenv*(3), *putenv*(3), *setenv*(3), *unsetenv*(3), *locale*(7), *ld.so*(8), *pam_env*(8)

NAME

epoll − I/O event notification facility

SYNOPSIS

#include <sys/epoll.h>

DESCRIPTION

The **epoll** API performs a similar task to *poll*(2): monitoring multiple file descriptors to see if I/O is possible on any of them. The **epoll** API can be used either as an edge-triggered or a level-triggered interface and scales well to large numbers of watched file descriptors.

The central concept of the **epoll** API is the **epoll** *instance*, an in-kernel data structure which, from a user-space perspective, can be considered as a container for two lists:

- The *interest* list (sometimes also called the **epoll** set): the set of file descriptors that the process has registered an interest in monitoring.
- The *ready* list: the set of file descriptors that are "ready" for I/O. The ready list is a subset of (or, more precisely, a set of references to) the file descriptors in the interest list. The ready list is dynamically populated by the kernel as a result of I/O activity on those file descriptors.

The following system calls are provided to create and manage an **epoll** instance:

- *epoll_create*(2) creates a new **epoll** instance and returns a file descriptor referring to that instance. (The more recent *epoll_create1*(2) extends the functionality of *epoll_create*(2).)
- Interest in particular file descriptors is then registered via *epoll_ctl*(2), which adds items to the interest list of the **epoll** instance.
- *epoll_wait*(2) waits for I/O events, blocking the calling thread if no events are currently available. (This system call can be thought of as fetching items from the ready list of the **epoll** instance.)

Level-triggered and edge-triggered

The **epoll** event distribution interface is able to behave both as edge-triggered (ET) and as level-triggered (LT). The difference between the two mechanisms can be described as follows. Suppose that this scenario happens:

(1) The file descriptor that represents the read side of a pipe (*rfd*) is registered on the **epoll** instance.

(2) A pipe writer writes 2 kB of data on the write side of the pipe.

(3) A call to *epoll_wait*(2) is done that will return *rfd* as a ready file descriptor.

(4) The pipe reader reads 1 kB of data from *rfd*.

(5) A call to *epoll_wait*(2) is done.

If the *rfd* file descriptor has been added to the **epoll** interface using the **EPOLLET** (edge-triggered) flag, the call to *epoll_wait*(2) done in step **5** will probably hang despite the available data still present in the file input buffer; meanwhile the remote peer might be expecting a response based on the data it already sent. The reason for this is that edge-triggered mode delivers events only when changes occur on the monitored file descriptor. So, in step **5** the caller might end up waiting for some data that is already present inside the input buffer. In the above example, an event on *rfd* will be generated because of the write done in **2** and the event is consumed in **3**. Since the read operation done in **4** does not consume the whole buffer data, the call to *epoll_wait*(2) done in step **5** might block indefinitely.

An application that employs the **EPOLLET** flag should use nonblocking file descriptors to avoid having a blocking read or write starve a task that is handling multiple file descriptors. The suggested way to use **epoll** as an edge-triggered (**EPOLLET**) interface is as follows:

(1) with nonblocking file descriptors; and

(2) by waiting for an event only after *read*(2) or *write*(2) return **EAGAIN**.

By contrast, when used as a level-triggered interface (the default, when **EPOLLET** is not specified), **epoll** is simply a faster *poll*(2), and can be used wherever the latter is used since it shares the same semantics.

Since even with edge-triggered **epoll**, multiple events can be generated upon receipt of multiple chunks of data, the caller has the option to specify the **EPOLLONESHOT** flag, to tell **epoll** to disable the associated file descriptor after the receipt of an event with *epoll_wait*(2). When the **EPOLLONESHOT**

flag is specified, it is the caller's responsibility to rearm the file descriptor using *epoll_ctl*(2) with **EPOLL_CTL_MOD**.

If multiple threads (or processes, if child processes have inherited the **epoll** file descriptor across *fork*(2)) are blocked in *epoll_wait*(2) waiting on the same epoll file descriptor and a file descriptor in the interest list that is marked for edge-triggered (**EPOLLET**) notification becomes ready, just one of the threads (or processes) is awoken from *epoll_wait*(2). This provides a useful optimization for avoiding "thundering herd" wake-ups in some scenarios.

Interaction with autosleep

If the system is in **autosleep** mode via */sys/power/autosleep* and an event happens which wakes the device from sleep, the device driver will keep the device awake only until that event is queued. To keep the device awake until the event has been processed, it is necessary to use the *epoll_ctl*(2) **EPOLLWAKEUP** flag.

When the **EPOLLWAKEUP** flag is set in the **events** field for a *struct epoll_event*, the system will be kept awake from the moment the event is queued, through the *epoll_wait*(2) call which returns the event until the subsequent *epoll_wait*(2) call. If the event should keep the system awake beyond that time, then a separate *wake_lock* should be taken before the second *epoll_wait*(2) call.

/proc interfaces

The following interfaces can be used to limit the amount of kernel memory consumed by epoll:

/proc/sys/fs/epoll/max_user_watches (since Linux 2.6.28)

This specifies a limit on the total number of file descriptors that a user can register across all epoll instances on the system. The limit is per real user ID. Each registered file descriptor costs roughly 90 bytes on a 32-bit kernel, and roughly 160 bytes on a 64-bit kernel. Currently, the default value for *max_user_watches* is 1/25 (4%) of the available low memory, divided by the registration cost in bytes.

Example for suggested usage

While the usage of **epoll** when employed as a level-triggered interface does have the same semantics as *poll*(2), the edge-triggered usage requires more clarification to avoid stalls in the application event loop. In this example, listener is a nonblocking socket on which *listen*(2) has been called. The function *do_use_fd()* uses the new ready file descriptor until **EAGAIN** is returned by either *read*(2) or *write*(2). An event-driven state machine application should, after having received **EAGAIN**, record its current state so that at the next call to *do_use_fd()* it will continue to *read*(2) or *write*(2) from where it stopped before.

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;

/* Code to set up listening socket, 'listen_sock',
   (socket(), bind(), listen()) omitted. */

epollfd = epoll_create1(0);
if (epollfd == -1) {
    perror("epoll_create1");
    exit(EXIT_FAILURE);
}

ev.events = EPOLLIN;
ev.data.fd = listen_sock;
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
    perror("epoll_ctl: listen_sock");
    exit(EXIT_FAILURE);
}

for (;;) {
    nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
    if (nfds == -1) {
        perror("epoll_wait");
```

```
            exit(EXIT_FAILURE);
        }

        for (n = 0; n < nfds; ++n) {
            if (events[n].data.fd == listen_sock) {
                conn_sock = accept(listen_sock,
                                   (struct sockaddr *) &addr, &addrlen);
                if (conn_sock == -1) {
                    perror("accept");
                    exit(EXIT_FAILURE);
                }
                setnonblocking(conn_sock);
                ev.events = EPOLLIN | EPOLLET;
                ev.data.fd = conn_sock;
                if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock,
                              &ev) == -1) {
                    perror("epoll_ctl: conn_sock");
                    exit(EXIT_FAILURE);
                }
            } else {
                do_use_fd(events[n].data.fd);
            }
        }
    }
```

When used as an edge-triggered interface, for performance reasons, it is possible to add the file descriptor inside the **epoll** interface (**EPOLL_CTL_ADD**) once by specifying (**EPOLLIN**|**EPOLLOUT**). This allows you to avoid continuously switching between **EPOLLIN** and **EPOLLOUT** calling *epoll_ctl*(2) with **EPOLL_CTL_MOD**.

Questions and answers

- What is the key used to distinguish the file descriptors registered in an interest list?

  The key is the combination of the file descriptor number and the open file description (also known as an "open file handle", the kernel's internal representation of an open file).

- What happens if you register the same file descriptor on an **epoll** instance twice?

  You will probably get **EEXIST**. However, it is possible to add a duplicate (**dup**(2), *dup2*(2), *fcntl*(2) **F_DUPFD**) file descriptor to the same **epoll** instance. This can be a useful technique for filtering events, if the duplicate file descriptors are registered with different *events* masks.

- Can two **epoll** instances wait for the same file descriptor? If so, are events reported to both **epoll** file descriptors?

  Yes, and events would be reported to both. However, careful programming may be needed to do this correctly.

- Is the **epoll** file descriptor itself poll/epoll/selectable?

  Yes. If an **epoll** file descriptor has events waiting, then it will indicate as being readable.

- What happens if one attempts to put an **epoll** file descriptor into its own file descriptor set?

  The *epoll_ctl*(2) call fails (**EINVAL**). However, you can add an **epoll** file descriptor inside another **epoll** file descriptor set.

- Can I send an **epoll** file descriptor over a UNIX domain socket to another process?

  Yes, but it does not make sense to do this, since the receiving process would not have copies of the file descriptors in the interest list.

- Will closing a file descriptor cause it to be removed from all **epoll** interest lists?

  Yes, but be aware of the following point. A file descriptor is a reference to an open file description (see *open*(2)). Whenever a file descriptor is duplicated via *dup*(2), *dup2*(2), *fcntl*(2) **F_DUPFD**, or *fork*(2), a new file descriptor referring to the same open file description is created. An open file description continues to exist until all file descriptors referring to it have been closed.

A file descriptor is removed from an interest list only after all the file descriptors referring to the underlying open file description have been closed. This means that even after a file descriptor that is part of an interest list has been closed, events may be reported for that file descriptor if other file descriptors referring to the same underlying file description remain open. To prevent this happening, the file descriptor must be explicitly removed from the interest list (using *epoll_ctl*(2) **EPOLL_CTL_DEL**) before it is duplicated. Alternatively, the application must ensure that all file descriptors are closed (which may be difficult if file descriptors were duplicated behind the scenes by library functions that used *dup*(2) or *fork*(2)).

- If more than one event occurs between *epoll_wait*(2) calls, are they combined or reported separately?

  They will be combined.

- Does an operation on a file descriptor affect the already collected but not yet reported events?

  You can do two operations on an existing file descriptor. Remove would be meaningless for this case. Modify will reread available I/O.

- Do I need to continuously read/write a file descriptor until **EAGAIN** when using the **EPOLLET** flag (edge-triggered behavior)?

  Receiving an event from *epoll_wait*(2) should suggest to you that such file descriptor is ready for the requested I/O operation. You must consider it ready until the next (nonblocking) read/write yields **EAGAIN**. When and how you will use the file descriptor is entirely up to you.

  For packet/token-oriented files (e.g., datagram socket, terminal in canonical mode), the only way to detect the end of the read/write I/O space is to continue to read/write until **EAGAIN**.

  For stream-oriented files (e.g., pipe, FIFO, stream socket), the condition that the read/write I/O space is exhausted can also be detected by checking the amount of data read from / written to the target file descriptor. For example, if you call *read*(2) by asking to read a certain amount of data and *read*(2) returns a lower number of bytes, you can be sure of having exhausted the read I/O space for the file descriptor. The same is true when writing using *write*(2). (Avoid this latter technique if you cannot guarantee that the monitored file descriptor always refers to a stream-oriented file.)

Possible pitfalls and ways to avoid them

- **Starvation (edge-triggered)**

  If there is a large amount of I/O space, it is possible that by trying to drain it the other files will not get processed causing starvation. (This problem is not specific to **epoll**.)

  The solution is to maintain a ready list and mark the file descriptor as ready in its associated data structure, thereby allowing the application to remember which files need to be processed but still round robin amongst all the ready files. This also supports ignoring subsequent events you receive for file descriptors that are already ready.

- **If using an event cache...**

  If you use an event cache or store all the file descriptors returned from *epoll_wait*(2), then make sure to provide a way to mark its closure dynamically (i.e., caused by a previous event's processing). Suppose you receive 100 events from *epoll_wait*(2), and in event #47 a condition causes event #13 to be closed. If you remove the structure and *close*(2) the file descriptor for event #13, then your event cache might still say there are events waiting for that file descriptor causing confusion.

  One solution for this is to call, during the processing of event 47, *epoll_ctl*(*EPOLL_CTL_DEL*) to delete file descriptor 13 and *close*(2), then mark its associated data structure as removed and link it to a cleanup list. If you find another event for file descriptor 13 in your batch processing, you will discover the file descriptor had been previously removed and there will be no confusion.

VERSIONS

Some other systems provide similar mechanisms; for example, FreeBSD has *kqueue*, and Solaris has */dev/poll*.

STANDARDS

Linux.

HISTORY

Linux 2.5.44. glibc 2.3.2.

NOTES

The set of file descriptors that is being monitored via an epoll file descriptor can be viewed via the entry for the epoll file descriptor in the process's */proc/*pid*/fdinfo* directory. See *proc*(5) for further details.

The *kcmp*(2) **KCMP_EPOLL_TFD** operation can be used to test whether a file descriptor is present in an epoll instance.

SEE ALSO

epoll_create(2), *epoll_create1*(2), *epoll_ctl*(2), *epoll_wait*(2), *poll*(2), *select*(2)

NAME

fanotify − monitoring filesystem events

DESCRIPTION

The fanotify API provides notification and interception of filesystem events. Use cases include virus scanning and hierarchical storage management. In the original fanotify API, only a limited set of events was supported. In particular, there was no support for create, delete, and move events. The support for those events was added in Linux 5.1. (See *inotify*(7) for details of an API that did notify those events pre Linux 5.1.)

Additional capabilities compared to the *inotify*(7) API include the ability to monitor all of the objects in a mounted filesystem, the ability to make access permission decisions, and the possibility to read or modify files before access by other applications.

The following system calls are used with this API: *fanotify_init*(2), *fanotify_mark*(2), *read*(2), *write*(2), and *close*(2).

fanotify_init(), fanotify_mark(), and notification groups

The *fanotify_init*(2) system call creates and initializes an fanotify notification group and returns a file descriptor referring to it.

An fanotify notification group is a kernel-internal object that holds a list of files, directories, filesystems, and mounts for which events shall be created.

For each entry in an fanotify notification group, two bit masks exist: the *mark* mask and the *ignore* mask. The mark mask defines file activities for which an event shall be created. The ignore mask defines activities for which no event shall be generated. Having these two types of masks permits a filesystem, mount, or directory to be marked for receiving events, while at the same time ignoring events for specific objects under a mount or directory.

The *fanotify_mark*(2) system call adds a file, directory, filesystem, or mount to a notification group and specifies which events shall be reported (or ignored), or removes or modifies such an entry.

A possible usage of the ignore mask is for a file cache. Events of interest for a file cache are modification of a file and closing of the same. Hence, the cached directory or mount is to be marked to receive these events. After receiving the first event informing that a file has been modified, the corresponding cache entry will be invalidated. No further modification events for this file are of interest until the file is closed. Hence, the modify event can be added to the ignore mask. Upon receiving the close event, the modify event can be removed from the ignore mask and the file cache entry can be updated.

The entries in the fanotify notification groups refer to files and directories via their inode number and to mounts via their mount ID. If files or directories are renamed or moved within the same mount, the respective entries survive. If files or directories are deleted or moved to another mount or if filesystems or mounts are unmounted, the corresponding entries are deleted.

The event queue

As events occur on the filesystem objects monitored by a notification group, the fanotify system generates events that are collected in a queue. These events can then be read (using *read*(2) or similar) from the fanotify file descriptor returned by *fanotify_init*(2).

Two types of events are generated: *notification* events and *permission* events. Notification events are merely informative and require no action to be taken by the receiving application with one exception: if a valid file descriptor is provided within a generic event, the file descriptor must be closed. Permission events are requests to the receiving application to decide whether permission for a file access shall be granted. For these events, the recipient must write a response which decides whether access is granted or not.

An event is removed from the event queue of the fanotify group when it has been read. Permission events that have been read are kept in an internal list of the fanotify group until either a permission decision has been taken by writing to the fanotify file descriptor or the fanotify file descriptor is closed.

Reading fanotify events

Calling *read*(2) for the file descriptor returned by *fanotify_init*(2) blocks (if the flag **FAN_NONBLOCK** is not specified in the call to *fanotify_init*(2)) until either a file event occurs or the call is interrupted by a signal (see *signal*(7)).

After a successful *read*(2), the read buffer contains one or more of the following structures:

```
struct fanotify_event_metadata {
    __u32 event_len;
    __u8 vers;
    __u8 reserved;
    __u16 metadata_len;
    __aligned_u64 mask;
    __s32 fd;
    __s32 pid;
};
```

Information records are supplemental pieces of information that may be provided alongside the generic *fanotify_event_metadata* structure. The *flags* passed to *fanotify_init*(2) have influence over the type of information records that may be returned for an event. For example, if a notification group is initialized with **FAN_REPORT_FID** or **FAN_REPORT_DIR_FID**, then event listeners should also expect to receive a *fanotify_event_info_fid* structure alongside the *fanotify_event_metadata* structure, whereby file handles are used to identify filesystem objects rather than file descriptors. Information records may also be stacked, meaning that using the various **FAN_REPORT_*** flags in conjunction with one another is supported. In such cases, multiple information records can be returned for an event alongside the generic *fanotify_event_metadata* structure. For example, if a notification group is initialized with **FAN_REPORT_TARGET_FID** and **FAN_REPORT_PIDFD**, then an event listener should expect to receive up to two *fanotify_event_info_fid* information records and one *fanotify_event_info_pidfd* information record alongside the generic *fanotify_event_metadata* structure. Importantly, fanotify provides no guarantee around the ordering of information records when a notification group is initialized with a stacked based configuration. Each information record has a nested structure of type *fanotify_event_info_header*. It is imperative for event listeners to inspect the *info_type* field of this structure in order to determine the type of information record that had been received for a given event.

In cases where an fanotify group identifies filesystem objects by file handles, event listeners should also expect to receive one or more of the below information record objects alongside the generic *fanotify_event_metadata* structure within the read buffer:

```
struct fanotify_event_info_fid {
    struct fanotify_event_info_header hdr;
    __kernel_fsid_t fsid;
    unsigned char handle[];
};
```

In cases where an fanotify group is initialized with **FAN_REPORT_PIDFD**, event listeners should expect to receive the below information record object alongside the generic *fanotify_event_metadata* structure within the read buffer:

```
struct fanotify_event_info_pidfd {
        struct fanotify_event_info_header hdr;
        __s32 pidfd;
};
```

In case of a **FAN_FS_ERROR** event, an additional information record describing the error that occurred is returned alongside the generic *fanotify_event_metadata* structure within the read buffer. This structure is defined as follows:

```
struct fanotify_event_info_error {
    struct fanotify_event_info_header hdr;
    __s32 error;
    __u32 error_count;
};
```

All information records contain a nested structure of type *fanotify_event_info_header*. This structure holds meta-information about the information record that may have been returned alongside the generic *fanotify_event_metadata* structure. This structure is defined as follows:

```
struct fanotify_event_info_header {
      __u8 info_type;
      __u8 pad;
      __u16 len;
```

```
};
```

For performance reasons, it is recommended to use a large buffer size (for example, 4096 bytes), so that multiple events can be retrieved by a single *read*(2).

The return value of *read*(2) is the number of bytes placed in the buffer, or −1 in case of an error (but see BUGS).

The fields of the *fanotify_event_metadata* structure are as follows:

event_len
: This is the length of the data for the current event and the offset to the next event in the buffer. Unless the group identifies filesystem objects by file handles, the value of *event_len* is always **FAN_EVENT_METADATA_LEN**. For a group that identifies filesystem objects by file handles, *event_len* also includes the variable length file identifier records.

vers
: This field holds a version number for the structure. It must be compared to **FANOTIFY_METADATA_VERSION** to verify that the structures returned at run time match the structures defined at compile time. In case of a mismatch, the application should abandon trying to use the fanotify file descriptor.

reserved
: This field is not used.

metadata_len
: This is the length of the structure. The field was introduced to facilitate the implementation of optional headers per event type. No such optional headers exist in the current implementation.

mask
: This is a bit mask describing the event (see below).

fd
: This is an open file descriptor for the object being accessed, or **FAN_NOFD** if a queue overflow occurred. With an fanotify group that identifies filesystem objects by file handles, applications should expect this value to be set to **FAN_NOFD** for each event that is received. The file descriptor can be used to access the contents of the monitored file or directory. The reading application is responsible for closing this file descriptor.

  When calling *fanotify_init*(2), the caller may specify (via the *event_f_flags* argument) various file status flags that are to be set on the open file description that corresponds to this file descriptor. In addition, the (kernel-internal) **FMODE_NONOTIFY** file status flag is set on the open file description. This flag suppresses fanotify event generation. Hence, when the receiver of the fanotify event accesses the notified file or directory using this file descriptor, no additional events will be created.

pid
: If flag **FAN_REPORT_TID** was set in *fanotify_init*(2), this is the TID of the thread that caused the event. Otherwise, this the PID of the process that caused the event.

A program listening to fanotify events can compare this PID to the PID returned by *getpid*(2), to determine whether the event is caused by the listener itself, or is due to a file access by another process.

The bit mask in *mask* indicates which events have occurred for a single filesystem object. Multiple bits may be set in this mask, if more than one event occurred for the monitored filesystem object. In particular, consecutive events for the same filesystem object and originating from the same process may be merged into a single event, with the exception that two permission events are never merged into one queue entry.

The bits that may appear in *mask* are as follows:

FAN_ACCESS
: A file or a directory (but see BUGS) was accessed (read).

FAN_OPEN
: A file or a directory was opened.

FAN_OPEN_EXEC
: A file was opened with the intent to be executed. See NOTES in *fanotify_mark*(2) for additional details.

FAN_ATTRIB
A file or directory metadata was changed.

FAN_CREATE
A child file or directory was created in a watched parent.

FAN_DELETE
A child file or directory was deleted in a watched parent.

FAN_DELETE_SELF
A watched file or directory was deleted.

FAN_FS_ERROR
A filesystem error was detected.

FAN_RENAME
A file or directory has been moved to or from a watched parent directory.

FAN_MOVED_FROM
A file or directory has been moved from a watched parent directory.

FAN_MOVED_TO
A file or directory has been moved to a watched parent directory.

FAN_MOVE_SELF
A watched file or directory was moved.

FAN_MODIFY
A file was modified.

FAN_CLOSE_WRITE
A file that was opened for writing (**O_WRONLY** or **O_RDWR**) was closed.

FAN_CLOSE_NOWRITE
A file or directory that was opened read-only (**O_RDONLY**) was closed.

FAN_Q_OVERFLOW
The event queue exceeded the limit on number of events. This limit can be overridden by specifying the **FAN_UNLIMITED_QUEUE** flag when calling *fanotify_init*(2).

FAN_ACCESS_PERM
An application wants to read a file or directory, for example using *read*(2) or *readdir*(2). The reader must write a response (as described below) that determines whether the permission to access the filesystem object shall be granted.

FAN_OPEN_PERM
An application wants to open a file or directory. The reader must write a response that determines whether the permission to open the filesystem object shall be granted.

FAN_OPEN_EXEC_PERM
An application wants to open a file for execution. The reader must write a response that determines whether the permission to open the filesystem object for execution shall be granted. See NOTES in *fanotify_mark*(2) for additional details.

To check for any close event, the following bit mask may be used:

FAN_CLOSE
A file was closed. This is a synonym for:

```
FAN_CLOSE_WRITE | FAN_CLOSE_NOWRITE
```

To check for any move event, the following bit mask may be used:

FAN_MOVE
A file or directory was moved. This is a synonym for:

```
FAN_MOVED_FROM | FAN_MOVED_TO
```

The following bits may appear in *mask* only in conjunction with other event type bits:

FAN_ONDIR
The events described in the *mask* have occurred on a directory object. Reporting events on directories requires setting this flag in the mark mask. See *fanotify_mark*(2) for additional details. The **FAN_ONDIR** flag is reported in an event mask only if the fanotify group identifies filesystem objects by file handles.

Information records that are supplied alongside the generic *fanotify_event_metadata* structure will always contain a nested structure of type *fanotify_event_info_header*. The fields of the *fanotify_event_info_header* are as follows:

info_type
A unique integer value representing the type of information record object received for an event. The value of this field can be set to one of the following: **FAN_EVENT_INFO_TYPE_FID**, **FAN_EVENT_INFO_TYPE_DFID**, **FAN_EVENT_INFO_TYPE_DFID_NAME**, or **FAN_EVENT_INFO_TYPE_PIDFD**. The value set for this field is dependent on the flags that have been supplied to *fanotify_init*(2). Refer to the field details of each information record object type below to understand the different cases in which the *info_type* values can be set.

pad This field is currently not used by any information record object type and therefore is set to zero.

len The value of *len* is set to the size of the information record object, including the *fanotify_event_info_header*. The total size of all additional information records is not expected to be larger than (*event_len* − *metadata_len*).

The fields of the *fanotify_event_info_fid* structure are as follows:

hdr This is a structure of type *fanotify_event_info_header*. For example, when an fanotify file descriptor is created using **FAN_REPORT_FID**, a single information record is expected to be attached to the event with *info_type* field value of **FAN_EVENT_INFO_TYPE_FID**. When an fanotify file descriptor is created using the combination of **FAN_REPORT_FID** and **FAN_REPORT_DIR_FID**, there may be two information records attached to the event: one with *info_type* field value of **FAN_EVENT_INFO_TYPE_DFID**, identifying a parent directory object, and one with *info_type* field value of **FAN_EVENT_INFO_TYPE_FID**, identifying a child object. Note that for the directory entry modification events **FAN_CREATE**, **FAN_DELETE**, **FAN_MOVE**, and **FAN_RENAME**, an information record identifying the created/deleted/moved child object is reported only if an fanotify group was initialized with the flag **FAN_REPORT_TARGET_FID**.

fsid This is a unique identifier of the filesystem containing the object associated with the event. It is a structure of type *__kernel_fsid_t* and contains the same value as *f_fsid* when calling *statfs*(2).

handle This field contains a variable-length structure of type *struct file_handle*. It is an opaque handle that corresponds to a specified object on a filesystem as returned by *name_to_handle_at*(2). It can be used to uniquely identify a file on a filesystem and can be passed as an argument to *open_by_handle_at*(2). If the value of *info_type* field is **FAN_EVENT_INFO_TYPE_DFID_NAME**, the file handle is followed by a null terminated string that identifies the created/deleted/moved directory entry name. For other events such as **FAN_OPEN**, **FAN_ATTRIB**, **FAN_DELETE_SELF**, and **FAN_MOVE_SELF**, if the value of *info_type* field is **FAN_EVENT_INFO_TYPE_FID**, the *handle* identifies the object correlated to the event. If the value of *info_type* field is **FAN_EVENT_INFO_TYPE_DFID**, the *handle* identifies the directory object correlated to the event or the parent directory of a non-directory object correlated to the event. If the value of *info_type* field is **FAN_EVENT_INFO_TYPE_DFID_NAME**, the *handle* identifies the same directory object that would be reported with **FAN_EVENT_INFO_TYPE_DFID** and the file handle is followed by a null terminated string that identifies the name of a directory entry in that directory, or '.' to identify the directory object itself.

The fields of the *fanotify_event_info_pidfd* structure are as follows:

hdr This is a structure of type *fanotify_event_info_header*. When an fanotify group is initialized using **FAN_REPORT_PIDFD**, the *info_type* field value of the *fanotify_event_info_header* is set to **FAN_EVENT_INFO_TYPE_PIDFD**.

pidfd This is a process file descriptor that refers to the process responsible for generating the event. The returned process file descriptor is no different from one which could be obtained manually if *pidfd_open*(2) were to be called on *fanotify_event_metadata.pid*. In the instance that an error is encountered during pidfd creation, one of two possible error types represented by a negative integer value may be returned in this *pidfd* field. In cases where the process responsible for generating the event has terminated prior to the event listener being able to read events from the notification queue, **FAN_NOPIDFD** is returned. The pidfd creation for an event is only performed at the time the events are read from the notification queue. All other possible pidfd creation failures are represented by **FAN_EPIDFD**. Once the event listener has dealt with an event and the pidfd is no longer required, the pidfd should be closed via *close*(2).

The fields of the *fanotify_event_info_error* structure are as follows:

hdr This is a structure of type *fanotify_event_info_header*. The *info_type* field is set to **FAN_EVENT_INFO_TYPE_ERROR**.

error Identifies the type of error that occurred.

error_count
This is a counter of the number of errors suppressed since the last error was read.

The following macros are provided to iterate over a buffer containing fanotify event metadata returned by a *read*(2) from an fanotify file descriptor:

FAN_EVENT_OK(meta, len)
This macro checks the remaining length *len* of the buffer *meta* against the length of the metadata structure and the *event_len* field of the first metadata structure in the buffer.

FAN_EVENT_NEXT(meta, len)
This macro uses the length indicated in the *event_len* field of the metadata structure pointed to by *meta* to calculate the address of the next metadata structure that follows *meta*. *len* is the number of bytes of metadata that currently remain in the buffer. The macro returns a pointer to the next metadata structure that follows *meta*, and reduces *len* by the number of bytes in the metadata structure that has been skipped over (i.e., it subtracts *meta->event_len* from *len*).

In addition, there is:

FAN_EVENT_METADATA_LEN
This macro returns the size (in bytes) of the structure *fanotify_event_metadata*. This is the minimum size (and currently the only size) of any event metadata.

Monitoring an fanotify file descriptor for events

When an fanotify event occurs, the fanotify file descriptor indicates as readable when passed to *epoll*(7), *poll*(2), or *select*(2).

Dealing with permission events

For permission events, the application must *write*(2) a structure of the following form to the fanotify file descriptor:

```
struct fanotify_response {
    __s32 fd;
    __u32 response;
};
```

The fields of this structure are as follows:

fd This is the file descriptor from the structure *fanotify_event_metadata*.

response
This field indicates whether or not the permission is to be granted. Its value must be either **FAN_ALLOW** to allow the file operation or **FAN_DENY** to deny the file operation.

If access is denied, the requesting application call will receive an **EPERM** error. Additionally, if the notification group has been created with the **FAN_ENABLE_AUDIT** flag, then the **FAN_AUDIT** flag can be set in the *response* field. In that case, the audit subsystem will log information about the access decision to the audit logs.

Monitoring filesystems for errors

A single **FAN_FS_ERROR** event is stored per filesystem at once. Extra error messages are suppressed and accounted for in the *error_count* field of the existing **FAN_FS_ERROR** event record, but details about the errors are lost.

Errors reported by **FAN_FS_ERROR** are generic *errno* values, but not all kinds of error types are reported by all filesystems.

Errors not directly related to a file (i.e. super block corruption) are reported with an invalid *handle*. For these errors, the *handle* will have the field *handle_type* set to **FILEID_INVALID**, and the handle buffer size set to **0**.

Closing the fanotify file descriptor

When all file descriptors referring to the fanotify notification group are closed, the fanotify group is released and its resources are freed for reuse by the kernel. Upon *close*(2), outstanding permission events will be set to allowed.

/proc interfaces

The file */proc/*pid*/fdinfo/*fd contains information about fanotify marks for file descriptor *fd* of process *pid*. See *proc*(5) for details.

Since Linux 5.13, the following interfaces can be used to control the amount of kernel resources consumed by fanotify:

/proc/sys/fs/fanotify/max_queued_events

The value in this file is used when an application calls *fanotify_init*(2) to set an upper limit on the number of events that can be queued to the corresponding fanotify group. Events in excess of this limit are dropped, but an **FAN_Q_OVERFLOW** event is always generated. Prior to Linux kernel 5.13, the hardcoded limit was 16384 events.

/proc/sys/fs/fanotify/max_user_group

This specifies an upper limit on the number of fanotify groups that can be created per real user ID. Prior to Linux kernel 5.13, the hardcoded limit was 128 groups per user.

/proc/sys/fs/fanotify/max_user_marks

This specifies an upper limit on the number of fanotify marks that can be created per real user ID. Prior to Linux kernel 5.13, the hardcoded limit was 8192 marks per group (not per user).

ERRORS

In addition to the usual errors for *read*(2), the following errors can occur when reading from the fanotify file descriptor:

EINVAL

The buffer is too small to hold the event.

EMFILE

The per-process limit on the number of open files has been reached. See the description of **RLIMIT_NOFILE** in *getrlimit*(2).

ENFILE

The system-wide limit on the total number of open files has been reached. See */proc/sys/fs/file-max* in *proc*(5).

ETXTBSY

This error is returned by *read*(2) if **O_RDWR** or **O_WRONLY** was specified in the *event_f_flags* argument when calling *fanotify_init*(2) and an event occurred for a monitored file that is currently being executed.

In addition to the usual errors for *write*(2), the following errors can occur when writing to the fanotify file descriptor:

EINVAL

Fanotify access permissions are not enabled in the kernel configuration or the value of *response* in the response structure is not valid.

ENOENT

The file descriptor *fd* in the response structure is not valid. This may occur when a response for the permission event has already been written.

STANDARDS

Linux.

HISTORY

The fanotify API was introduced in Linux 2.6.36 and enabled in Linux 2.6.37. fdinfo support was added in Linux 3.8.

NOTES

The fanotify API is available only if the kernel was built with the **CONFIG_FANOTIFY** configuration option enabled. In addition, fanotify permission handling is available only if the **CONFIG_FANOTIFY_ACCESS_PERMISSIONS** configuration option is enabled.

Limitations and caveats

Fanotify reports only events that a user-space program triggers through the filesystem API. As a result, it does not catch remote events that occur on network filesystems.

The fanotify API does not report file accesses and modifications that may occur because of *mmap*(2), *msync*(2), and *munmap*(2).

Events for directories are created only if the directory itself is opened, read, and closed. Adding, removing, or changing children of a marked directory does not create events for the monitored directory itself.

Fanotify monitoring of directories is not recursive: to monitor subdirectories under a directory, additional marks must be created. The **FAN_CREATE** event can be used for detecting when a subdirectory has been created under a marked directory. An additional mark must then be set on the newly created subdirectory. This approach is racy, because it can lose events that occurred inside the newly created subdirectory, before a mark is added on that subdirectory. Monitoring mounts offers the capability to monitor a whole directory tree in a race-free manner. Monitoring filesystems offers the capability to monitor changes made from any mount of a filesystem instance in a race-free manner.

The event queue can overflow. In this case, events are lost.

BUGS

Before Linux 3.19, *fallocate*(2) did not generate fanotify events. Since Linux 3.19, calls to *fallocate*(2) generate **FAN_MODIFY** events.

As of Linux 3.17, the following bugs exist:

- On Linux, a filesystem object may be accessible through multiple paths, for example, a part of a filesystem may be remounted using the −−*bind* option of *mount*(8)A listener that marked a mount will be notified only of events that were triggered for a filesystem object using the same mount. Any other event will pass unnoticed.
- When an event is generated, no check is made to see whether the user ID of the receiving process has authorization to read or write the file before passing a file descriptor for that file. This poses a security risk, when the **CAP_SYS_ADMIN** capability is set for programs executed by unprivileged users.
- If a call to *read*(2) processes multiple events from the fanotify queue and an error occurs, the return value will be the total length of the events successfully copied to the user-space buffer before the error occurred. The return value will not be −1, and *errno* will not be set. Thus, the reading application has no way to detect the error.

EXAMPLES

The two example programs below demonstrate the usage of the fanotify API.

Example program: fanotify_example.c

The first program is an example of fanotify being used with its event object information passed in the form of a file descriptor. The program marks the mount passed as a command-line argument and waits for events of type **FAN_OPEN_PERM** and **FAN_CLOSE_WRITE**. When a permission event occurs, a **FAN_ALLOW** response is given.

The following shell session shows an example of running this program. This session involved editing the file */home/user/temp/notes*. Before the file was opened, a **FAN_OPEN_PERM** event occurred. After the file was closed, a **FAN_CLOSE_WRITE** event occurred. Execution of the program ends when the user presses the ENTER key.

```
# ./fanotify_example /home
Press enter key to terminate.
Listening for events.
FAN_OPEN_PERM: File /home/user/temp/notes
FAN_CLOSE_WRITE: File /home/user/temp/notes

Listening for events stopped.
```

Program source: fanotify_example.c

```
#define _GNU_SOURCE     /* Needed to get O_LARGEFILE definition */
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fanotify.h>
#include <unistd.h>

/* Read all available fanotify events from the file descriptor 'fd'. */

static void
handle_events(int fd)
{
    const struct fanotify_event_metadata *metadata;
    struct fanotify_event_metadata buf[200];
    ssize_t len;
    char path[PATH_MAX];
    ssize_t path_len;
    char procfd_path[PATH_MAX];
    struct fanotify_response response;

    /* Loop while events can be read from fanotify file descriptor. */

    for (;;) {

        /* Read some events. */

        len = read(fd, buf, sizeof(buf));
        if (len == -1 && errno != EAGAIN) {
            perror("read");
            exit(EXIT_FAILURE);
        }

        /* Check if end of available data reached. */

        if (len <= 0)
            break;

        /* Point to the first event in the buffer. */

        metadata = buf;

        /* Loop over all events in the buffer. */

        while (FAN_EVENT_OK(metadata, len)) {

            /* Check that run-time and compile-time structures match. */
```

```
            if (metadata->vers != FANOTIFY_METADATA_VERSION) {
                fprintf(stderr,
                        "Mismatch of fanotify metadata version.\n");
                exit(EXIT_FAILURE);
            }

            /* metadata->fd contains either FAN_NOFD, indicating a
               queue overflow, or a file descriptor (a nonnegative
               integer). Here, we simply ignore queue overflow. */

            if (metadata->fd >= 0) {

                /* Handle open permission event. */

                if (metadata->mask & FAN_OPEN_PERM) {
                    printf("FAN_OPEN_PERM: ");

                    /* Allow file to be opened. */

                    response.fd = metadata->fd;
                    response.response = FAN_ALLOW;
                    write(fd, &response, sizeof(response));
                }

                /* Handle closing of writable file event. */

                if (metadata->mask & FAN_CLOSE_WRITE)
                    printf("FAN_CLOSE_WRITE: ");

                /* Retrieve and print pathname of the accessed file. */

                snprintf(procfd_path, sizeof(procfd_path),
                         "/proc/self/fd/%d", metadata->fd);
                path_len = readlink(procfd_path, path,
                                    sizeof(path) - 1);
                if (path_len == -1) {
                    perror("readlink");
                    exit(EXIT_FAILURE);
                }

                path[path_len] = '\0';
                printf("File %s\n", path);

                /* Close the file descriptor of the event. */

                close(metadata->fd);
            }

            /* Advance to next event. */

            metadata = FAN_EVENT_NEXT(metadata, len);
        }
    }
}

int
main(int argc, char *argv[])
{
```

```
char buf;
int fd, poll_num;
nfds_t nfds;
struct pollfd fds[2];

/* Check mount point is supplied. */

if (argc != 2) {
    fprintf(stderr, "Usage: %s MOUNT\n", argv[0]);
    exit(EXIT_FAILURE);
}

printf("Press enter key to terminate.\n");

/* Create the file descriptor for accessing the fanotify API. */

fd = fanotify_init(FAN_CLOEXEC | FAN_CLASS_CONTENT | FAN_NONBLOCK,
                   O_RDONLY | O_LARGEFILE);
if (fd == -1) {
    perror("fanotify_init");
    exit(EXIT_FAILURE);
}

/* Mark the mount for:
   - permission events before opening files
   - notification events after closing a write-enabled
     file descriptor. */

if (fanotify_mark(fd, FAN_MARK_ADD | FAN_MARK_MOUNT,
                  FAN_OPEN_PERM | FAN_CLOSE_WRITE, AT_FDCWD,
                  argv[1]) == -1) {
    perror("fanotify_mark");
    exit(EXIT_FAILURE);
}

/* Prepare for polling. */

nfds = 2;

fds[0].fd = STDIN_FILENO;       /* Console input */
fds[0].events = POLLIN;

fds[1].fd = fd;                 /* Fanotify input */
fds[1].events = POLLIN;

/* This is the loop to wait for incoming events. */

printf("Listening for events.\n");

while (1) {
    poll_num = poll(fds, nfds, -1);
    if (poll_num == -1) {
        if (errno == EINTR)     /* Interrupted by a signal */
            continue;           /* Restart poll() */

        perror("poll");         /* Unexpected error */
        exit(EXIT_FAILURE);
    }
```

```
            if (poll_num > 0) {
                if (fds[0].revents & POLLIN) {

                    /* Console input is available: empty stdin and quit. */

                    while (read(STDIN_FILENO, &buf, 1) > 0 && buf != '\n')
                        continue;
                    break;
                }

                if (fds[1].revents & POLLIN) {

                    /* Fanotify events are available. */

                    handle_events(fd);
                }
            }
        }

        printf("Listening for events stopped.\n");
        exit(EXIT_SUCCESS);
    }
```

Example program: fanotify_fid.c

The second program is an example of fanotify being used with a group that identifies objects by file handles. The program marks the filesystem object that is passed as a command-line argument and waits until an event of type **FAN_CREATE** has occurred. The event mask indicates which type of filesystem object—either a file or a directory—was created. Once all events have been read from the buffer and processed accordingly, the program simply terminates.

The following shell sessions show two different invocations of this program, with different actions performed on a watched object.

The first session shows a mark being placed on */home/user*. This is followed by the creation of a regular file, */home/user/testfile.txt*. This results in a **FAN_CREATE** event being generated and reported against the file's parent watched directory object and with the created file name. Program execution ends once all events captured within the buffer have been processed.

```
# ./fanotify_fid /home/user
Listening for events.
FAN_CREATE (file created):
        Directory /home/user has been modified.
        Entry 'testfile.txt' is not a subdirectory.
All events processed successfully. Program exiting.

$ touch /home/user/testfile.txt              # In another terminal
```

The second session shows a mark being placed on */home/user*. This is followed by the creation of a directory, */home/user/testdir*. This specific action results in a **FAN_CREATE** event being generated and is reported with the **FAN_ONDIR** flag set and with the created directory name.

```
# ./fanotify_fid /home/user
Listening for events.
FAN_CREATE | FAN_ONDIR (subdirectory created):
        Directory /home/user has been modified.
        Entry 'testdir' is a subdirectory.
All events processed successfully. Program exiting.

$ mkdir -p /home/user/testdir          # In another terminal
```

Program source: fanotify_fid.c

```
#define _GNU_SOURCE
```

```
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fanotify.h>
#include <unistd.h>

#define BUF_SIZE 256

int
main(int argc, char *argv[])
{
    int fd, ret, event_fd, mount_fd;
    ssize_t len, path_len;
    char path[PATH_MAX];
    char procfd_path[PATH_MAX];
    char events_buf[BUF_SIZE];
    struct file_handle *file_handle;
    struct fanotify_event_metadata *metadata;
    struct fanotify_event_info_fid *fid;
    const char *file_name;
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Invalid number of command line arguments.\n");
        exit(EXIT_FAILURE);
    }

    mount_fd = open(argv[1], O_DIRECTORY | O_RDONLY);
    if (mount_fd == -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    /* Create an fanotify file descriptor with FAN_REPORT_DFID_NAME as
       a flag so that program can receive fid events with directory
       entry name. */

    fd = fanotify_init(FAN_CLASS_NOTIF | FAN_REPORT_DFID_NAME, 0);
    if (fd == -1) {
        perror("fanotify_init");
        exit(EXIT_FAILURE);
    }

    /* Place a mark on the filesystem object supplied in argv[1]. */

    ret = fanotify_mark(fd, FAN_MARK_ADD | FAN_MARK_ONLYDIR,
                        FAN_CREATE | FAN_ONDIR,
                        AT_FDCWD, argv[1]);
    if (ret == -1) {
        perror("fanotify_mark");
        exit(EXIT_FAILURE);
    }

    printf("Listening for events.\n");
```

```
        /* Read events from the event queue into a buffer. */

        len = read(fd, events_buf, sizeof(events_buf));
        if (len == -1 && errno != EAGAIN) {
            perror("read");
            exit(EXIT_FAILURE);
        }

        /* Process all events within the buffer. */

        for (metadata = (struct fanotify_event_metadata *) events_buf;
                FAN_EVENT_OK(metadata, len);
                metadata = FAN_EVENT_NEXT(metadata, len)) {
            fid = (struct fanotify_event_info_fid *) (metadata + 1);
            file_handle = (struct file_handle *) fid->handle;

            /* Ensure that the event info is of the correct type. */

            if (fid->hdr.info_type == FAN_EVENT_INFO_TYPE_FID ||
                fid->hdr.info_type == FAN_EVENT_INFO_TYPE_DFID) {
                file_name = NULL;
            } else if (fid->hdr.info_type == FAN_EVENT_INFO_TYPE_DFID_NAME) {
                file_name = file_handle->f_handle +
                            file_handle->handle_bytes;
            } else {
                fprintf(stderr, "Received unexpected event info type.\n");
                exit(EXIT_FAILURE);
            }

            if (metadata->mask == FAN_CREATE)
                printf("FAN_CREATE (file created):\n");

            if (metadata->mask == (FAN_CREATE | FAN_ONDIR))
                printf("FAN_CREATE | FAN_ONDIR (subdirectory created):\n");

          /* metadata->fd is set to FAN_NOFD when the group identifies
             objects by file handles.  To obtain a file descriptor for
             the file object corresponding to an event you can use the
             struct file_handle that's provided within the
             fanotify_event_info_fid in conjunction with the
             open_by_handle_at(2) system call.  A check for ESTALE is
             done to accommodate for the situation where the file handle
             for the object was deleted prior to this system call. */

            event_fd = open_by_handle_at(mount_fd, file_handle, O_RDONLY);
            if (event_fd == -1) {
                if (errno == ESTALE) {
                    printf("File handle is no longer valid. "
                            "File has been deleted\n");
                    continue;
                } else {
                    perror("open_by_handle_at");
                    exit(EXIT_FAILURE);
                }
            }

            snprintf(procfd_path, sizeof(procfd_path), "/proc/self/fd/%d",
                    event_fd);
```

```
            /* Retrieve and print the path of the modified dentry. */

            path_len = readlink(procfd_path, path, sizeof(path) - 1);
            if (path_len == -1) {
                perror("readlink");
                exit(EXIT_FAILURE);
            }

            path[path_len] = '\0';
            printf("\tDirectory '%s' has been modified.\n", path);

            if (file_name) {
                ret = fstatat(event_fd, file_name, &sb, 0);
                if (ret == -1) {
                    if (errno != ENOENT) {
                        perror("fstatat");
                        exit(EXIT_FAILURE);
                    }
                    printf("\tEntry '%s' does not exist.\n", file_name);
                } else if ((sb.st_mode & S_IFMT) == S_IFDIR) {
                    printf("\tEntry '%s' is a subdirectory.\n", file_name);
                } else {
                    printf("\tEntry '%s' is not a subdirectory.\n",
                            file_name);
                }
            }

            /* Close associated file descriptor for this event. */

            close(event_fd);
        }

        printf("All events processed successfully. Program exiting.\n");
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fanotify_init(2), *fanotify_mark*(2), *inotify*(7)

NAME

feature_test_macros − feature test macros

DESCRIPTION

Feature test macros allow the programmer to control the definitions that are exposed by system header files when a program is compiled.

NOTE: In order to be effective, a feature test macro *must be defined before including any header files*. This can be done either in the compilation command (*cc −DMACRO=value*) or by defining the macro within the source code before including any headers. The requirement that the macro must be defined before including any header file exists because header files may freely include one another. Thus, for example, in the following lines, defining the **_GNU_SOURCE** macro may have no effect because the header <*abc.h*> itself includes <*xyz.h*> (POSIX explicitly allows this):

```
#include <abc.h>
#define _GNU_SOURCE
#include <xyz.h>
```

Some feature test macros are useful for creating portable applications, by preventing nonstandard definitions from being exposed. Other macros can be used to expose nonstandard definitions that are not exposed by default.

The precise effects of each of the feature test macros described below can be ascertained by inspecting the <*features.h*> header file. **Note**: applications do *not* need to directly include <*features.h*>; indeed, doing so is actively discouraged. See NOTES.

Specification of feature test macro requirements in manual pages

When a function requires that a feature test macro is defined, the manual page SYNOPSIS typically includes a note of the following form (this example from the *acct*(2) manual page):

#include <unistd.h>

int acct(const char * *filename***);**

```
Feature Test Macro Requirements for glibc (see
feature_test_macros(7)):
```

acct(): _BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

The **||** means that in order to obtain the declaration of *acct*(2) from <*unistd.h*>, *either* of the following macro definitions must be made before including any header files:

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE        /* or any value < 500 */
```

Alternatively, equivalent definitions can be included in the compilation command:

```
cc −D_BSD_SOURCE
cc −D_XOPEN_SOURCE           # Or any value < 500
```

Note that, as described below, **some feature test macros are defined by default**, so that it may not always be necessary to explicitly specify the feature test macro(s) shown in the SYNOPSIS.

In a few cases, manual pages use a shorthand for expressing the feature test macro requirements (this example from *readahead*(2)):

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>

ssize_t readahead(int fd, off_t *offset, size_t count);
```

This format is employed when the feature test macros ensure that the proper function declarations are visible, and the macros are not defined by default.

Feature test macros understood by glibc

The paragraphs below explain how feature test macros are handled in glibc 2.*x*, *x* > 0.

First, though, a summary of a few details for the impatient:

- The macros that you most likely need to use in modern source code are **_POSIX_C_SOURCE** (for definitions from various versions of POSIX.1), **_XOPEN_SOURCE** (for definitions from various versions of SUS), **_GNU_SOURCE** (for GNU and/or Linux specific stuff), and **_DEFAULT_SOURCE** (to get definitions that would normally be provided by default).
- Certain macros are defined with default values. Thus, although one or more macros may be indicated as being required in the SYNOPSIS of a man page, it may not be necessary to define them explicitly. Full details of the defaults are given later in this man page.
- Defining **_XOPEN_SOURCE** with a value of 600 or greater produces the same effects as defining **_POSIX_C_SOURCE** with a value of 200112L or greater. Where one sees

  ```
  _POSIX_C_SOURCE >= 200112L
  ```

  in the feature test macro requirements in the SYNOPSIS of a man page, it is implicit that the following has the same effect:

  ```
  _XOPEN_SOURCE >= 600
  ```
- Defining **_XOPEN_SOURCE** with a value of 700 or greater produces the same effects as defining **_POSIX_C_SOURCE** with a value of 200809L or greater. Where one sees

  ```
  _POSIX_C_SOURCE >= 200809L
  ```

  in the feature test macro requirements in the SYNOPSIS of a man page, it is implicit that the following has the same effect:

  ```
  _XOPEN_SOURCE >= 700
  ```

glibc understands the following feature test macros:

__STRICT_ANSI__
: ISO Standard C. This macro is implicitly defined by *gcc*(1) when invoked with, for example, the *−std=c99* or *−ansi* flag.

_POSIX_C_SOURCE
: Defining this macro causes header files to expose definitions as follows:
  - The value 1 exposes definitions conforming to POSIX.1-1990 and ISO C (1990).
  - The value 2 or greater additionally exposes definitions for POSIX.2-1992.
  - The value 199309L or greater additionally exposes definitions for POSIX.1b (real-time extensions).
  - The value 199506L or greater additionally exposes definitions for POSIX.1c (threads).
  - (Since glibc 2.3.3) The value 200112L or greater additionally exposes definitions corresponding to the POSIX.1-2001 base specification (excluding the XSI extension). This value also causes C95 (since glibc 2.12) and C99 (since glibc 2.10) features to be exposed (in other words, the equivalent of defining **_ISOC99_SOURCE**).
  - (Since glibc 2.10) The value 200809L or greater additionally exposes definitions corresponding to the POSIX.1-2008 base specification (excluding the XSI extension).

_POSIX_SOURCE
: Defining this obsolete macro with any value is equivalent to defining **_POSIX_C_SOURCE** with the value 1.

  Since this macro is obsolete, its usage is generally not documented when discussing feature test macro requirements in the man pages.

_XOPEN_SOURCE
: Defining this macro causes header files to expose definitions as follows:
  - Defining with any value exposes definitions conforming to POSIX.1, POSIX.2, and XPG4.
  - The value 500 or greater additionally exposes definitions for SUSv2 (UNIX 98).
  - (Since glibc 2.2) The value 600 or greater additionally exposes definitions for SUSv3 (UNIX 03; i.e., the POSIX.1-2001 base specification plus the XSI extension) and C99 definitions.

- (Since glibc 2.10) The value 700 or greater additionally exposes definitions for SUSv4 (i.e., the POSIX.1-2008 base specification plus the XSI extension).

If **__STRICT_ANSI__** is not defined, or **_XOPEN_SOURCE** is defined with a value greater than or equal to 500 *and* neither **_POSIX_SOURCE** nor **_POSIX_C_SOURCE** is explicitly defined, then the following macros are implicitly defined:

- **_POSIX_SOURCE** is defined with the value 1.
- **_POSIX_C_SOURCE** is defined, according to the value of **_XOPEN_SOURCE**:

  _XOPEN_SOURCE < 500
  : **_POSIX_C_SOURCE** is defined with the value 2.

  500 <= **_XOPEN_SOURCE** < 600
  : **_POSIX_C_SOURCE** is defined with the value 199506L.

  600 <= **_XOPEN_SOURCE** < 700
  : **_POSIX_C_SOURCE** is defined with the value 200112L.

  700 <= **_XOPEN_SOURCE** (since glibc 2.10)
  : **_POSIX_C_SOURCE** is defined with the value 200809L.

In addition, defining **_XOPEN_SOURCE** with a value of 500 or greater produces the same effects as defining **_XOPEN_SOURCE_EXTENDED**.

_XOPEN_SOURCE_EXTENDED
: If this macro is defined, *and* **_XOPEN_SOURCE** is defined, then expose definitions corresponding to the XPG4v2 (SUSv1) UNIX extensions (UNIX 95). Defining **_XOPEN_SOURCE** with a value of 500 or more also produces the same effect as defining **_XOPEN_SOURCE_EXTENDED**. Use of **_XOPEN_SOURCE_EXTENDED** in new source code should be avoided.

  Since defining **_XOPEN_SOURCE** with a value of 500 or more has the same effect as defining **_XOPEN_SOURCE_EXTENDED**, the latter (obsolete) feature test macro is generally not described in the SYNOPSIS in man pages.

_ISOC99_SOURCE (since glibc 2.1.3)
: Exposes declarations consistent with the ISO C99 standard.

  Earlier glibc 2.1.x versions recognized an equivalent macro named **_ISOC9X_SOURCE** (because the C99 standard had not then been finalized). Although the use of this macro is obsolete, glibc continues to recognize it for backward compatibility.

  Defining **_ISOC99_SOURCE** also exposes ISO C (1990) Amendment 1 ("C95") definitions. (The primary change in C95 was support for international character sets.)

  Invoking the C compiler with the option *−std=c99* produces the same effects as defining this macro.

_ISOC11_SOURCE (since glibc 2.16)
: Exposes declarations consistent with the ISO C11 standard. Defining this macro also enables C99 and C95 features (like **_ISOC99_SOURCE**).

  Invoking the C compiler with the option *−std=c11* produces the same effects as defining this macro.

_LARGEFILE64_SOURCE
: Expose definitions for the alternative API specified by the LFS (Large File Summit) as a "transitional extension" to the Single UNIX Specification. (See .) The alternative API consists of a set of new objects (i.e., functions and types) whose names are suffixed with "64" (e.g., *off64_t* versus *off_t*, **lseek64**() versus **lseek**(), etc.). New programs should not employ this macro; instead *_FILE_OFFSET_BITS=64* should be employed.

_LARGEFILE_SOURCE
: This macro was historically used to expose certain functions (specifically *fseeko*(3) and *ftello*(3)) that address limitations of earlier APIs (**fseek**(3) and *ftell*(3)) that use *long* for file offsets. This macro is implicitly defined if **_XOPEN_SOURCE** is defined with a value greater than or equal to 500. New programs should not employ this macro; defining

_XOPEN_SOURCE as just described or defining **_FILE_OFFSET_BITS** with the value 64 is the preferred mechanism to achieve the same result.

_FILE_OFFSET_BITS
Defining this macro with the value 64 automatically converts references to 32-bit functions and data types related to file I/O and filesystem operations into references to their 64-bit counterparts. This is useful for performing I/O on large files (> 2 Gigabytes) on 32-bit systems. It is also useful when calling functions like *copy_file_range*(2) that were added more recently and that come only in 64-bit flavors. (Defining this macro permits correctly written programs to use large files with only a recompilation being required.)

64-bit systems naturally permit file sizes greater than 2 Gigabytes, and on those systems this macro has no effect.

_TIME_BITS
Defining this macro with the value 64 changes the width of *time_t*(3type) to 64-bit which allows handling of timestamps beyond 2038. It is closely related to **_FILE_OFFSET_BITS** and depending on implementation, may require it set. This macro is available as of glibc 2.34.

_BSD_SOURCE (deprecated since glibc 2.20)
Defining this macro with any value causes header files to expose BSD-derived definitions.

In glibc versions up to and including 2.18, defining this macro also causes BSD definitions to be preferred in some situations where standards conflict, unless one or more of **_SVID_SOURCE**, **_POSIX_SOURCE**, **_POSIX_C_SOURCE**, **_XOPEN_SOURCE**, **_XOPEN_SOURCE_EXTENDED**, or **_GNU_SOURCE** is defined, in which case BSD definitions are disfavored. Since glibc 2.19, **_BSD_SOURCE** no longer causes BSD definitions to be preferred in case of conflicts.

Since glibc 2.20, this macro is deprecated. It now has the same effect as defining **_DEFAULT_SOURCE**, but generates a compile-time warning (unless **_DEFAULT_SOURCE** is also defined). Use **_DEFAULT_SOURCE** instead. To allow code that requires **_BSD_SOURCE** in glibc 2.19 and earlier and **_DEFAULT_SOURCE** in glibc 2.20 and later to compile without warnings, define *both* **_BSD_SOURCE** and **_DEFAULT_SOURCE**.

_SVID_SOURCE (deprecated since glibc 2.20)
Defining this macro with any value causes header files to expose System V-derived definitions. (SVID == System V Interface Definition; see *standards*(7).)

Since glibc 2.20, this macro is deprecated in the same fashion as **_BSD_SOURCE**.

_DEFAULT_SOURCE (since glibc 2.19)
This macro can be defined to ensure that the "default" definitions are provided even when the defaults would otherwise be disabled, as happens when individual macros are explicitly defined, or the compiler is invoked in one of its "standard" modes (e.g., *cc −std=c99*). Defining **_DEFAULT_SOURCE** without defining other individual macros or invoking the compiler in one of its "standard" modes has no effect.

The "default" definitions comprise those required by POSIX.1-2008 and ISO C99, as well as various definitions originally derived from BSD and System V. On glibc 2.19 and earlier, these defaults were approximately equivalent to explicitly defining the following:

```
cc −D_BSD_SOURCE −D_SVID_SOURCE −D_POSIX_C_SOURCE=200809
```

_ATFILE_SOURCE (since glibc 2.4)
Defining this macro with any value causes header files to expose declarations of a range of functions with the suffix "at"; see *openat*(2). Since glibc 2.10, this macro is also implicitly defined if **_POSIX_C_SOURCE** is defined with a value greater than or equal to 200809L.

_GNU_SOURCE
Defining this macro (with any value) implicitly defines **_ATFILE_SOURCE**, **_LARGEFILE64_SOURCE**, **_ISOC99_SOURCE**, **_XOPEN_SOURCE_EXTENDED**, **_POSIX_SOURCE**, **_POSIX_C_SOURCE** with the value 200809L (200112L before glibc 2.10; 199506L before glibc 2.5; 199309L before glibc 2.1) and **_XOPEN_SOURCE** with the value 700 (600 before glibc 2.10; 500 before glibc 2.2). In addition, various GNU-specific extensions are also exposed.

Since glibc 2.19, defining **_GNU_SOURCE** also has the effect of implicitly defining **_DEFAULT_SOURCE**. Before glibc 2.20, defining **_GNU_SOURCE** also had the effect of implicitly defining **_BSD_SOURCE** and **_SVID_SOURCE**.

_REENTRANT
Historically, on various C libraries it was necessary to define this macro in all multithreaded code. (Some C libraries may still require this.) In glibc, this macro also exposed definitions of certain reentrant functions.

However, glibc has been thread-safe by default for many years; since glibc 2.3, the only effect of defining **_REENTRANT** has been to enable one or two of the same declarations that are also enabled by defining **_POSIX_C_SOURCE** with a value of 199606L or greater.

_REENTRANT is now obsolete. In glibc 2.25 and later, defining **_REENTRANT** is equivalent to defining **_POSIX_C_SOURCE** with the value 199606L. If a higher POSIX conformance level is selected by any other means (such as **_POSIX_C_SOURCE** itself, **_XOPEN_SOURCE**, **_DEFAULT_SOURCE**, or **_GNU_SOURCE**), then defining **_REENTRANT** has no effect.

This macro is automatically defined if one compiles with *cc −pthread*.

_THREAD_SAFE
Synonym for the (deprecated) **_REENTRANT**, provided for compatibility with some other implementations.

_FORTIFY_SOURCE (since glibc 2.3.4)
Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions (for example, *memcpy*(3), *memset*(3), *stpcpy*(3), *strcpy*(3), *strncpy*(3), *strcat*(3), *strncat*(3), *sprintf*(3), *snprintf*(3), *vsprintf*(3), *vsnprintf*(3), *gets*(3), and wide character variants thereof). For some functions, argument consistency is checked; for example, a check is made that *open*(2) has been supplied with a *mode* argument when the specified flags include **O_CREAT**. Not all problems are detected, just some common cases.

If **_FORTIFY_SOURCE** is set to 1, with compiler optimization level 1 (*gcc −O1*) and above, checks that shouldn't change the behavior of conforming programs are performed. With **_FORTIFY_SOURCE** set to 2, some more checking is added, but some conforming programs might fail.

Some of the checks can be performed at compile time (via macros logic implemented in header files), and result in compiler warnings; other checks take place at run time, and result in a run-time error if the check fails.

With **_FORTIFY_SOURCE** set to 3, additional checking is added to intercept some function calls used with an argument of variable size where the compiler can deduce an upper bound for its value. For example, a program where *malloc*(3)'s size argument is variable can now be fortified.

Use of this macro requires compiler support, available since gcc 4.0 and clang 2.6. Use of **_FORTIFY_SOURCE** set to 3 requires gcc 12.0 or later, or clang 9.0 or later, in conjunction with glibc 2.33 or later.

Default definitions, implicit definitions, and combining definitions
If no feature test macros are explicitly defined, then the following feature test macros are defined by default: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_SVID_SOURCE** (in glibc 2.19 and earlier), **_DEFAULT_SOURCE** (since glibc 2.19), **_POSIX_SOURCE**, and **_POSIX_C_SOURCE**=200809L (200112L before glibc 2.10; 199506L before glibc 2.4; 199309L before glibc 2.1).

If any of **__STRICT_ANSI__**, **_ISOC99_SOURCE**, **_ISOC11_SOURCE** (since glibc 2.18), **_POSIX_SOURCE**, **_POSIX_C_SOURCE**, **_XOPEN_SOURCE**, **_XOPEN_SOURCE_EXTENDED** (in glibc 2.11 and earlier), **_BSD_SOURCE** (in glibc 2.19 and earlier), or **_SVID_SOURCE** (in glibc 2.19 and earlier) is explicitly defined, then **_BSD_SOURCE**, **_SVID_SOURCE**, and **_DEFAULT_SOURCE** are not defined by default.

If **_POSIX_SOURCE** and **_POSIX_C_SOURCE** are not explicitly defined, and either **__STRICT_ANSI__** is not defined or **_XOPEN_SOURCE** is defined with a value of 500 or more,

then

- **_POSIX_SOURCE** is defined with the value 1; and
- **_POSIX_C_SOURCE** is defined with one of the following values:
  - 2, if **_XOPEN_SOURCE** is defined with a value less than 500;
  - 199506L, if **_XOPEN_SOURCE** is defined with a value greater than or equal to 500 and less than 600; or
  - (since glibc 2.4) 200112L, if **_XOPEN_SOURCE** is defined with a value greater than or equal to 600 and less than 700.
  - (Since glibc 2.10) 200809L, if **_XOPEN_SOURCE** is defined with a value greater than or equal to 700.
  - Older versions of glibc do not know about the values 200112L and 200809L for **_POSIX_C_SOURCE**, and the setting of this macro will depend on the glibc version.
  - If **_XOPEN_SOURCE** is undefined, then the setting of **_POSIX_C_SOURCE** depends on the glibc version: 199506L, before glibc 2.4; 200112L, since glibc 2.4 to glibc 2.9; and 200809L, since glibc 2.10.

Multiple macros can be defined; the results are additive.

STANDARDS

POSIX.1 specifies **_POSIX_C_SOURCE**, **_POSIX_SOURCE**, and **_XOPEN_SOURCE**.

_FILE_OFFSET_BITS is not specified by any standard, but is employed on some other implementations.

_BSD_SOURCE, **_SVID_SOURCE**, **_DEFAULT_SOURCE**, **_ATFILE_SOURCE**, **_GNU_SOURCE**, **_FORTIFY_SOURCE**, **_REENTRANT**, and **_THREAD_SAFE** are specific to glibc.

HISTORY

_XOPEN_SOURCE_EXTENDED was specified by XPG4v2 (aka SUSv1), but is not present in SUSv2 and later.

NOTES

<*features.h*> is a Linux/glibc-specific header file. Other systems have an analogous file, but typically with a different name. This header file is automatically included by other header files as required: it is not necessary to explicitly include it in order to employ feature test macros.

According to which of the above feature test macros are defined, <*features.h*> internally defines various other macros that are checked by other glibc header files. These macros have names prefixed by two underscores (e.g., **__USE_MISC**). Programs should *never* define these macros directly: instead, the appropriate feature test macro(s) from the list above should be employed.

EXAMPLES

The program below can be used to explore how the various feature test macros are set depending on the glibc version and what feature test macros are explicitly set. The following shell session, on a system with glibc 2.10, shows some examples of what we would see:

```
$ cc ftm.c
$ ./a.out
_POSIX_SOURCE defined
_POSIX_C_SOURCE defined: 200809L
_BSD_SOURCE defined
_SVID_SOURCE defined
_ATFILE_SOURCE defined
$ cc -D_XOPEN_SOURCE=500 ftm.c
$ ./a.out
_POSIX_SOURCE defined
_POSIX_C_SOURCE defined: 199506L
_XOPEN_SOURCE defined: 500
$ cc -D_GNU_SOURCE ftm.c
$ ./a.out
```

```
    _POSIX_SOURCE defined
    _POSIX_C_SOURCE defined: 200809L
    _ISOC99_SOURCE defined
    _XOPEN_SOURCE defined: 700
    _XOPEN_SOURCE_EXTENDED defined
    _LARGEFILE64_SOURCE defined
    _BSD_SOURCE defined
    _SVID_SOURCE defined
    _ATFILE_SOURCE defined
    _GNU_SOURCE defined
```

Program source

```
/* ftm.c */

#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
#ifdef _POSIX_SOURCE
    printf("_POSIX_SOURCE defined\n");
#endif

#ifdef _POSIX_C_SOURCE
    printf("_POSIX_C_SOURCE defined: %jdL\n",
            (intmax_t) _POSIX_C_SOURCE);
#endif

#ifdef _ISOC99_SOURCE
    printf("_ISOC99_SOURCE defined\n");
#endif

#ifdef _ISOC11_SOURCE
    printf("_ISOC11_SOURCE defined\n");
#endif

#ifdef _XOPEN_SOURCE
    printf("_XOPEN_SOURCE defined: %d\n", _XOPEN_SOURCE);
#endif

#ifdef _XOPEN_SOURCE_EXTENDED
    printf("_XOPEN_SOURCE_EXTENDED defined\n");
#endif

#ifdef _LARGEFILE64_SOURCE
    printf("_LARGEFILE64_SOURCE defined\n");
#endif

#ifdef _FILE_OFFSET_BITS
    printf("_FILE_OFFSET_BITS defined: %d\n", _FILE_OFFSET_BITS);
#endif

#ifdef _TIME_BITS
    printf("_TIME_BITS defined: %d\n", _TIME_BITS);
#endif
```

```
    #ifdef _BSD_SOURCE
        printf("_BSD_SOURCE defined\n");
    #endif

    #ifdef _SVID_SOURCE
        printf("_SVID_SOURCE defined\n");
    #endif

    #ifdef _DEFAULT_SOURCE
        printf("_DEFAULT_SOURCE defined\n");
    #endif

    #ifdef _ATFILE_SOURCE
        printf("_ATFILE_SOURCE defined\n");
    #endif

    #ifdef _GNU_SOURCE
        printf("_GNU_SOURCE defined\n");
    #endif

    #ifdef _REENTRANT
        printf("_REENTRANT defined\n");
    #endif

    #ifdef _THREAD_SAFE
        printf("_THREAD_SAFE defined\n");
    #endif

    #ifdef _FORTIFY_SOURCE
        printf("_FORTIFY_SOURCE defined\n");
    #endif

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

libc(7), *standards*(7), *system_data_types*(7)

The section "Feature Test Macros" under *info libc*.

/usr/include/features.h

NAME

fifo − first-in first-out special file, named pipe

DESCRIPTION

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem. Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.

The kernel maintains exactly one pipe object for each FIFO special file that is opened by at least one process. The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

A process can open a FIFO in nonblocking mode. In this case, opening for read-only succeeds even if no one has opened on the write side yet and opening for write-only fails with **ENXIO** (no such device or address) unless the other end has already been opened.

Under Linux, opening a FIFO for read and write will succeed both in blocking and nonblocking mode. POSIX leaves this behavior undefined. This can be used to open a FIFO for writing while there are no readers available. A process that uses both ends of the connection in order to communicate with itself should be very careful to avoid deadlocks.

NOTES

For details of the semantics of I/O on FIFOs, see *pipe*(7).

When a process tries to write to a FIFO that is not opened for read on the other side, the process is sent a **SIGPIPE** signal.

FIFO special files can be created by *mkfifo*(3), and are indicated by *ls −l* with the file type 'p'.

SEE ALSO

mkfifo(1), *open*(2), *pipe*(2), *sigaction*(2), *signal*(2), *socketpair*(2), *mkfifo*(3), *pipe*(7)

NAME

futex − fast user-space locking

SYNOPSIS

#include <linux/futex.h>

DESCRIPTION

The Linux kernel provides futexes ("Fast user-space mutexes") as a building block for fast user-space locking and semaphores. Futexes are very basic and lend themselves well for building higher-level locking abstractions such as mutexes, condition variables, read-write locks, barriers, and semaphores.

Most programmers will in fact not be using futexes directly but will instead rely on system libraries built on them, such as the Native POSIX Thread Library (NPTL) (see *pthreads*(7)).

A futex is identified by a piece of memory which can be shared between processes or threads. In these different processes, the futex need not have identical addresses. In its bare form, a futex has semaphore semantics; it is a counter that can be incremented and decremented atomically; processes can wait for the value to become positive.

Futex operation occurs entirely in user space for the noncontended case. The kernel is involved only to arbitrate the contended case. As any sane design will strive for noncontention, futexes are also optimized for this situation.

In its bare form, a futex is an aligned integer which is touched only by atomic assembler instructions. This integer is four bytes long on all platforms. Processes can share this integer using *mmap*(2), via shared memory segments, or because they share memory space, in which case the application is commonly called multithreaded.

Semantics

Any futex operation starts in user space, but it may be necessary to communicate with the kernel using the *futex*(2) system call.

To "up" a futex, execute the proper assembler instructions that will cause the host CPU to atomically increment the integer. Afterward, check if it has in fact changed from 0 to 1, in which case there were no waiters and the operation is done. This is the noncontended case which is fast and should be common.

In the contended case, the atomic increment changed the counter from −1 (or some other negative number). If this is detected, there are waiters. User space should now set the counter to 1 and instruct the kernel to wake up any waiters using the **FUTEX_WAKE** operation.

Waiting on a futex, to "down" it, is the reverse operation. Atomically decrement the counter and check if it changed to 0, in which case the operation is done and the futex was uncontended. In all other circumstances, the process should set the counter to −1 and request that the kernel wait for another process to up the futex. This is done using the **FUTEX_WAIT** operation.

The *futex*(2) system call can optionally be passed a timeout specifying how long the kernel should wait for the futex to be upped. In this case, semantics are more complex and the programmer is referred to *futex*(2) for more details. The same holds for asynchronous futex waiting.

VERSIONS

Initial futex support was merged in Linux 2.5.7 but with different semantics from those described above. Current semantics are available from Linux 2.5.40 onward.

NOTES

To reiterate, bare futexes are not intended as an easy-to-use abstraction for end users. Implementors are expected to be assembly literate and to have read the sources of the futex user-space library referenced below.

This man page illustrates the most common use of the *futex*(2) primitives; it is by no means the only one.

SEE ALSO

clone(2), *futex*(2), *get_robust_list*(2), *set_robust_list*(2), *set_tid_address*(2), *pthreads*(7)

Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux (proceedings of the Ottawa Linux Symposium 2002), futex example library, futex-*.tar.bz2 .

NAME

glob − globbing pathnames

DESCRIPTION

Long ago, in UNIX V6, there was a program */etc/glob* that would expand wildcard patterns. Soon afterward this became a shell built-in.

These days there is also a library routine *glob*(3) that will perform this function for a user program.

The rules are as follows (POSIX.2, 3.13).

Wildcard matching

A string is a wildcard pattern if it contains one of the characters '?', '*', or '['. Globbing is the operation that expands a wildcard pattern into the list of pathnames matching the pattern. Matching is defined by:

A '?' (not between brackets) matches any single character.

A '*' (not between brackets) matches any string, including the empty string.

Character classes

An expression "*[...]*" where the first character after the leading '[' is not an '!' matches a single character, namely any of the characters enclosed by the brackets. The string enclosed by the brackets cannot be empty; therefore ']' can be allowed between the brackets, provided that it is the first character. (Thus, "*[][!]*" matches the three characters '[', ']', and '!'.)

Ranges

There is one special convention: two characters separated by '−' denote a range. (Thus, "*[A−Fa−f0−9]*" is equivalent to "*[ABCDEFabcdef0123456789]*".) One may include '−' in its literal meaning by making it the first or last character between the brackets. (Thus, "*[]−]*" matches just the two characters ']' and '−', and "*[−−0]*" matches the three characters '−', '.', and '0', since '/' cannot be matched.)

Complementation

An expression "*[!...]*" matches a single character, namely any character that is not matched by the expression obtained by removing the first '!' from it. (Thus, "*[!]a−]*" matches any single character except ']', 'a', and '−'.)

One can remove the special meaning of '?', '*', and '[' by preceding them by a backslash, or, in case this is part of a shell command line, enclosing them in quotes. Between brackets these characters stand for themselves. Thus, "*[[?*\]*" matches the four characters '[', '?', '*', and '\'.

Pathnames

Globbing is applied on each of the components of a pathname separately. A '/' in a pathname cannot be matched by a '?' or '*' wildcard, or by a range like "*[.−0]*". A range containing an explicit '/' character is syntactically incorrect. (POSIX requires that syntactically incorrect patterns are left unchanged.)

If a filename starts with a '.', this character must be matched explicitly. (Thus, *rm* * will not remove .profile, and *tar c* * will not archive all your files; *tar c* . is better.)

Empty lists

The nice and simple rule given above: "expand a wildcard pattern into the list of matching pathnames" was the original UNIX definition. It allowed one to have patterns that expand into an empty list, as in

```
xv −wait 0 *.gif *.jpg
```

where perhaps no *.gif files are present (and this is not an error). However, POSIX requires that a wildcard pattern is left unchanged when it is syntactically incorrect, or the list of matching pathnames is empty. With *bash* one can force the classical behavior using this command:

```
shopt −s nullglob
```

(Similar problems occur elsewhere. For example, where old scripts have

```
rm `find . −name "*~"`
```

new scripts require

```
rm −f nosuchfile `find . −name "*~"`
```

to avoid error messages from *rm* called with an empty argument list.)

NOTES

Regular expressions

Note that wildcard patterns are not regular expressions, although they are a bit similar. First of all, they match filenames, rather than text, and secondly, the conventions are not the same: for example, in a regular expression '*' means zero or more copies of the preceding thing.

Now that regular expressions have bracket expressions where the negation is indicated by a '^', POSIX has declared the effect of a wildcard pattern "*[^...]*" to be undefined.

Character classes and internationalization

Of course ranges were originally meant to be ASCII ranges, so that "*[−%]*" stands for "*[!"#$%]*" and "*[a−z]*" stands for "any lowercase letter". Some UNIX implementations generalized this so that a range X−Y stands for the set of characters with code between the codes for X and for Y. However, this requires the user to know the character coding in use on the local system, and moreover, is not convenient if the collating sequence for the local alphabet differs from the ordering of the character codes. Therefore, POSIX extended the bracket notation greatly, both for wildcard patterns and for regular expressions. In the above we saw three types of items that can occur in a bracket expression: namely (i) the negation, (ii) explicit single characters, and (iii) ranges. POSIX specifies ranges in an internationally more useful way and adds three more types:

(iii) Ranges X−Y comprise all characters that fall between X and Y (inclusive) in the current collating sequence as defined by the **LC_COLLATE** category in the current locale.

(iv) Named character classes, like

[:alnum:] [:alpha:] [:blank:] [:cntrl:]
[:digit:] [:graph:] [:lower:] [:print:]
[:punct:] [:space:] [:upper:] [:xdigit:]

so that one can say "*[[:lower:]]*" instead of "*[a−z]*", and have things work in Denmark, too, where there are three letters past 'z' in the alphabet. These character classes are defined by the **LC_CTYPE** category in the current locale.

(v) Collating symbols, like "*[.ch.]*" or "*[.a-acute.]*", where the string between "*[.*" and ".*]*" is a collating element defined for the current locale. Note that this may be a multicharacter element.

(vi) Equivalence class expressions, like "*[=a=]*", where the string between "*[=*" and "*=]*" is any collating element from its equivalence class, as defined for the current locale. For example, "*[[=a=]]*" might be equivalent to "*[aáàäâ]*", that is, to "*[a[.a-acute.][.a-grave.][.a-umlaut.][.a-circumflex.]]*".

SEE ALSO

sh(1), *fnmatch*(3), *glob*(3), *locale*(7), *regex*(7)

NAME

hier − description of the filesystem hierarchy

DESCRIPTION

A typical Linux system has, among others, the following directories:

/ This is the root directory. This is where the whole tree starts.

/bin This directory contains executable programs which are needed in single user mode and to bring the system up or repair it.

/boot Contains static files for the boot loader. This directory holds only the files which are needed during the boot process. The map installer and configuration files should go to */sbin* and */etc*. The operating system kernel (initrd for example) must be located in either / or */boot*.

/dev Special or device files, which refer to physical devices. See *mknod*(1)

/etc Contains configuration files which are local to the machine. Some larger software packages, like X11, can have their own subdirectories below */etc*. Site-wide configuration files may be placed here or in */usr/etc*. Nevertheless, programs should always look for these files in */etc* and you may have links for these files to */usr/etc*.

/etc/opt
Host-specific configuration files for add-on applications installed in */opt*.

/etc/sgml
This directory contains the configuration files for SGML (optional).

/etc/skel
When a new user account is created, files from this directory are usually copied into the user's home directory.

/etc/X11
Configuration files for the X11 window system (optional).

/etc/xml
This directory contains the configuration files for XML (optional).

/home On machines with home directories for users, these are usually beneath this directory, directly or not. The structure of this directory depends on local administration decisions (optional).

/lib This directory should hold those shared libraries that are necessary to boot the system and to run the commands in the root filesystem.

/lib<qual>
These directories are variants of */lib* on system which support more than one binary format requiring separate libraries (optional).

/lib/modules
Loadable kernel modules (optional).

/lost+found
This directory contains items lost in the filesystem. These items are usually chunks of files mangled as a consequence of a faulty disk or a system crash.

/media This directory contains mount points for removable media such as CD and DVD disks or USB sticks. On systems where more than one device exists for mounting a certain type of media, mount directories can be created by appending a digit to the name of those available above starting with '0', but the unqualified name must also exist.

/media/floppy[1−9]
Floppy drive (optional).

/media/cdrom[1−9]
CD-ROM drive (optional).

/media/cdrecorder[1−9]
CD writer (optional).

/media/zip[1−9]
: Zip drive (optional).

/media/usb[1−9]
: USB drive (optional).

/mnt
: This directory is a mount point for a temporarily mounted filesystem. In some distributions, */mnt* contains subdirectories intended to be used as mount points for several temporary filesystems.

/opt
: This directory should contain add-on packages that contain static files.

/proc
: This is a mount point for the *proc* filesystem, which provides information about running processes and the kernel. This pseudo-filesystem is described in more detail in *proc*(5).

/root
: This directory is usually the home directory for the root user (optional).

/run
: This directory contains information which describes the system since it was booted. Once this purpose was served by */var/run* and programs may continue to use it.

/sbin
: Like */bin*, this directory holds commands needed to boot the system, but which are usually not executed by normal users.

/srv
: This directory contains site-specific data that is served by this system.

/sys
: This is a mount point for the sysfs filesystem, which provides information about the kernel like */proc*, but better structured, following the formalism of kobject infrastructure.

/tmp
: This directory contains temporary files which may be deleted with no notice, such as by a regular job or at system boot up.

/usr
: This directory is usually mounted from a separate partition. It should hold only shareable, read-only data, so that it can be mounted by various machines running Linux.

/usr/X11R6
: The X−Window system, version 11 release 6 (present in FHS 2.3, removed in FHS 3.0).

/usr/X11R6/bin
: Binaries which belong to the X−Window system; often, there is a symbolic link from the more traditional */usr/bin/X11* to here.

/usr/X11R6/lib
: Data files associated with the X−Window system.

/usr/X11R6/lib/X11
: These contain miscellaneous files needed to run X; Often, there is a symbolic link from */usr/lib/X11* to this directory.

/usr/X11R6/include/X11
: Contains include files needed for compiling programs using the X11 window system. Often, there is a symbolic link from */usr/include/X11* to this directory.

/usr/bin
: This is the primary directory for executable programs. Most programs executed by normal users which are not needed for booting or for repairing the system and which are not installed locally should be placed in this directory.

/usr/bin/mh
: Commands for the MH mail handling system (optional).

/usr/bin/X11
: This is the traditional place to look for X11 executables; on Linux, it usually is a symbolic link to */usr/X11R6/bin*.

/usr/dict
: Replaced by */usr/share/dict*.

/usr/doc
: Replaced by */usr/share/doc*.

/usr/etc
: Site-wide configuration files to be shared between several machines may be stored in this directory. However, commands should always reference those files using the */etc* directory. Links from files in */etc* should point to the appropriate files in */usr/etc*.

/usr/games
: Binaries for games and educational programs (optional).

/usr/include
: Include files for the C compiler.

/usr/include/bsd
: BSD compatibility include files (optional).

/usr/include/X11
: Include files for the C compiler and the X−Window system. This is usually a symbolic link to */usr/X11R6/include/X11*.

/usr/include/asm
: Include files which declare some assembler functions. This used to be a symbolic link to */usr/src/linux/include/asm*.

/usr/include/linux
: This contains information which may change from system release to system release and used to be a symbolic link to */usr/src/linux/include/linux* to get at operating-system-specific information.

  (Note that one should have include files there that work correctly with the current libc and in user space. However, Linux kernel source is not designed to be used with user programs and does not know anything about the libc you are using. It is very likely that things will break if you let */usr/include/asm* and */usr/include/linux* point at a random kernel tree. Debian systems don't do this and use headers from a known good kernel version, provided in the libc*−dev package.)

/usr/include/g++
: Include files to use with the GNU C++ compiler.

/usr/lib Object libraries, including dynamic libraries, plus some executables which usually are not invoked directly. More complicated programs may have whole subdirectories there.

/usr/libexec
: Directory contains binaries for internal use only and they are not meant to be executed directly by users shell or scripts.

/usr/lib<qual>
: These directories are variants of */usr/lib* on system which support more than one binary format requiring separate libraries, except that the symbolic link */usr/lib*qual*/X11* is not required (optional).

/usr/lib/X11
: The usual place for data files associated with X programs, and configuration files for the X system itself. On Linux, it usually is a symbolic link to */usr/X11R6/lib/X11*.

/usr/lib/gcc−lib
: contains executables and include files for the GNU C compiler, *gcc*(1)

/usr/lib/groff
: Files for the GNU groff document formatting system.

/usr/lib/uucp
: Files for *uucp*(1)

/usr/local
: This is where programs which are local to the site typically go.

/usr/local/bin
: Binaries for programs local to the site.

/usr/local/doc
: Local documentation.

/usr/local/etc
: Configuration files associated with locally installed programs.

/usr/local/games
: Binaries for locally installed games.

/usr/local/lib
: Files associated with locally installed programs.

/usr/local/lib<qual>
: These directories are variants of */usr/local/lib* on system which support more than one binary format requiring separate libraries (optional).

/usr/local/include
: Header files for the local C compiler.

/usr/local/info
: Info pages associated with locally installed programs.

/usr/local/man
: Man pages associated with locally installed programs.

/usr/local/sbin
: Locally installed programs for system administration.

/usr/local/share
: Local application data that can be shared among different architectures of the same OS.

/usr/local/src
: Source code for locally installed software.

/usr/man
: Replaced by */usr/share/man*.

/usr/sbin
: This directory contains program binaries for system administration which are not essential for the boot process, for mounting */usr*, or for system repair.

/usr/share
: This directory contains subdirectories with specific application data, that can be shared among different architectures of the same OS. Often one finds stuff here that used to live in */usr/doc* or */usr/lib* or */usr/man*.

/usr/share/color
: Contains color management information, like International Color Consortium (ICC) Color profiles (optional).

/usr/share/dict
: Contains the word lists used by spell checkers (optional).

/usr/share/dict/words
: List of English words (optional).

/usr/share/doc
: Documentation about installed programs (optional).

/usr/share/games
: Static data files for games in */usr/games* (optional).

/usr/share/info
: Info pages go here (optional).

/usr/share/locale
: Locale information goes here (optional).

/usr/share/man
: Manual pages go here in subdirectories according to the man page sections.

*/usr/share/man/*locale*/man[1−9]*
: These directories contain manual pages for the specific locale in source code form. Systems which use a unique language and code set for all manual pages may omit the <locale> substring.

/usr/share/misc
: Miscellaneous data that can be shared among different architectures of the same OS.

/usr/share/nls
: The message catalogs for native language support go here (optional).

/usr/share/ppd
: Postscript Printer Definition (PPD) files (optional).

/usr/share/sgml
: Files for SGML (optional).

/usr/share/sgml/docbook
: DocBook DTD (optional).

/usr/share/sgml/tei
: TEI DTD (optional).

/usr/share/sgml/html
: HTML DTD (optional).

/usr/share/sgml/mathml
: MathML DTD (optional).

/usr/share/terminfo
: The database for terminfo (optional).

/usr/share/tmac
: Troff macros that are not distributed with groff (optional).

/usr/share/xml
: Files for XML (optional).

/usr/share/xml/docbook
: DocBook DTD (optional).

/usr/share/xml/xhtml
: XHTML DTD (optional).

/usr/share/xml/mathml
: MathML DTD (optional).

/usr/share/zoneinfo
: Files for timezone information (optional).

/usr/src
: Source files for different parts of the system, included with some packages for reference purposes. Don't work here with your own projects, as files below /usr should be read-only except when installing software (optional).

/usr/src/linux
: This was the traditional place for the kernel source. Some distributions put here the source for the default kernel they ship. You should probably use another directory when building your own kernel.

/usr/tmp
: Obsolete. This should be a link to */var/tmp*. This link is present only for compatibility reasons and shouldn't be used.

/var This directory contains files which may change in size, such as spool and log files.

/var/account
: Process accounting logs (optional).

/var/adm
This directory is superseded by */var/log* and should be a symbolic link to */var/log*.

/var/backups
Reserved for historical reasons.

/var/cache
Data cached for programs.

/var/cache/fonts
Locally generated fonts (optional).

/var/cache/man
Locally formatted man pages (optional).

/var/cache/www
WWW proxy or cache data (optional).

/var/cache/<package>
Package specific cache data (optional).

/var/catman/cat[1−9] or */var/cache/man/cat[1−9]*
These directories contain preformatted manual pages according to their man page section. (The use of preformatted manual pages is deprecated.)

/var/crash
System crash dumps (optional).

/var/cron
Reserved for historical reasons.

/var/games
Variable game data (optional).

/var/lib Variable state information for programs.

/var/lib/color
Variable files containing color management information (optional).

/var/lib/hwclock
State directory for hwclock (optional).

/var/lib/misc
Miscellaneous state data.

/var/lib/xdm
X display manager variable data (optional).

/var/lib/<editor>
Editor backup files and state (optional).

/var/lib/<name>
These directories must be used for all distribution packaging support.

/var/lib/<package>
State data for packages and subsystems (optional).

/var/lib/<pkgtool>
Packaging support files (optional).

/var/local
Variable data for */usr/local*.

/var/lock
Lock files are placed in this directory. The naming convention for device lock files is *LCK..<device>* where *<device>* is the device's name in the filesystem. The format used is that of HDU UUCP lock files, that is, lock files contain a PID as a 10-byte ASCII decimal number, followed by a newline character.

/var/log
Miscellaneous log files.

/var/opt
: Variable data for */opt*.

/var/mail
: Users' mailboxes. Replaces */var/spool/mail*.

/var/msgs
: Reserved for historical reasons.

/var/preserve
: Reserved for historical reasons.

/var/run
: Run-time variable files, like files holding process identifiers (PIDs) and logged user information *(utmp)*. Files in this directory are usually cleared when the system boots.

/var/spool
: Spooled (or queued) files for various programs.

/var/spool/at
: Spooled jobs for *at*(1)

/var/spool/cron
: Spooled jobs for *cron*(8)

/var/spool/lpd
: Spooled files for printing (optional).

/var/spool/lpd/printer
: Spools for a specific printer (optional).

/var/spool/mail
: Replaced by */var/mail*.

/var/spool/mqueue
: Queued outgoing mail (optional).

/var/spool/news
: Spool directory for news (optional).

/var/spool/rwho
: Spooled files for *rwhod*(8) (optional).

/var/spool/smail
: Spooled files for the *smail*(1) mail delivery program.

/var/spool/uucp
: Spooled files for *uucp*(1) (optional).

/var/tmp
: Like */tmp*, this directory holds temporary files stored for an unspecified duration.

/var/yp Database files for NIS, formerly known as the Sun Yellow Pages (YP).

STANDARDS

The Filesystem Hierarchy Standard (FHS), Version 3.0, published March 19, 2015

BUGS

This list is not exhaustive; different distributions and systems may be configured differently.

SEE ALSO

find(1), *ln*(1), *proc*(5), *file-hierarchy*(7), *mount*(8)

The Filesystem Hierarchy Standard

NAME

hostname − hostname resolution description

DESCRIPTION

Hostnames are domains, where a domain is a hierarchical, dot-separated list of subdomains; for example, the machine "monet", in the "example" subdomain of the "com" domain would be represented as "monet.example.com".

Each element of the hostname must be from 1 to 63 characters long and the entire hostname, including the dots, can be at most 253 characters long. Valid characters for hostnames are *ASCII*(7) letters from *a* to *z*, the digits from *0* to *9*, and the hyphen (−). A hostname may not start with a hyphen.

Hostnames are often used with network client and server programs, which must generally translate the name to an address for use. (This task is generally performed by either *getaddrinfo*(3) or the obsolete *gethostbyname*(3).)

Hostnames are resolved by the NSS framework in glibc according to the **hosts** configuration in *nsswitch.conf*(5). The DNS-based name resolver (in the **dns** NSS service module) resolves them in the following fashion.

If the name consists of a single component, that is, contains no dot, and if the environment variable **HOSTALIASES** is set to the name of a file, that file is searched for any string matching the input hostname. The file should consist of lines made up of two white-space separated strings, the first of which is the hostname alias, and the second of which is the complete hostname to be substituted for that alias. If a case-insensitive match is found between the hostname to be resolved and the first field of a line in the file, the substituted name is looked up with no further processing.

If the input name ends with a trailing dot, the trailing dot is removed, and the remaining name is looked up with no further processing.

If the input name does not end with a trailing dot, it is looked up by searching through a list of domains until a match is found. The default search list includes first the local domain, then its parent domains with at least 2 name components (longest first). For example, in the domain cs.example.com, the name lithium.cchem will be checked first as lithium.cchem.cs.example and then as lithium.cchem.example.com. lithium.cchem.com will not be tried, as there is only one component remaining from the local domain. The search path can be changed from the default by a system-wide configuration file (see *resolver*(5)).

SEE ALSO

getaddrinfo(3), *gethostbyname*(3), *nsswitch.conf*(5), *resolver*(5), *mailaddr*(7), *named*(8)

IETF RFC 1123

IETF RFC 1178

NAME

icmp − Linux IPv4 ICMP kernel module.

DESCRIPTION

This kernel protocol module implements the Internet Control Message Protocol defined in RFC 792. It is used to signal error conditions and for diagnosis. The user doesn't interact directly with this module; instead it communicates with the other protocols in the kernel and these pass the ICMP errors to the application layers. The kernel ICMP module also answers ICMP requests.

A user protocol may receive ICMP packets for all local sockets by opening a raw socket with the protocol **IPPROTO_ICMP**. See *raw*(7) for more information. The types of ICMP packets passed to the socket can be filtered using the **ICMP_FILTER** socket option. ICMP packets are always processed by the kernel too, even when passed to a user socket.

Linux limits the rate of ICMP error packets to each destination. **ICMP_REDIRECT** and **ICMP_DEST_UNREACH** are also limited by the destination route of the incoming packets.

/proc interfaces

ICMP supports a set of */proc* interfaces to configure some global IP parameters. The parameters can be accessed by reading or writing files in the directory */proc/sys/net/ipv4/*. Most of these parameters are rate limitations for specific ICMP types. Linux 2.2 uses a token bucket filter to limit ICMPs. The value is the timeout in jiffies until the token bucket filter is cleared after a burst. A jiffy is a system dependent unit, usually 10ms on i386 and about 1ms on alpha and ia64.

icmp_destunreach_rate (Linux 2.2 to Linux 2.4.9)
: Maximum rate to send ICMP Destination Unreachable packets. This limits the rate at which packets are sent to any individual route or destination. The limit does not affect sending of **ICMP_FRAG_NEEDED** packets needed for path MTU discovery.

icmp_echo_ignore_all (since Linux 2.2)
: If this value is nonzero, Linux will ignore all **ICMP_ECHO** requests.

icmp_echo_ignore_broadcasts (since Linux 2.2)
: If this value is nonzero, Linux will ignore all **ICMP_ECHO** packets sent to broadcast addresses.

icmp_echoreply_rate (Linux 2.2 to Linux 2.4.9)
: Maximum rate for sending **ICMP_ECHOREPLY** packets in response to **ICMP_ECHOREQUEST** packets.

icmp_errors_use_inbound_ifaddr (Boolean; default: disabled; since Linux 2.6.12)
: If disabled, ICMP error messages are sent with the primary address of the exiting interface.

  If enabled, the message will be sent with the primary address of the interface that received the packet that caused the ICMP error. This is the behavior that many network administrators will expect from a router. And it can make debugging complicated network layouts much easier.

  Note that if no primary address exists for the interface selected, then the primary address of the first non-loopback interface that has one will be used regardless of this setting.

icmp_ignore_bogus_error_responses (Boolean; default: disabled; since Linux 2.2)
: Some routers violate RFC1122 by sending bogus responses to broadcast frames. Such violations are normally logged via a kernel warning. If this parameter is enabled, the kernel will not give such warnings, which will avoid log file clutter.

icmp_paramprob_rate (Linux 2.2 to Linux 2.4.9)
: Maximum rate for sending **ICMP_PARAMETERPROB** packets. These packets are sent when a packet arrives with an invalid IP header.

icmp_ratelimit (integer; default: 1000; since Linux 2.4.10)
: Limit the maximum rates for sending ICMP packets whose type matches *icmp_ratemask* (see below) to specific targets. 0 to disable any limiting, otherwise the minimum space between responses in milliseconds.

icmp_ratemask (integer; default: see below; since Linux 2.4.10)
: Mask made of ICMP types for which rates are being limited.

Significant bits: IHGFEDCBA9876543210
Default mask: 0000001100000011000 (0x1818)

Bit definitions (see the Linux kernel source file *include/linux/icmp.h*):

0 Echo Reply
3 Destination Unreachable *
4 Source Quench *
5 Redirect
8 Echo Request
B Time Exceeded *
C Parameter Problem *
D Timestamp Request
E Timestamp Reply
F Info Request
G Info Reply
H Address Mask Request
I Address Mask Reply

The bits marked with an asterisk are rate limited by default (see the default mask above).

icmp_timeexceed_rate (Linux 2.2 to Linux 2.4.9)
Maximum rate for sending **ICMP_TIME_EXCEEDED** packets. These packets are sent to prevent loops when a packet has crossed too many hops.

ping_group_range (two integers; default: see below; since Linux 2.6.39)
Range of the group IDs (minimum and maximum group IDs, inclusive) that are allowed to create ICMP Echo sockets. The default is "1 0", which means no group is allowed to create ICMP Echo sockets.

VERSIONS

Support for the **ICMP_ADDRESS** request was removed in Linux 2.2.

Support for **ICMP_SOURCE_QUENCH** was removed in Linux 2.2.

NOTES

As many other implementations don't support **IPPROTO_ICMP** raw sockets, this feature should not be relied on in portable programs.

ICMP_REDIRECT packets are not sent when Linux is not acting as a router. They are also accepted only from the old gateway defined in the routing table and the redirect routes are expired after some time.

The 64-bit timestamp returned by **ICMP_TIMESTAMP** is in milliseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

Linux ICMP internally uses a raw socket to send ICMPs. This raw socket may appear in *netstat*(8) output with a zero inode.

SEE ALSO

ip(7), *rdisc*(8)

RFC 792 for a description of the ICMP protocol.

NAME

inode − file inode information

DESCRIPTION

Each file has an inode containing metadata about the file. An application can retrieve this metadata using *stat*(2) (or related calls), which returns a *stat* structure, or *statx*(2), which returns a *statx* structure.

The following is a list of the information typically found in, or associated with, the file inode, with the names of the corresponding structure fields returned by *stat*(2) and *statx*(2):

Device where inode resides
: *stat.st_dev*; *statx.stx_dev_minor* and *statx.stx_dev_major*

  Each inode (as well as the associated file) resides in a filesystem that is hosted on a device. That device is identified by the combination of its major ID (which identifies the general class of device) and minor ID (which identifies a specific instance in the general class).

Inode number
: *stat.st_ino*; *statx.stx_ino*

  Each file in a filesystem has a unique inode number. Inode numbers are guaranteed to be unique only within a filesystem (i.e., the same inode numbers may be used by different filesystems, which is the reason that hard links may not cross filesystem boundaries). This field contains the file's inode number.

File type and mode
: *stat.st_mode*; *statx.stx_mode*

  See the discussion of file type and mode, below.

Link count
: *stat.st_nlink*; *statx.stx_nlink*

  This field contains the number of hard links to the file. Additional links to an existing file are created using *link*(2).

User ID
: *stat.st_uid*; *statx.stx_uid*

  This field records the user ID of the owner of the file. For newly created files, the file user ID is the effective user ID of the creating process. The user ID of a file can be changed using *chown*(2).

Group ID
: *stat.st_gid*; *statx.stx_gid*

  The inode records the ID of the group owner of the file. For newly created files, the file group ID is either the group ID of the parent directory or the effective group ID of the creating process, depending on whether or not the set-group-ID bit is set on the parent directory (see below). The group ID of a file can be changed using *chown*(2).

Device represented by this inode
: *stat.st_rdev*; *statx.stx_rdev_minor* and *statx.stx_rdev_major*

  If this file (inode) represents a device, then the inode records the major and minor ID of that device.

File size
: *stat.st_size*; *statx.stx_size*

  This field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Preferred block size for I/O
: *stat.st_blksize*; *statx.stx_blksize*

  This field gives the "preferred" blocksize for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Number of blocks allocated to the file
: *stat.st_blocks*; *statx.stx_blocks*

  This field indicates the number of blocks allocated to the file, 512-byte units, (This may be smaller than *st_size*/512 when the file has holes.)

  The POSIX.1 standard notes that the unit for the *st_blocks* member of the *stat* structure is not defined by the standard. On many implementations it is 512 bytes; on a few systems, a different unit is used, such as 1024. Furthermore, the unit may differ on a per-filesystem basis.

Last access timestamp (atime)
: *stat.st_atime*; *statx.stx_atime*

  This is the file's last access timestamp. It is changed by file accesses, for example, by *execve*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2) (of more than zero bytes). Other interfaces, such as *mmap*(2), may or may not update the atime timestamp

  Some filesystem types allow mounting in such a way that file and/or directory accesses do not cause an update of the atime timestamp. (See *noatime*, *nodiratime*, and *relatime* in *mount*(8), and related information in *mount*(2).) In addition, the atime timestamp is not updated if a file is opened with the **O_NOATIME** flag; see *open*(2).

File creation (birth) timestamp (btime)
: (not returned in the *stat* structure); *statx.stx_btime*

  The file's creation timestamp. This is set on file creation and not changed subsequently.

  The btime timestamp was not historically present on UNIX systems and is not currently supported by most Linux filesystems.

Last modification timestamp (mtime)
: *stat.st_mtime*; *statx.stx_mtime*

  This is the file's last modification timestamp. It is changed by file modifications, for example, by *mknod*(2), *truncate*(2), *utime*(2), and *write*(2) (of more than zero bytes). Moreover, the mtime timestamp of a directory is changed by the creation or deletion of files in that directory. The mtime timestamp is *not* changed for changes in owner, group, hard link count, or mode.

Last status change timestamp (ctime)
: *stat.st_ctime*; *statx.stx_ctime*

  This is the file's last status change timestamp. It is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The timestamp fields report time measured with a zero point at the *Epoch*, 1970-01-01 00:00:00 +0000, UTC (see *time*(7)).

Nanosecond timestamps are supported on XFS, JFS, Btrfs, and ext4 (since Linux 2.6.23). Nanosecond timestamps are not supported in ext2, ext3, and Reiserfs. In order to return timestamps with nanosecond precision, the timestamp fields in the *stat* and *statx* structures are defined as structures that include a nanosecond component. See *stat*(2) and *statx*(2) for details. On filesystems that do not support subsecond timestamps, the nanosecond fields in the *stat* and *statx* structures are returned with the value 0.

The file type and mode

The *stat.st_mode* field (for *statx*(2), the *statx.stx_mode* field) contains the file type and mode.

POSIX refers to the *stat.st_mode* bits corresponding to the mask **S_IFMT** (see below) as the *file type*, the 12 bits corresponding to the mask 07777 as the *file mode bits* and the least significant 9 bits (0777) as the *file permission bits*.

The following mask values are defined for the file type:

S_IFMT	0170000	bit mask for the file type bit field
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device

S_IFIFO 0010000 FIFO

Thus, to test for a regular file (for example), one could write:

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
}
```

Because tests of the above form are common, additional macros are defined by POSIX to allow the test of the file type in *st_mode* to be written more concisely:

S_ISREG(m) is it a regular file?

S_ISDIR(m) directory?

S_ISCHR(m) character device?

S_ISBLK(m) block device?

S_ISFIFO(m) FIFO (named pipe)?

S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

The preceding code snippet could thus be rewritten as:

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
    /* Handle regular file */
}
```

The definitions of most of the above file type test macros are provided if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_SVID_SOURCE** (in glibc 2.19 and earlier), or **_DEFAULT_SOURCE** (in glibc 2.20 and later). In addition, definitions of all of the above macros except **S_IFSOCK** and **S_ISSOCK**() are provided if **_XOPEN_SOURCE** is defined.

The definition of **S_IFSOCK** can also be exposed either by defining **_XOPEN_SOURCE** with a value of 500 or greater or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

The definition of **S_ISSOCK**() is exposed if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_DEFAULT_SOURCE** (in glibc 2.20 and later), **_XOPEN_SOURCE** with a value of 500 or greater, **_POSIX_C_SOURCE** with a value of 200112L or greater, or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

The following mask values are defined for the file mode component of the *st_mode* field:

S_ISUID 04000 set-user-ID bit (see *execve*(2))
S_ISGID 02000 set-group-ID bit (see below)
S_ISVTX 01000 sticky bit (see below)
S_IRWXU 00700 owner has read, write, and execute permission
S_IRUSR 00400 owner has read permission
S_IWUSR 00200 owner has write permission
S_IXUSR 00100 owner has execute permission
S_IRWXG 00070 group has read, write, and execute permission
S_IRGRP 00040 group has read permission
S_IWGRP 00020 group has write permission
S_IXGRP 00010 group has execute permission
S_IRWXO 00007 others (not in group) have read, write, and execute permission
S_IROTH 00004 others have read permission
S_IWOTH 00002 others have write permission
S_IXOTH 00001 others have execute permission

The set-group-ID bit (**S_ISGID**) has several special uses. For a directory, it indicates that BSD semantics are to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the **S_ISGID** bit set. For an executable file, the set-group-ID bit causes the effective group ID of a process that

executes the file to change as described in *execve*(2). For a file that does not have the group execution bit (**S_IXGRP**) set, the set-group-ID bit indicates mandatory file/record locking.

The sticky bit (**S_ISVTX**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

POSIX.1-1990 did not describe the **S_IFMT**, **S_IFSOCK**, **S_IFLNK**, **S_IFREG**, **S_IFBLK**, **S_IFDIR**, **S_IFCHR**, **S_IFIFO**, and **S_ISVTX** constants, but instead specified the use of the macros **S_ISDIR**() and so on.

The **S_ISLNK**() and **S_ISSOCK**() macros were not in POSIX.1-1996; the former is from SVID 4, the latter from SUSv2.

UNIX V7 (and later systems) had **S_IREAD**, **S_IWRITE**, **S_IEXEC**, and where POSIX prescribes the synonyms **S_IRUSR**, **S_IWUSR**, and **S_IXUSR**.

NOTES

For pseudofiles that are autogenerated by the kernel, the file size (*stat.st_size*; *statx.stx_size*) reported by the kernel is not accurate. For example, the value 0 is returned for many files under the */proc* directory, while various files under */sys* report a size of 4096 bytes, even though the file content is smaller. For such files, one should simply try to read as many bytes as possible (and append '\0' to the returned buffer if it is to be interpreted as a string).

SEE ALSO

stat(1), *stat*(2), *statx*(2), *symlink*(7)

NAME

inotify − monitoring filesystem events

DESCRIPTION

The *inotify* API provides a mechanism for monitoring filesystem events. Inotify can be used to monitor individual files, or to monitor directories. When a directory is monitored, inotify will return events for the directory itself, and for files inside the directory.

The following system calls are used with this API:

- *inotify_init*(2) creates an inotify instance and returns a file descriptor referring to the inotify instance. The more recent *inotify_init1*(2) is like *inotify_init*(2), but has a *flags* argument that provides access to some extra functionality.
- *inotify_add_watch*(2) manipulates the "watch list" associated with an inotify instance. Each item ("watch") in the watch list specifies the pathname of a file or directory, along with some set of events that the kernel should monitor for the file referred to by that pathname. *inotify_add_watch*(2) either creates a new watch item, or modifies an existing watch. Each watch has a unique "watch descriptor", an integer returned by *inotify_add_watch*(2) when the watch is created.
- When events occur for monitored files and directories, those events are made available to the application as structured data that can be read from the inotify file descriptor using *read*(2) (see below).
- *inotify_rm_watch*(2) removes an item from an inotify watch list.
- When all file descriptors referring to an inotify instance have been closed (using *close*(2)), the underlying object and its resources are freed for reuse by the kernel; all associated watches are automatically freed.

With careful programming, an application can use inotify to efficiently monitor and cache the state of a set of filesystem objects. However, robust applications should allow for the fact that bugs in the monitoring logic or races of the kind described below may leave the cache inconsistent with the filesystem state. It is probably wise to do some consistency checking, and rebuild the cache when inconsistencies are detected.

Reading events from an inotify file descriptor

To determine what events have occurred, an application *read*(2)s from the inotify file descriptor. If no events have so far occurred, then, assuming a blocking file descriptor, *read*(2) will block until at least one event occurs (unless interrupted by a signal, in which case the call fails with the error **EINTR**; see *signal*(7)).

Each successful *read*(2) returns a buffer containing one or more of the following structures:

```
struct inotify_event {
    int      wd;       /* Watch descriptor */
    uint32_t mask;     /* Mask describing event */
    uint32_t cookie;   /* Unique cookie associating related
                          events (for rename(2)) */
    uint32_t len;      /* Size of name field */
    char     name[];   /* Optional null-terminated name */
};
```

wd identifies the watch for which this event occurs. It is one of the watch descriptors returned by a previous call to *inotify_add_watch*(2).

mask contains bits that describe the event that occurred (see below).

cookie is a unique integer that connects related events. Currently, this is used only for rename events, and allows the resulting pair of **IN_MOVED_FROM** and **IN_MOVED_TO** events to be connected by the application. For all other event types, *cookie* is set to 0.

The *name* field is present only when an event is returned for a file inside a watched directory; it identifies the filename within the watched directory. This filename is null-terminated, and may include further null bytes ('\0') to align subsequent reads to a suitable address boundary.

The *len* field counts all of the bytes in *name*, including the null bytes; the length of each *inotify_event* structure is thus *sizeof(struct inotify_event)+len*.

The behavior when the buffer given to *read*(2) is too small to return information about the next event

depends on the kernel version: before Linux 2.6.21, *read*(2) returns 0; since Linux 2.6.21, *read*(2) fails with the error **EINVAL**. Specifying a buffer of size

```
sizeof(struct inotify_event) + NAME_MAX + 1
```

will be sufficient to read at least one event.

inotify events

The *inotify_add_watch*(2) *mask* argument and the *mask* field of the *inotify_event* structure returned when *read*(2)ing an inotify file descriptor are both bit masks identifying inotify events. The following bits can be specified in *mask* when calling *inotify_add_watch*(2) and may be returned in the *mask* field returned by *read*(2):

IN_ACCESS (+)
File was accessed (e.g., *read*(2), *execve*(2)).

IN_ATTRIB (*)
Metadata changed—for example, permissions (e.g., *chmod*(2)), timestamps (e.g., *utimensat*(2)), extended attributes (**setxattr**(2)), link count (since Linux 2.6.25; e.g., for the target of *link*(2) and for *unlink*(2)), and user/group ID (e.g., *chown*(2)).

IN_CLOSE_WRITE (+)
File opened for writing was closed.

IN_CLOSE_NOWRITE (*)
File or directory not opened for writing was closed.

IN_CREATE (+)
File/directory created in watched directory (e.g., *open*(2) **O_CREAT**, *mkdir*(2), *link*(2), *symlink*(2), *bind*(2) on a UNIX domain socket).

IN_DELETE (+)
File/directory deleted from watched directory.

IN_DELETE_SELF
Watched file/directory was itself deleted. (This event also occurs if an object is moved to another filesystem, since *mv*(1) in effect copies the file to the other filesystem and then deletes it from the original filesystem.) In addition, an **IN_IGNORED** event will subsequently be generated for the watch descriptor.

IN_MODIFY (+)
File was modified (e.g., *write*(2), *truncate*(2)).

IN_MOVE_SELF
Watched file/directory was itself moved.

IN_MOVED_FROM (+)
Generated for the directory containing the old filename when a file is renamed.

IN_MOVED_TO (+)
Generated for the directory containing the new filename when a file is renamed.

IN_OPEN (*)
File or directory was opened.

Inotify monitoring is inode-based: when monitoring a file (but not when monitoring the directory containing a file), an event can be generated for activity on any link to the file (in the same or a different directory).

When monitoring a directory:

- the events marked above with an asterisk (*) can occur both for the directory itself and for objects inside the directory; and
- the events marked with a plus sign (+) occur only for objects inside the directory (not for the directory itself).

Note: when monitoring a directory, events are not generated for the files inside the directory when the events are performed via a pathname (i.e., a link) that lies outside the monitored directory.

When events are generated for objects inside a watched directory, the *name* field in the returned

inotify_event structure identifies the name of the file within the directory.

The **IN_ALL_EVENTS** macro is defined as a bit mask of all of the above events. This macro can be used as the *mask* argument when calling *inotify_add_watch*(2).

Two additional convenience macros are defined:

IN_MOVE
Equates to **IN_MOVED_FROM | IN_MOVED_TO**.

IN_CLOSE
Equates to **IN_CLOSE_WRITE | IN_CLOSE_NOWRITE**.

The following further bits can be specified in *mask* when calling *inotify_add_watch*(2):

IN_DONT_FOLLOW (since Linux 2.6.15)
Don't dereference *pathname* if it is a symbolic link.

IN_EXCL_UNLINK (since Linux 2.6.36)
By default, when watching events on the children of a directory, events are generated for children even after they have been unlinked from the directory. This can result in large numbers of uninteresting events for some applications (e.g., if watching */tmp*, in which many applications create temporary files whose names are immediately unlinked). Specifying **IN_EXCL_UNLINK** changes the default behavior, so that events are not generated for children after they have been unlinked from the watched directory.

IN_MASK_ADD
If a watch instance already exists for the filesystem object corresponding to *pathname*, add (OR) the events in *mask* to the watch mask (instead of replacing the mask); the error **EINVAL** results if **IN_MASK_CREATE** is also specified.

IN_ONESHOT
Monitor the filesystem object corresponding to *pathname* for one event, then remove from watch list.

IN_ONLYDIR (since Linux 2.6.15)
Watch *pathname* only if it is a directory; the error **ENOTDIR** results if *pathname* is not a directory. Using this flag provides an application with a race-free way of ensuring that the monitored object is a directory.

IN_MASK_CREATE (since Linux 4.18)
Watch *pathname* only if it does not already have a watch associated with it; the error **EEXIST** results if *pathname* is already being watched.

Using this flag provides an application with a way of ensuring that new watches do not modify existing ones. This is useful because multiple paths may refer to the same inode, and multiple calls to *inotify_add_watch*(2) without this flag may clobber existing watch masks.

The following bits may be set in the *mask* field returned by *read*(2):

IN_IGNORED
Watch was removed explicitly (**inotify_rm_watch**(2)) or automatically (file was deleted, or filesystem was unmounted). See also BUGS.

IN_ISDIR
Subject of this event is a directory.

IN_Q_OVERFLOW
Event queue overflowed (*wd* is −1 for this event).

IN_UNMOUNT
Filesystem containing watched object was unmounted. In addition, an **IN_IGNORED** event will subsequently be generated for the watch descriptor.

Examples

Suppose an application is watching the directory *dir* and the file *dir/myfile* for all events. The examples below show some events that will be generated for these two objects.

fd = open("dir/myfile", O_RDWR);
: Generates **IN_OPEN** events for both *dir* and *dir/myfile*.

read(fd, buf, count);
: Generates **IN_ACCESS** events for both *dir* and *dir/myfile*.

write(fd, buf, count);
: Generates **IN_MODIFY** events for both *dir* and *dir/myfile*.

fchmod(fd, mode);
: Generates **IN_ATTRIB** events for both *dir* and *dir/myfile*.

close(fd);
: Generates **IN_CLOSE_WRITE** events for both *dir* and *dir/myfile*.

Suppose an application is watching the directories *dir1* and *dir2*, and the file *dir1/myfile*. The following examples show some events that may be generated.

link("dir1/myfile", "dir2/new");
: Generates an **IN_ATTRIB** event for *myfile* and an **IN_CREATE** event for *dir2*.

rename("dir1/myfile", "dir2/myfile");
: Generates an **IN_MOVED_FROM** event for *dir1*, an **IN_MOVED_TO** event for *dir2*, and an **IN_MOVE_SELF** event for *myfile*. The **IN_MOVED_FROM** and **IN_MOVED_TO** events will have the same *cookie* value.

Suppose that *dir1/xx* and *dir2/yy* are (the only) links to the same file, and an application is watching *dir1*, *dir2*, *dir1/xx*, and *dir2/yy*. Executing the following calls in the order given below will generate the following events:

unlink("dir2/yy");
: Generates an **IN_ATTRIB** event for *xx* (because its link count changes) and an **IN_DELETE** event for *dir2*.

unlink("dir1/xx");
: Generates **IN_ATTRIB**, **IN_DELETE_SELF**, and **IN_IGNORED** events for *xx*, and an **IN_DELETE** event for *dir1*.

Suppose an application is watching the directory *dir* and (the empty) directory *dir/subdir*. The following examples show some events that may be generated.

mkdir("dir/new", mode);
: Generates an **IN_CREATE | IN_ISDIR** event for *dir*.

rmdir("dir/subdir");
: Generates **IN_DELETE_SELF** and **IN_IGNORED** events for *subdir*, and an **IN_DELETE | IN_ISDIR** event for *dir*.

/proc interfaces

The following interfaces can be used to limit the amount of kernel memory consumed by inotify:

/proc/sys/fs/inotify/max_queued_events
: The value in this file is used when an application calls *inotify_init*(2) to set an upper limit on the number of events that can be queued to the corresponding inotify instance. Events in excess of this limit are dropped, but an **IN_Q_OVERFLOW** event is always generated.

/proc/sys/fs/inotify/max_user_instances
: This specifies an upper limit on the number of inotify instances that can be created per real user ID.

/proc/sys/fs/inotify/max_user_watches
: This specifies an upper limit on the number of watches that can be created per real user ID.

STANDARDS

Linux.

HISTORY

Inotify was merged into Linux 2.6.13. The required library interfaces were added in glibc 2.4. (**IN_DONT_FOLLOW**, **IN_MASK_ADD**, and **IN_ONLYDIR** were added in glibc 2.5.)

NOTES

Inotify file descriptors can be monitored using *select*(2), *poll*(2), and *epoll*(7). When an event is available, the file descriptor indicates as readable.

Since Linux 2.6.25, signal-driven I/O notification is available for inotify file descriptors; see the discussion of **F_SETFL** (for setting the **O_ASYNC** flag), **F_SETOWN**, and **F_SETSIG** in *fcntl*(2). The *siginfo_t* structure (described in *sigaction*(2)) that is passed to the signal handler has the following fields set: *si_fd* is set to the inotify file descriptor number; *si_signo* is set to the signal number; *si_code* is set to **POLL_IN**; and **POLLIN** is set in *si_band*.

If successive output inotify events produced on the inotify file descriptor are identical (same *wd*, *mask*, *cookie*, and *name*), then they are coalesced into a single event if the older event has not yet been read (but see BUGS). This reduces the amount of kernel memory required for the event queue, but also means that an application can't use inotify to reliably count file events.

The events returned by reading from an inotify file descriptor form an ordered queue. Thus, for example, it is guaranteed that when renaming from one directory to another, events will be produced in the correct order on the inotify file descriptor.

The set of watch descriptors that is being monitored via an inotify file descriptor can be viewed via the entry for the inotify file descriptor in the process's */proc/*pid*/fdinfo* directory. See *proc*(5) for further details. The **FIONREAD** *ioctl*(2) returns the number of bytes available to read from an inotify file descriptor.

Limitations and caveats

The inotify API provides no information about the user or process that triggered the inotify event. In particular, there is no easy way for a process that is monitoring events via inotify to distinguish events that it triggers itself from those that are triggered by other processes.

Inotify reports only events that a user-space program triggers through the filesystem API. As a result, it does not catch remote events that occur on network filesystems. (Applications must fall back to polling the filesystem to catch such events.) Furthermore, various pseudo-filesystems such as */proc*, */sys*, and */dev/pts* are not monitorable with inotify.

The inotify API does not report file accesses and modifications that may occur because of *mmap*(2), *msync*(2), and *munmap*(2).

The inotify API identifies affected files by filename. However, by the time an application processes an inotify event, the filename may already have been deleted or renamed.

The inotify API identifies events via watch descriptors. It is the application's responsibility to cache a mapping (if one is needed) between watch descriptors and pathnames. Be aware that directory renamings may affect multiple cached pathnames.

Inotify monitoring of directories is not recursive: to monitor subdirectories under a directory, additional watches must be created. This can take a significant amount time for large directory trees.

If monitoring an entire directory subtree, and a new subdirectory is created in that tree or an existing directory is renamed into that tree, be aware that by the time you create a watch for the new subdirectory, new files (and subdirectories) may already exist inside the subdirectory. Therefore, you may want to scan the contents of the subdirectory immediately after adding the watch (and, if desired, recursively add watches for any subdirectories that it contains).

Note that the event queue can overflow. In this case, events are lost. Robust applications should handle the possibility of lost events gracefully. For example, it may be necessary to rebuild part or all of the application cache. (One simple, but possibly expensive, approach is to close the inotify file descriptor, empty the cache, create a new inotify file descriptor, and then re-create watches and cache entries for the objects to be monitored.)

If a filesystem is mounted on top of a monitored directory, no event is generated, and no events are generated for objects immediately under the new mount point. If the filesystem is subsequently unmounted, events will subsequently be generated for the directory and the objects it contains.

Dealing with rename() events

As noted above, the **IN_MOVED_FROM** and **IN_MOVED_TO** event pair that is generated by *rename*(2) can be matched up via their shared cookie value. However, the task of matching has some challenges.

These two events are usually consecutive in the event stream available when reading from the inotify file descriptor. However, this is not guaranteed. If multiple processes are triggering events for monitored objects, then (on rare occasions) an arbitrary number of other events may appear between the **IN_MOVED_FROM** and **IN_MOVED_TO** events. Furthermore, it is not guaranteed that the event pair is atomically inserted into the queue: there may be a brief interval where the **IN_MOVED_FROM** has appeared, but the **IN_MOVED_TO** has not.

Matching up the **IN_MOVED_FROM** and **IN_MOVED_TO** event pair generated by *rename*(2) is thus inherently racy. (Don't forget that if an object is renamed outside of a monitored directory, there may not even be an **IN_MOVED_TO** event.) Heuristic approaches (e.g., assume the events are always consecutive) can be used to ensure a match in most cases, but will inevitably miss some cases, causing the application to perceive the **IN_MOVED_FROM** and **IN_MOVED_TO** events as being unrelated. If watch descriptors are destroyed and re-created as a result, then those watch descriptors will be inconsistent with the watch descriptors in any pending events. (Re-creating the inotify file descriptor and rebuilding the cache may be useful to deal with this scenario.)

Applications should also allow for the possibility that the **IN_MOVED_FROM** event was the last event that could fit in the buffer returned by the current call to *read*(2), and the accompanying **IN_MOVED_TO** event might be fetched only on the next *read*(2), which should be done with a (small) timeout to allow for the fact that insertion of the **IN_MOVED_FROM**+**IN_MOVED_TO** event pair is not atomic, and also the possibility that there may not be any **IN_MOVED_TO** event.

BUGS

Before Linux 3.19, *fallocate*(2) did not create any inotify events. Since Linux 3.19, calls to *fallocate*(2) generate **IN_MODIFY** events.

Before Linux 2.6.16, the **IN_ONESHOT** *mask* flag does not work.

As originally designed and implemented, the **IN_ONESHOT** flag did not cause an **IN_IGNORED** event to be generated when the watch was dropped after one event. However, as an unintended effect of other changes, since Linux 2.6.36, an **IN_IGNORED** event is generated in this case.

Before Linux 2.6.25, the kernel code that was intended to coalesce successive identical events (i.e., the two most recent events could potentially be coalesced if the older had not yet been read) instead checked if the most recent event could be coalesced with the *oldest* unread event.

When a watch descriptor is removed by calling *inotify_rm_watch*(2) (or because a watch file is deleted or the filesystem that contains it is unmounted), any pending unread events for that watch descriptor remain available to read. As watch descriptors are subsequently allocated with *inotify_add_watch*(2), the kernel cycles through the range of possible watch descriptors (1 to **INT_MAX**) incrementally. When allocating a free watch descriptor, no check is made to see whether that watch descriptor number has any pending unread events in the inotify queue. Thus, it can happen that a watch descriptor is reallocated even when pending unread events exist for a previous incarnation of that watch descriptor number, with the result that the application might then read those events and interpret them as belonging to the file associated with the newly recycled watch descriptor. In practice, the likelihood of hitting this bug may be extremely low, since it requires that an application cycle through **INT_MAX** watch descriptors, release a watch descriptor while leaving unread events for that watch descriptor in the queue, and then recycle that watch descriptor. For this reason, and because there have been no reports of the bug occurring in real-world applications, as of Linux 3.15, no kernel changes have yet been made to eliminate this possible bug.

EXAMPLES

The following program demonstrates the usage of the inotify API. It marks the directories passed as a command-line arguments and waits for events of type **IN_OPEN**, **IN_CLOSE_NOWRITE**, and **IN_CLOSE_WRITE**.

The following output was recorded while editing the file */home/user/temp/foo* and listing directory */tmp*. Before the file and the directory were opened, **IN_OPEN** events occurred. After the file was closed, an **IN_CLOSE_WRITE** event occurred. After the directory was closed, an **IN_CLOSE_NOWRITE** event occurred. Execution of the program ended when the user pressed the ENTER key.

Example output

```
$ ./a.out /tmp /home/user/temp
Press enter key to terminate.
```

```
    Listening for events.
    IN_OPEN: /home/user/temp/foo [file]
    IN_CLOSE_WRITE: /home/user/temp/foo [file]
    IN_OPEN: /tmp/ [directory]
    IN_CLOSE_NOWRITE: /tmp/ [directory]

    Listening for events stopped.
```

Program source

```
#include <errno.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/inotify.h>
#include <unistd.h>
#include <string.h>

/* Read all available inotify events from the file descriptor 'fd'.
   wd is the table of watch descriptors for the directories in argv.
   argc is the length of wd and argv.
   argv is the list of watched directories.
   Entry 0 of wd and argv is unused. */

static void
handle_events(int fd, int *wd, int argc, char* argv[])
{
    /* Some systems cannot read integer variables if they are not
       properly aligned. On other systems, incorrect alignment may
       decrease performance. Hence, the buffer used for reading from
       the inotify file descriptor should have the same alignment as
       struct inotify_event. */

    char buf[4096]
        __attribute__ ((aligned(__alignof__(struct inotify_event))));
    const struct inotify_event *event;
    ssize_t len;

    /* Loop while events can be read from inotify file descriptor. */

    for (;;) {

        /* Read some events. */

        len = read(fd, buf, sizeof(buf));
        if (len == -1 && errno != EAGAIN) {
            perror("read");
            exit(EXIT_FAILURE);
        }

        /* If the nonblocking read() found no events to read, then
           it returns -1 with errno set to EAGAIN. In that case,
           we exit the loop. */

        if (len <= 0)
            break;

        /* Loop over all events in the buffer. */
```

```
            for (char *ptr = buf; ptr < buf + len;
                    ptr += sizeof(struct inotify_event) + event->len) {

                event = (const struct inotify_event *) ptr;

                /* Print event type. */

                if (event->mask & IN_OPEN)
                    printf("IN_OPEN: ");
                if (event->mask & IN_CLOSE_NOWRITE)
                    printf("IN_CLOSE_NOWRITE: ");
                if (event->mask & IN_CLOSE_WRITE)
                    printf("IN_CLOSE_WRITE: ");

                /* Print the name of the watched directory. */

                for (size_t i = 1; i < argc; ++i) {
                    if (wd[i] == event->wd) {
                        printf("%s/", argv[i]);
                        break;
                    }
                }

                /* Print the name of the file. */

                if (event->len)
                    printf("%s", event->name);

                /* Print type of filesystem object. */

                if (event->mask & IN_ISDIR)
                    printf(" [directory]\n");
                else
                    printf(" [file]\n");
            }
        }
    }

    int
    main(int argc, char* argv[])
    {
        char buf;
        int fd, i, poll_num;
        int *wd;
        nfds_t nfds;
        struct pollfd fds[2];

        if (argc < 2) {
            printf("Usage: %s PATH [PATH ...]\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        printf("Press ENTER key to terminate.\n");

        /* Create the file descriptor for accessing the inotify API. */

        fd = inotify_init1(IN_NONBLOCK);
        if (fd == -1) {
            perror("inotify_init1");
```

```
    exit(EXIT_FAILURE);
}

/* Allocate memory for watch descriptors. */

wd = calloc(argc, sizeof(int));
if (wd == NULL) {
    perror("calloc");
    exit(EXIT_FAILURE);
}

/* Mark directories for events
   - file was opened
   - file was closed */

for (i = 1; i < argc; i++) {
    wd[i] = inotify_add_watch(fd, argv[i],
                              IN_OPEN | IN_CLOSE);
    if (wd[i] == -1) {
        fprintf(stderr, "Cannot watch '%s': %s\n",
                argv[i], strerror(errno));
        exit(EXIT_FAILURE);
    }
}

/* Prepare for polling. */

nfds = 2;

fds[0].fd = STDIN_FILENO;       /* Console input */
fds[0].events = POLLIN;

fds[1].fd = fd;                 /* Inotify input */
fds[1].events = POLLIN;

/* Wait for events and/or terminal input. */

printf("Listening for events.\n");
while (1) {
    poll_num = poll(fds, nfds, -1);
    if (poll_num == -1) {
        if (errno == EINTR)
            continue;
        perror("poll");
        exit(EXIT_FAILURE);
    }

    if (poll_num > 0) {

        if (fds[0].revents & POLLIN) {

            /* Console input is available. Empty stdin and quit. */

            while (read(STDIN_FILENO, &buf, 1) > 0 && buf != '\n')
                continue;
            break;
        }

        if (fds[1].revents & POLLIN) {
```

```
                    /* Inotify events are available. */

                    handle_events(fd, wd, argc, argv);
                }
            }
        }

        printf("Listening for events stopped.\n");

        /* Close inotify file descriptor. */

        close(fd);

        free(wd);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

inotifywait(1), *inotifywatch*(1), *inotify_add_watch*(2), *inotify_init*(2), *inotify_init1*(2), *inotify_rm_watch*(2), *read*(2), *stat*(2), *fanotify*(7)

Documentation/filesystems/inotify.txt in the Linux kernel source tree

NAME

ip − Linux IPv4 protocol implementation

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */

tcp_socket = **socket(AF_INET, SOCK_STREAM, 0);**
udp_socket = **socket(AF_INET, SOCK_DGRAM, 0);**
raw_socket = **socket(AF_INET, SOCK_RAW,** *protocol***);**

DESCRIPTION

Linux implements the Internet Protocol, version 4, described in RFC 791 and RFC 1122. **ip** contains a level 2 multicasting implementation conforming to RFC 1112. It also contains an IP router including a packet filter.

The programming interface is BSD-sockets compatible. For more information on sockets, see *socket*(7).

An IP socket is created using *socket*(2):

```
socket(AF_INET, socket_type, protocol);
```

Valid socket types include **SOCK_STREAM** to open a stream socket, **SOCK_DGRAM** to open a datagram socket, and **SOCK_RAW** to open a *raw*(7) socket to access the IP protocol directly.

protocol is the IP protocol in the IP header to be received or sent. Valid values for *protocol* include:

- 0 and **IPPROTO_TCP** for *tcp*(7) stream sockets;
- 0 and **IPPROTO_UDP** for *udp*(7) datagram sockets;
- **IPPROTO_SCTP** for *sctp*(7) stream sockets; and
- **IPPROTO_UDPLITE** for *udplite*(7) datagram sockets.

For **SOCK_RAW** you may specify a valid IANA IP protocol defined in RFC 1700 assigned numbers.

When a process wants to receive new incoming packets or connections, it should bind a socket to a local interface address using *bind*(2). In this case, only one IP socket may be bound to any given local (address, port) pair. When **INADDR_ANY** is specified in the bind call, the socket will be bound to *all* local interfaces. When *listen*(2) is called on an unbound socket, the socket is automatically bound to a random free port with the local address set to **INADDR_ANY**. When *connect*(2) is called on an unbound socket, the socket is automatically bound to a random free port or to a usable shared port with the local address set to **INADDR_ANY**.

A TCP local socket address that has been bound is unavailable for some time after closing, unless the **SO_REUSEADDR** flag has been set. Care should be taken when using this flag as it makes TCP less reliable.

Address format

An IP socket address is defined as a combination of an IP interface address and a 16-bit port number. The basic IP protocol does not supply port numbers, they are implemented by higher level protocols like *udp*(7) and *tcp*(7). On raw sockets *sin_port* is set to the IP protocol.

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

sin_family is always set to **AF_INET**. This is required; in Linux 2.2 most networking functions return **EINVAL** when this setting is missing. *sin_port* contains the port in network byte order. The port

numbers below 1024 are called *privileged ports* (or sometimes: *reserved ports*). Only a privileged process (on Linux: a process that has the **CAP_NET_BIND_SERVICE** capability in the user namespace governing its network namespace) may *bind*(2) to these sockets. Note that the raw IPv4 protocol as such has no concept of a port, they are implemented only by higher protocols like *tcp*(7) and *udp*(7).

sin_addr is the IP host address. The *s_addr* member of *struct in_addr* contains the host interface address in network byte order. *in_addr* should be assigned one of the **INADDR_*** values (e.g., **INADDR_LOOPBACK**) using *htonl*(3) or set using the *inet_aton*(3), *inet_addr*(3), *inet_makeaddr*(3) library functions or directly with the name resolver (see *gethostbyname*(3)).

IPv4 addresses are divided into unicast, broadcast, and multicast addresses. Unicast addresses specify a single interface of a host, broadcast addresses specify all hosts on a network, and multicast addresses address all hosts in a multicast group. Datagrams to broadcast addresses can be sent or received only when the **SO_BROADCAST** socket flag is set. In the current implementation, connection-oriented sockets are allowed to use only unicast addresses.

Note that the address and the port are always stored in network byte order. In particular, this means that you need to call *htons*(3) on the number that is assigned to a port. All address/port manipulation functions in the standard library work in network byte order.

Special and reserved addresses

There are several special addresses:

INADDR_LOOPBACK (127.0.0.1)
: always refers to the local host via the loopback device;

INADDR_ANY (0.0.0.0)
: means any address for socket binding;

INADDR_BROADCAST (255.255.255.255)
: has the same effect on *bind*(2) as **INADDR_ANY** for historical reasons. A packet addressed to **INADDR_BROADCAST** through a socket which has **SO_BROADCAST** set will be broadcast to all hosts on the local network segment, as long as the link is broadcast-capable.

Highest-numbered address
Lowest-numbered address
: On any locally-attached non-point-to-point IP subnet with a link type that supports broadcasts, the highest-numbered address (e.g., the .255 address on a subnet with netmask 255.255.255.0) is designated as a broadcast address. It cannot usefully be assigned to an individual interface, and can only be addressed with a socket on which the **SO_BROADCAST** option has been set. Internet standards have historically also reserved the lowest-numbered address (e.g., the .0 address on a subnet with netmask 255.255.255.0) for broadcast, though they call it "obsolete" for this purpose. (Some sources also refer to this as the "network address.") Since Linux 5.14, it is treated as an ordinary unicast address and can be assigned to an interface.

Internet standards have traditionally also reserved various addresses for particular uses, though Linux no longer treats some of these specially.

[0.0.0.1, 0.255.255.255]
[240.0.0.0, 255.255.255.254]
: Addresses in these ranges (0/8 and 240/4) are reserved globally. Since Linux 5.3 and Linux 2.6.25, respectively, the 0/8 and 240/4 addresses, other than **INADDR_ANY** and **INADDR_BROADCAST**, are treated as ordinary unicast addresses. Systems that follow the traditional behaviors may not interoperate with these historically reserved addresses.

[127.0.0.1, 127.255.255.254]
: Addresses in this range (127/8) are treated as loopback addresses akin to the standardized local loopback address **INADDR_LOOPBACK** (127.0.0.1);

[224.0.0.0, 239.255.255.255]
: Addresses in this range (224/4) are dedicated to multicast use.

Socket options

IP supports some protocol-specific socket options that can be set with *setsockopt*(2) and read with *getsockopt*(2). The socket option level for IP is **IPPROTO_IP**. A boolean integer flag is zero when it is false, otherwise true.

When an invalid socket option is specified, *getsockopt*(2) and *setsockopt*(2) fail with the error **ENOPROTOOPT**.

IP_ADD_MEMBERSHIP (since Linux 1.2)
Join a multicast group. Argument is an *ip_mreqn* structure.

```
struct ip_mreqn {
    struct in_addr imr_multiaddr; /* IP multicast group
                                     address */
    struct in_addr imr_address;   /* IP address of local
                                     interface */
    int            imr_ifindex;   /* interface index */
};
```

imr_multiaddr contains the address of the multicast group the application wants to join or leave. It must be a valid multicast address (or *setsockopt*(2) fails with the error **EINVAL**). *imr_address* is the address of the local interface with which the system should join the multicast group; if it is equal to **INADDR_ANY**, an appropriate interface is chosen by the system. *imr_ifindex* is the interface index of the interface that should join/leave the *imr_multiaddr* group, or 0 to indicate any interface.

The *ip_mreqn* structure is available only since Linux 2.2. For compatibility, the old *ip_mreq* structure (present since Linux 1.2) is still supported; it differs from *ip_mreqn* only by not including the *imr_ifindex* field. (The kernel determines which structure is being passed based on the size passed in *optlen*.)

IP_ADD_MEMBERSHIP is valid only for *setsockopt*(2).

IP_ADD_SOURCE_MEMBERSHIP (since Linux 2.4.22 / Linux 2.5.68)
Join a multicast group and allow receiving data only from a specified source. Argument is an *ip_mreq_source* structure.

```
struct ip_mreq_source {
    struct in_addr imr_multiaddr;  /* IP multicast group
                                      address */
    struct in_addr imr_interface;  /* IP address of local
                                      interface */
    struct in_addr imr_sourceaddr; /* IP address of
                                      multicast source */
};
```

The *ip_mreq_source* structure is similar to *ip_mreqn* described under **IP_ADD_MEMBERSHIP**. The *imr_multiaddr* field contains the address of the multicast group the application wants to join or leave. The *imr_interface* field is the address of the local interface with which the system should join the multicast group. Finally, the *imr_sourceaddr* field contains the address of the source the application wants to receive data from.

This option can be used multiple times to allow receiving data from more than one source.

IP_BIND_ADDRESS_NO_PORT (since Linux 4.2)
Inform the kernel to not reserve an ephemeral port when using *bind*(2) with a port number of 0. The port will later be automatically chosen at *connect*(2) time, in a way that allows sharing a source port as long as the 4-tuple is unique.

IP_BLOCK_SOURCE (since Linux 2.4.22 / 2.5.68)
Stop receiving multicast data from a specific source in a given group. This is valid only after the application has subscribed to the multicast group using either **IP_ADD_MEMBERSHIP** or **IP_ADD_SOURCE_MEMBERSHIP**.

Argument is an *ip_mreq_source* structure as described under **IP_ADD_SOURCE_MEMBERSHIP**.

IP_DROP_MEMBERSHIP (since Linux 1.2)
Leave a multicast group. Argument is an *ip_mreqn* or *ip_mreq* structure similar to **IP_ADD_MEMBERSHIP**.

IP_DROP_SOURCE_MEMBERSHIP (since Linux 2.4.22 / 2.5.68)
Leave a source-specific group—that is, stop receiving data from a given multicast group that come from a given source. If the application has subscribed to multiple sources within the same group, data from the remaining sources will still be delivered. To stop receiving data from all sources at once, use **IP_DROP_MEMBERSHIP**.

Argument is an *ip_mreq_source* structure as described under **IP_ADD_SOURCE_MEMBERSHIP**.

IP_FREEBIND (since Linux 2.4)
If enabled, this boolean option allows binding to an IP address that is nonlocal or does not (yet) exist. This permits listening on a socket, without requiring the underlying network interface or the specified dynamic IP address to be up at the time that the application is trying to bind to it. This option is the per-socket equivalent of the *ip_nonlocal_bind /proc* interface described below.

IP_HDRINCL (since Linux 2.0)
If enabled, the user supplies an IP header in front of the user data. Valid only for **SOCK_RAW** sockets; see *raw*(7) for more information. When this flag is enabled, the values set by **IP_OPTIONS**, **IP_TTL**, and **IP_TOS** are ignored.

IP_LOCAL_PORT_RANGE (since Linux 6.3)
Set or get the per-socket default local port range. This option can be used to clamp down the global local port range, defined by the *ip_local_port_range /proc* interface described below, for a given socket.

The option takes an *uint32_t* value with the high 16 bits set to the upper range bound, and the low 16 bits set to the lower range bound. Range bounds are inclusive. The 16-bit values should be in host byte order.

The lower bound has to be less than the upper bound when both bounds are not zero. Otherwise, setting the option fails with EINVAL.

If either bound is outside of the global local port range, or is zero, then that bound has no effect.

To reset the setting, pass zero as both the upper and the lower bound.

IP_MSFILTER (since Linux 2.4.22 / 2.5.68)
This option provides access to the advanced full-state filtering API. Argument is an *ip_msfilter* structure.

```
struct ip_msfilter {
    struct in_addr imsf_multiaddr; /* IP multicast group
                                      address */
    struct in_addr imsf_interface; /* IP address of local
                                      interface */
    uint32_t       imsf_fmode;     /* Filter-mode */

    uint32_t       imsf_numsrc;    /* Number of sources in
                                      the following array */
    struct in_addr imsf_slist[1];  /* Array of source
                                      addresses */
};
```

There are two macros, **MCAST_INCLUDE** and **MCAST_EXCLUDE**, which can be used to specify the filtering mode. Additionally, the *IP_MSFILTER_SIZE*(n) macro exists to determine how much memory is needed to store *ip_msfilter* structure with *n* sources in the source list.

For the full description of multicast source filtering refer to RFC 3376.

IP_MTU (since Linux 2.2)
Retrieve the current known path MTU of the current socket. Returns an integer.

IP_MTU is valid only for *getsockopt*(2) and can be employed only when the socket has been connected.

IP_MTU_DISCOVER (since Linux 2.2)
Set or receive the Path MTU Discovery setting for a socket. When enabled, Linux will perform Path MTU Discovery as defined in RFC 1191 on **SOCK_STREAM** sockets. For non-**SOCK_STREAM** sockets, **IP_PMTUDISC_DO** forces the don't-fragment flag to be set on all outgoing packets. It is the user's responsibility to packetize the data in MTU-sized chunks and to do the retransmits if necessary. The kernel will reject (with **EMSGSIZE**) datagrams that are bigger than the known path MTU. **IP_PMTUDISC_WANT** will fragment a datagram if needed according to the path MTU, or will set the don't-fragment flag otherwise.

The system-wide default can be toggled between **IP_PMTUDISC_WANT** and **IP_PMTUDISC_DONT** by writing (respectively, zero and nonzero values) to the */proc/sys/net/ipv4/ip_no_pmtu_disc* file.

Path MTU discovery value	Meaning
IP_PMTUDISC_WANT	Use per-route settings.
IP_PMTUDISC_DONT	Never do Path MTU Discovery.
IP_PMTUDISC_DO	Always do Path MTU Discovery.
IP_PMTUDISC_PROBE	Set DF but ignore Path MTU.

When PMTU discovery is enabled, the kernel automatically keeps track of the path MTU per destination host. When it is connected to a specific peer with *connect*(2), the currently known path MTU can be retrieved conveniently using the **IP_MTU** socket option (e.g., after an **EMSGSIZE** error occurred). The path MTU may change over time. For connectionless sockets with many destinations, the new MTU for a given destination can also be accessed using the error queue (see **IP_RECVERR**). A new error will be queued for every incoming MTU update.

While MTU discovery is in progress, initial packets from datagram sockets may be dropped. Applications using UDP should be aware of this and not take it into account for their packet retransmit strategy.

To bootstrap the path MTU discovery process on unconnected sockets, it is possible to start with a big datagram size (headers up to 64 kilobytes long) and let it shrink by updates of the path MTU.

To get an initial estimate of the path MTU, connect a datagram socket to the destination address using *connect*(2) and retrieve the MTU by calling *getsockopt*(2) with the **IP_MTU** option.

It is possible to implement RFC 4821 MTU probing with **SOCK_DGRAM** or **SOCK_RAW** sockets by setting a value of **IP_PMTUDISC_PROBE** (available since Linux 2.6.22). This is also particularly useful for diagnostic tools such as *tracepath*(8) that wish to deliberately send probe packets larger than the observed Path MTU.

IP_MULTICAST_ALL (since Linux 2.6.31)
This option can be used to modify the delivery policy of multicast messages. The argument is a boolean integer (defaults to 1). If set to 1, the socket will receive messages from all the groups that have been joined globally on the whole system. Otherwise, it will deliver messages only from the groups that have been explicitly joined (for example via the **IP_ADD_MEMBERSHIP** option) on this particular socket.

IP_MULTICAST_IF (since Linux 1.2)
Set the local device for a multicast socket. The argument for *setsockopt*(2) is an *ip_mreqn* or (since Linux 3.5) *ip_mreq* structure similar to **IP_ADD_MEMBERSHIP**, or an *in_addr* structure. (The kernel determines which structure is being passed based on the size passed in *optlen*.) For *getsockopt*(2), the argument is an *in_addr* structure.

IP_MULTICAST_LOOP (since Linux 1.2)
Set or read a boolean integer argument that determines whether sent multicast packets should be looped back to the local sockets.

IP_MULTICAST_TTL (since Linux 1.2)
Set or read the time-to-live value of outgoing multicast packets for this socket. It is very important for multicast packets to set the smallest TTL possible. The default is 1 which means that multicast packets don't leave the local network unless the user program explicitly requests it. Argument is an integer.

IP_NODEFRAG (since Linux 2.6.36)
If enabled (argument is nonzero), the reassembly of outgoing packets is disabled in the netfilter layer. The argument is an integer.

This option is valid only for **SOCK_RAW** sockets.

IP_OPTIONS (since Linux 2.0)
Set or get the IP options to be sent with every packet from this socket. The arguments are a pointer to a memory buffer containing the options and the option length. The *setsockopt*(2) call sets the IP options associated with a socket. The maximum option size for IPv4 is 40 bytes. See RFC 791 for the allowed options. When the initial connection request packet for a **SOCK_STREAM** socket contains IP options, the IP options will be set automatically to the options from the initial packet with routing headers reversed. Incoming packets are not allowed to change options after the connection is established. The processing of all incoming source routing options is disabled by default and can be enabled by using the *accept_source_route /proc* interface. Other options like timestamps are still handled. For datagram sockets, IP options can be set only by the local user. Calling *getsockopt*(2) with **IP_OPTIONS** puts the current IP options used for sending into the supplied buffer.

IP_PASSSEC (since Linux 2.6.17)
If labeled IPSEC or NetLabel is configured on the sending and receiving hosts, this option enables receiving of the security context of the peer socket in an ancillary message of type **SCM_SECURITY** retrieved using *recvmsg*(2). This option is supported only for UDP sockets; for TCP or SCTP sockets, see the description of the **SO_PEERSEC** option below.

The value given as an argument to *setsockopt*(2) and returned as the result of *getsockopt*(2) is an integer boolean flag.

The security context returned in the **SCM_SECURITY** ancillary message is of the same format as the one described under the **SO_PEERSEC** option below.

Note: the reuse of the **SCM_SECURITY** message type for the **IP_PASSSEC** socket option was likely a mistake, since other IP control messages use their own numbering scheme in the IP namespace and often use the socket option value as the message type. There is no conflict currently since the IP option with the same value as **SCM_SECURITY** is **IP_HDRINCL** and this is never used for a control message type.

IP_PKTINFO (since Linux 2.2)
Pass an **IP_PKTINFO** ancillary message that contains a *pktinfo* structure that supplies some information about the incoming packet. This works only for datagram oriented sockets. The argument is a flag that tells the socket whether the **IP_PKTINFO** message should be passed or not. The message itself can be sent/retrieved only as a control message with a packet using *recvmsg*(2) or *sendmsg*(2).

```
struct in_pktinfo {
    unsigned int   ipi_ifindex;  /* Interface index */
    struct in_addr ipi_spec_dst; /* Local address */
    struct in_addr ipi_addr;     /* Header Destination
                                    address */
};
```

ipi_ifindex is the unique index of the interface the packet was received on. *ipi_spec_dst* is the local address of the packet and *ipi_addr* is the destination address in the packet header. If **IP_PKTINFO** is passed to *sendmsg*(2) and *ipi_spec_dst* is not zero, then it is used as the local source address for the routing table lookup and for setting up IP source route options. When *ipi_ifindex* is not zero, the primary local address of the interface specified by the index overwrites *ipi_spec_dst* for the routing table lookup.

Not supported for **SOCK_STREAM** sockets.

IP_RECVERR (since Linux 2.2)
Enable extended reliable error message passing. When enabled on a datagram socket, all generated errors will be queued in a per-socket error queue. When the user receives an error from a socket operation, the errors can be received by calling *recvmsg*(2) with the **MSG_ERRQUEUE** flag set. The *sock_extended_err* structure describing the error will be passed in an

ancillary message with the type **IP_RECVERR** and the level **IPPROTO_IP**. This is useful for reliable error handling on unconnected sockets. The received data portion of the error queue contains the error packet.

The **IP_RECVERR** control message contains a *sock_extended_err* structure:

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err {
    uint32_t ee_errno;   /* error number */
    uint8_t  ee_origin;  /* where the error originated */
    uint8_t  ee_type;    /* type */
    uint8_t  ee_code;    /* code */
    uint8_t  ee_pad;
    uint32_t ee_info;    /* additional information */
    uint32_t ee_data;    /* other data */
    /* More data may follow */
};

struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

ee_errno contains the *errno* number of the queued error. *ee_origin* is the origin code of where the error originated. The other fields are protocol-specific. The macro **SO_EE_OFFENDER** returns a pointer to the address of the network object where the error originated from given a pointer to the ancillary message. If this address is not known, the *sa_family* member of the *sockaddr* contains **AF_UNSPEC** and the other fields of the *sockaddr* are undefined.

IP uses the *sock_extended_err* structure as follows: *ee_origin* is set to **SO_EE_ORIGIN_ICMP** for errors received as an ICMP packet, or **SO_EE_ORIGIN_LOCAL** for locally generated errors. Unknown values should be ignored. *ee_type* and *ee_code* are set from the type and code fields of the ICMP header. *ee_info* contains the discovered MTU for **EMSGSIZE** errors. The message also contains the *sockaddr_in of the node* caused the error, which can be accessed with the **SO_EE_OFFENDER** macro. The *sin_family* field of the **SO_EE_OFFENDER** address is **AF_UNSPEC** when the source was unknown. When the error originated from the network, all IP options (**IP_OPTIONS**, **IP_TTL**, etc.) enabled on the socket and contained in the error packet are passed as control messages. The payload of the packet causing the error is returned as normal payload. Note that TCP has no error queue; **MSG_ERRQUEUE** is not permitted on **SOCK_STREAM** sockets. **IP_RECVERR** is valid for TCP, but all errors are returned by socket function return or **SO_ERROR** only.

For raw sockets, **IP_RECVERR** enables passing of all received ICMP errors to the application, otherwise errors are reported only on connected sockets

It sets or retrieves an integer boolean flag. **IP_RECVERR** defaults to off.

IP_RECVOPTS (since Linux 2.2)
Pass all incoming IP options to the user in a **IP_OPTIONS** control message. The routing header and other options are already filled in for the local host. Not supported for **SOCK_STREAM** sockets.

IP_RECVORIGDSTADDR (since Linux 2.6.29)
This boolean option enables the **IP_ORIGDSTADDR** ancillary message in *recvmsg*(2), in which the kernel returns the original destination address of the datagram being received. The ancillary message contains a *struct sockaddr_in*. Not supported for **SOCK_STREAM** sockets.

IP_RECVTOS (since Linux 2.2)
If enabled, the **IP_TOS** ancillary message is passed with incoming packets. It contains a byte which specifies the Type of Service/Precedence field of the packet header. Expects a boolean integer flag. Not supported for **SOCK_STREAM** sockets.

IP_RECVTTL (since Linux 2.2)
When this flag is set, pass a **IP_TTL** control message with the time-to-live field of the received packet as a 32 bit integer. Not supported for **SOCK_STREAM** sockets.

IP_RETOPTS (since Linux 2.2)
Identical to **IP_RECVOPTS**, but returns raw unprocessed options with timestamp and route record options not filled in for this hop. Not supported for **SOCK_STREAM** sockets.

IP_ROUTER_ALERT (since Linux 2.2)
Pass all to-be forwarded packets with the IP Router Alert option set to this socket. Valid only for raw sockets. This is useful, for instance, for user-space RSVP daemons. The tapped packets are not forwarded by the kernel; it is the user's responsibility to send them out again. Socket binding is ignored, such packets are filtered only by protocol. Expects an integer flag.

IP_TOS (since Linux 1.0)
Set or receive the Type-Of-Service (TOS) field that is sent with every IP packet originating from this socket. It is used to prioritize packets on the network. TOS is a byte. There are some standard TOS flags defined: **IPTOS_LOWDELAY** to minimize delays for interactive traffic, **IPTOS_THROUGHPUT** to optimize throughput, **IPTOS_RELIABILITY** to optimize for reliability, **IPTOS_MINCOST** should be used for "filler data" where slow transmission doesn't matter. At most one of these TOS values can be specified. Other bits are invalid and shall be cleared. Linux sends **IPTOS_LOWDELAY** datagrams first by default, but the exact behavior depends on the configured queueing discipline. Some high-priority levels may require superuser privileges (the **CAP_NET_ADMIN** capability).

IP_TRANSPARENT (since Linux 2.6.24)
Setting this boolean option enables transparent proxying on this socket. This socket option allows the calling application to bind to a nonlocal IP address and operate both as a client and a server with the foreign address as the local endpoint. NOTE: this requires that routing be set up in a way that packets going to the foreign address are routed through the TProxy box (i.e., the system hosting the application that employs the **IP_TRANSPARENT** socket option). Enabling this socket option requires superuser privileges (the **CAP_NET_ADMIN** capability).

TProxy redirection with the iptables TPROXY target also requires that this option be set on the redirected socket.

IP_TTL (since Linux 1.0)
Set or retrieve the current time-to-live field that is used in every packet sent from this socket.

IP_UNBLOCK_SOURCE (since Linux 2.4.22 / 2.5.68)
Unblock previously blocked multicast source. Returns **EADDRNOTAVAIL** when given source is not being blocked.

Argument is an *ip_mreq_source* structure as described under **IP_ADD_SOURCE_MEMBERSHIP**.

SO_PEERSEC (since Linux 2.6.17)
If labeled IPSEC or NetLabel is configured on both the sending and receiving hosts, this read-only socket option returns the security context of the peer socket connected to this socket. By default, this will be the same as the security context of the process that created the peer socket unless overridden by the policy or by a process with the required permissions.

The argument to *getsockopt*(2) is a pointer to a buffer of the specified length in bytes into which the security context string will be copied. If the buffer length is less than the length of the security context string, then *getsockopt*(2) returns −1, sets *errno* to **ERANGE**, and returns the required length via *optlen*. The caller should allocate at least **NAME_MAX** bytes for the buffer initially, although this is not guaranteed to be sufficient. Resizing the buffer to the returned length and retrying may be necessary.

The security context string may include a terminating null character in the returned length, but is not guaranteed to do so: a security context "foo" might be represented as either {'f','o','o'} of length 3 or {'f','o','o','\0'} of length 4, which are considered to be interchangeable. The string is printable, does not contain non-terminating null characters, and is in an unspecified encoding (in particular, it is not guaranteed to be ASCII or UTF-8).

The use of this option for sockets in the **AF_INET** address family is supported since Linux 2.6.17 for TCP sockets, and since Linux 4.17 for SCTP sockets.

For SELinux, NetLabel conveys only the MLS portion of the security context of the peer across the wire, defaulting the rest of the security context to the values defined in the policy for the netmsg initial security identifier (SID). However, NetLabel can be configured to pass full security contexts over loopback. Labeled IPSEC always passes full security contexts as part of establishing the security association (SA) and looks them up based on the association for each packet.

/proc interfaces

The IP protocol supports a set of */proc* interfaces to configure some global parameters. The parameters can be accessed by reading or writing files in the directory */proc/sys/net/ipv4/*. Interfaces described as *Boolean* take an integer value, with a nonzero value ("true") meaning that the corresponding option is enabled, and a zero value ("false") meaning that the option is disabled.

ip_always_defrag (Boolean; since Linux 2.2.13)

[New with Linux 2.2.13; in earlier kernel versions this feature was controlled at compile time by the **CONFIG_IP_ALWAYS_DEFRAG** option; this option is not present in Linux 2.4.x and later]

When this boolean flag is enabled (not equal 0), incoming fragments (parts of IP packets that arose when some host between origin and destination decided that the packets were too large and cut them into pieces) will be reassembled (defragmented) before being processed, even if they are about to be forwarded.

Enable only if running either a firewall that is the sole link to your network or a transparent proxy; never ever use it for a normal router or host. Otherwise, fragmented communication can be disturbed if the fragments travel over different links. Defragmentation also has a large memory and CPU time cost.

This is automagically turned on when masquerading or transparent proxying are configured.

ip_autoconfig (since Linux 2.2 to Linux 2.6.17)

Not documented.

ip_default_ttl (integer; default: 64; since Linux 2.2)

Set the default time-to-live value of outgoing packets. This can be changed per socket with the **IP_TTL** option.

ip_dynaddr (Boolean; default: disabled; since Linux 2.0.31)

Enable dynamic socket address and masquerading entry rewriting on interface address change. This is useful for dialup interface with changing IP addresses. 0 means no rewriting, 1 turns it on and 2 enables verbose mode.

ip_forward (Boolean; default: disabled; since Linux 1.2)

Enable IP forwarding with a boolean flag. IP forwarding can be also set on a per-interface basis.

ip_local_port_range (since Linux 2.2)

This file contains two integers that define the default local port range allocated to sockets that are not explicitly bound to a port number—that is, the range used for *ephemeral ports*. An ephemeral port is allocated to a socket in the following circumstances:

- the port number in a socket address is specified as 0 when calling *bind*(2);
- *listen*(2) is called on a stream socket that was not previously bound;
- *connect*(2) was called on a socket that was not previously bound;
- *sendto*(2) is called on a datagram socket that was not previously bound.

Allocation of ephemeral ports starts with the first number in *ip_local_port_range* and ends with the second number. If the range of ephemeral ports is exhausted, then the relevant system call returns an error (but see BUGS).

Note that the port range in *ip_local_port_range* should not conflict with the ports used by masquerading (although the case is handled). Also, arbitrary choices may cause problems with some firewall packet filters that make assumptions about the local ports in use. The first

number should be at least greater than 1024, or better, greater than 4096, to avoid clashes with well known ports and to minimize firewall problems.

ip_no_pmtu_disc (Boolean; default: disabled; since Linux 2.2)
: If enabled, don't do Path MTU Discovery for TCP sockets by default. Path MTU discovery may fail if misconfigured firewalls (that drop all ICMP packets) or misconfigured interfaces (e.g., a point-to-point link where the both ends don't agree on the MTU) are on the path. It is better to fix the broken routers on the path than to turn off Path MTU Discovery globally, because not doing it incurs a high cost to the network.

ip_nonlocal_bind (Boolean; default: disabled; since Linux 2.4)
: If set, allows processes to *bind*(2) to nonlocal IP addresses, which can be quite useful, but may break some applications.

ip6frag_time (integer; default: 30)
: Time in seconds to keep an IPv6 fragment in memory.

ip6frag_secret_interval (integer; default: 600)
: Regeneration interval (in seconds) of the hash secret (or lifetime for the hash secret) for IPv6 fragments.

ipfrag_high_thresh (integer)
ipfrag_low_thresh (integer)
: If the amount of queued IP fragments reaches *ipfrag_high_thresh*, the queue is pruned down to *ipfrag_low_thresh*. Contains an integer with the number of bytes.

*neigh/** See *arp*(7).

Ioctls

All ioctls described in *socket*(7) apply to **ip**.

Ioctls to configure generic device parameters are described in *netdevice*(7).

ERRORS

EACCES
: The user tried to execute an operation without the necessary permissions. These include: sending a packet to a broadcast address without having the **SO_BROADCAST** flag set; sending a packet via a *prohibit* route; modifying firewall settings without superuser privileges (the **CAP_NET_ADMIN** capability); binding to a privileged port without superuser privileges (the **CAP_NET_BIND_SERVICE** capability).

EADDRINUSE
: Tried to bind to an address already in use.

EADDRNOTAVAIL
: A nonexistent interface was requested or the requested source address was not local.

EAGAIN
: Operation on a nonblocking socket would block.

EALREADY
: A connection operation on a nonblocking socket is already in progress.

ECONNABORTED
: A connection was closed during an *accept*(2).

EHOSTUNREACH
: No valid routing table entry matches the destination address. This error can be caused by an ICMP message from a remote router or for the local routing table.

EINVAL
: Invalid argument passed. For send operations this can be caused by sending to a *blackhole* route.

EISCONN
: *connect*(2) was called on an already connected socket.

EMSGSIZE
Datagram is bigger than an MTU on the path and it cannot be fragmented.

ENOBUFS
ENOMEM
Not enough free memory. This often means that the memory allocation is limited by the socket buffer limits, not by the system memory, but this is not 100% consistent.

ENOENT
SIOCGSTAMP was called on a socket where no packet arrived.

ENOPKG
A kernel subsystem was not configured.

ENOPROTOOPT and **EOPNOTSUPP**
Invalid socket option passed.

ENOTCONN
The operation is defined only on a connected socket, but the socket wasn't connected.

EPERM
User doesn't have permission to set high priority, change configuration, or send signals to the requested process or group.

EPIPE The connection was unexpectedly closed or shut down by the other end.

ESOCKTNOSUPPORT
The socket is not configured or an unknown socket type was requested.

Other errors may be generated by the overlaying protocols; see *tcp*(7), *raw*(7), *udp*(7), and *socket*(7).

NOTES

IP_FREEBIND, **IP_MSFILTER**, **IP_MTU**, **IP_MTU_DISCOVER**, **IP_RECVORIGDSTADDR**, **IP_PASSSEC**, **IP_PKTINFO**, **IP_RECVERR**, **IP_ROUTER_ALERT**, and **IP_TRANSPARENT** are Linux-specific.

Be very careful with the **SO_BROADCAST** option − it is not privileged in Linux. It is easy to overload the network with careless broadcasts. For new application protocols it is better to use a multicast group instead of broadcasting. Broadcasting is discouraged. See RFC 6762 for an example of a protocol (mDNS) using the more modern multicast approach to communicating with an open-ended group of hosts on the local network.

Some other BSD sockets implementations provide **IP_RCVDSTADDR** and **IP_RECVIF** socket options to get the destination address and the interface of received datagrams. Linux has the more general **IP_PKTINFO** for the same task.

Some BSD sockets implementations also provide an **IP_RECVTTL** option, but an ancillary message with type **IP_RECVTTL** is passed with the incoming packet. This is different from the **IP_TTL** option used in Linux.

Using the **SOL_IP** socket options level isn't portable; BSD-based stacks use the **IPPROTO_IP** level.

INADDR_ANY (0.0.0.0) and **INADDR_BROADCAST** (255.255.255.255) are byte-order-neutral. This means *htonl*(3) has no effect on them.

Compatibility

For compatibility with Linux 2.0, the obsolete **socket(AF_INET, SOCK_PACKET,** *protocol***)** syntax is still supported to open a *packet*(7) socket. This is deprecated and should be replaced by **socket(AF_PACKET, SOCK_RAW,** *protocol***)** instead. The main difference is the new *sockaddr_ll* address structure for generic link layer information instead of the old **sockaddr_pkt**.

BUGS

There are too many inconsistent error values.

The error used to diagnose exhaustion of the ephemeral port range differs across the various system calls (**connect**(2), *bind*(2), *listen*(2), *sendto*(2)) that can assign ephemeral ports.

The ioctls to configure IP-specific interface options and ARP tables are not described.

Receiving the original destination address with **MSG_ERRQUEUE** in *msg_name* by *recvmsg*(2) does not work in some Linux 2.2 kernels.

SEE ALSO

recvmsg(2), *sendmsg*(2), *byteorder*(3), *capabilities*(7), *icmp*(7), *ipv6*(7), *netdevice*(7), *netlink*(7), *raw*(7), *socket*(7), *tcp*(7), *udp*(7), *ip*(8)

The kernel source file *Documentation/networking/ip−sysctl.txt*.

RFC 791 for the original IP specification. RFC 1122 for the IPv4 host requirements. RFC 1812 for the IPv4 router requirements.

NAME

ipc_namespaces − overview of Linux IPC namespaces

DESCRIPTION

IPC namespaces isolate certain IPC resources, namely, System V IPC objects (see *sysvipc*(7)) and (since Linux 2.6.30) POSIX message queues (see *mq_overview*(7)). The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.

Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem. Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces.

The following */proc* interfaces are distinct in each IPC namespace:

- The POSIX message queue interfaces in */proc/sys/fs/mqueue*.
- The System V IPC interfaces in */proc/sys/kernel*, namely: *msgmax*, *msgmnb*, *msgmni*, *sem*, *shmall*, *shmmax*, *shmmni*, and *shm_rmid_forced*.
- The System V IPC interfaces in */proc/sysvipc*.

When an IPC namespace is destroyed (i.e., when the last process that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed.

Use of IPC namespaces requires a kernel that is configured with the **CONFIG_IPC_NS** option.

SEE ALSO

nsenter(1), *unshare*(1), *clone*(2), *setns*(2), *unshare*(2), *mq_overview*(7), *namespaces*(7), *sysvipc*(7)

NAME

ipv6 − Linux IPv6 protocol implementation

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

tcp6_socket = **socket(AF_INET6, SOCK_STREAM, 0);**
raw6_socket = **socket(AF_INET6, SOCK_RAW,** *protocol***);**
udp6_socket = **socket(AF_INET6, SOCK_DGRAM,** *protocol***);**

DESCRIPTION

Linux 2.2 optionally implements the Internet Protocol, version 6. This man page contains a description of the IPv6 basic API as implemented by the Linux kernel and glibc 2.1. The interface is based on the BSD sockets interface; see *socket*(7).

The IPv6 API aims to be mostly compatible with the IPv4 API (see *ip*(7)). Only differences are described in this man page.

To bind an **AF_INET6** socket to any process, the local address should be copied from the *in6addr_any* variable which has *in6_addr* type. In static initializations, **IN6ADDR_ANY_INIT** may also be used, which expands to a constant expression. Both of them are in network byte order.

The IPv6 loopback address (::1) is available in the global *in6addr_loopback* variable. For initializations, **IN6ADDR_LOOPBACK_INIT** should be used.

IPv4 connections can be handled with the v6 API by using the v4-mapped-on-v6 address type; thus a program needs to support only this API type to support both protocols. This is handled transparently by the address handling functions in the C library.

IPv4 and IPv6 share the local port space. When you get an IPv4 connection or packet to an IPv6 socket, its source address will be mapped to v6.

Address format

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* port number */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* Scope ID (new in Linux 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];   /* IPv6 address */
};
```

sin6_family is always set to **AF_INET6**; *sin6_port* is the protocol port (see *sin_port* in *ip*(7)); *sin6_flowinfo* is the IPv6 flow identifier; *sin6_addr* is the 128-bit IPv6 address. *sin6_scope_id* is an ID depending on the scope of the address. It is new in Linux 2.4. Linux supports it only for link-local addresses, in that case *sin6_scope_id* contains the interface index (see *netdevice*(7))

IPv6 supports several address types: unicast to address a single host, multicast to address a group of hosts, anycast to address the nearest member of a group of hosts (not implemented in Linux), IPv4-on-IPv6 to address an IPv4 host, and other reserved address types.

The address notation for IPv6 is a group of 8 4-digit hexadecimal numbers, separated with a ':'. "::" stands for a string of 0 bits. Special addresses are ::1 for loopback and ::FFFF:<IPv4 address> for IPv4-mapped-on-IPv6.

The port space of IPv6 is shared with IPv4.

Socket options

IPv6 supports some protocol-specific socket options that can be set with *setsockopt*(2) and read with *getsockopt*(2). The socket option level for IPv6 is **IPPROTO_IPV6**. A boolean integer flag is zero when it is false, otherwise true.

IPV6_ADDRFORM
Turn an **AF_INET6** socket into a socket of a different address family. Only **AF_INET** is currently supported for that. It is allowed only for IPv6 sockets that are connected and bound to a v4-mapped-on-v6 address. The argument is a pointer to an integer containing **AF_INET**. This is useful to pass v4-mapped sockets as file descriptors to programs that don't know how to deal with the IPv6 API.

IPV6_ADD_MEMBERSHIP, IPV6_DROP_MEMBERSHIP
Control membership in multicast groups. Argument is a pointer to a *struct ipv6_mreq*.

IPV6_MTU
getsockopt(): Retrieve the current known path MTU of the current socket. Valid only when the socket has been connected. Returns an integer.

setsockopt(): Set the MTU to be used for the socket. The MTU is limited by the device MTU or the path MTU when path MTU discovery is enabled. Argument is a pointer to integer.

IPV6_MTU_DISCOVER
Control path-MTU discovery on the socket. See **IP_MTU_DISCOVER** in *ip*(7) for details.

IPV6_MULTICAST_HOPS
Set the multicast hop limit for the socket. Argument is a pointer to an integer. −1 in the value means use the route default, otherwise it should be between 0 and 255.

IPV6_MULTICAST_IF
Set the device for outgoing multicast packets on the socket. This is allowed only for **SOCK_DGRAM** and **SOCK_RAW** socket. The argument is a pointer to an interface index (see *netdevice*(7)) in an integer.

IPV6_MULTICAST_LOOP
Control whether the socket sees multicast packets that it has send itself. Argument is a pointer to boolean.

IPV6_RECVPKTINFO (since Linux 2.6.14)
Set delivery of the **IPV6_PKTINFO** control message on incoming datagrams. Such control messages contain a *struct in6_pktinfo*, as per RFC 3542. Allowed only for **SOCK_DGRAM** or **SOCK_RAW** sockets. Argument is a pointer to a boolean value in an integer.

IPV6_RTHDR, IPV6_AUTHHDR, IPV6_DSTOPTS, IPV6_HOPOPTS, IPV6_FLOWINFO, IPV6_HOPLIMIT
Set delivery of control messages for incoming datagrams containing extension headers from the received packet. **IPV6_RTHDR** delivers the routing header, **IPV6_AUTHHDR** delivers the authentication header, **IPV6_DSTOPTS** delivers the destination options, **IPV6_HOPOPTS** delivers the hop options, **IPV6_FLOWINFO** delivers an integer containing the flow ID, **IPV6_HOPLIMIT** delivers an integer containing the hop count of the packet. The control messages have the same type as the socket option. All these header options can also be set for outgoing packets by putting the appropriate control message into the control buffer of *sendmsg*(2). Allowed only for **SOCK_DGRAM** or **SOCK_RAW** sockets. Argument is a pointer to a boolean value.

IPV6_RECVERR
Control receiving of asynchronous error options. See **IP_RECVERR** in *ip*(7) for details. Argument is a pointer to boolean.

IPV6_ROUTER_ALERT
Pass forwarded packets containing a router alert hop-by-hop option to this socket. Allowed only for **SOCK_RAW** sockets. The tapped packets are not forwarded by the kernel, it is the user's responsibility to send them out again. Argument is a pointer to an integer. A positive integer indicates a router alert option value to intercept. Packets carrying a router alert option with a value field containing this integer will be delivered to the socket. A negative integer disables delivery of packets with router alert options to this socket.

IPV6_UNICAST_HOPS
Set the unicast hop limit for the socket. Argument is a pointer to an integer. −1 in the value means use the route default, otherwise it should be between 0 and 255.

IPV6_V6ONLY (since Linux 2.4.21 and 2.6)
If this flag is set to true (nonzero), then the socket is restricted to sending and receiving IPv6 packets only. In this case, an IPv4 and an IPv6 application can bind to a single port at the same time.

If this flag is set to false (zero), then the socket can be used to send and receive packets to and from an IPv6 address or an IPv4-mapped IPv6 address.

The argument is a pointer to a boolean value in an integer.

The default value for this flag is defined by the contents of the file */proc/sys/net/ipv6/bindv6only*. The default value for that file is 0 (false).

ERRORS

ENODEV
The user tried to *bind*(2) to a link-local IPv6 address, but the *sin6_scope_id* in the supplied *sockaddr_in6* structure is not a valid interface index.

VERSIONS

Linux 2.4 will break binary compatibility for the *sockaddr_in6* for 64-bit hosts by changing the alignment of *in6_addr* and adding an additional *sin6_scope_id* field. The kernel interfaces stay compatible, but a program including *sockaddr_in6* or *in6_addr* into other structures may not be. This is not a problem for 32-bit hosts like i386.

The *sin6_flowinfo* field is new in Linux 2.4. It is transparently passed/read by the kernel when the passed address length contains it. Some programs that pass a longer address buffer and then check the outgoing address length may break.

NOTES

The *sockaddr_in6* structure is bigger than the generic *sockaddr*. Programs that assume that all address types can be stored safely in a *struct sockaddr* need to be changed to use *struct sockaddr_storage* for that instead.

SOL_IP, **SOL_IPV6**, **SOL_ICMPV6**, and other **SOL_*** socket options are nonportable variants of **IPPROTO_***. See also *ip*(7).

BUGS

The IPv6 extended API as in RFC 2292 is currently only partly implemented; although the 2.2 kernel has near complete support for receiving options, the macros for generating IPv6 options are missing in glibc 2.1.

IPSec support for EH and AH headers is missing.

Flow label management is not complete and not documented here.

This man page is not complete.

SEE ALSO

cmsg(3), *ip*(7)

RFC 2553: IPv6 BASIC API; Linux tries to be compliant to this. RFC 2460: IPv6 specification.

NAME

iso_8859-1 − ISO/IEC 8859-1 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-1 encodes the characters used in many West European languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-1 characters

The following table displays the characters in ISO/IEC 8859-1 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF

276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ð	LATIN CAPITAL LETTER ETH
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN

370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

ISO/IEC 8859-1 is also known as Latin-1.

SEE ALSO

ascii(7), *charsets*(7), *cp1252*(7), *iso_8859-15*(7), *utf-8*(7)

NAME

iso_8859-2 − ISO/IEC 8859-2 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-2 encodes the Latin characters used in many Central and East European languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-2 characters

The following table displays the characters in ISO/IEC 8859-2 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ą	LATIN CAPITAL LETTER A WITH OGONEK
242	162	A2	˘	BREVE
243	163	A3	Ł	LATIN CAPITAL LETTER L WITH STROKE
244	164	A4	¤	CURRENCY SIGN
245	165	A5	Ľ	LATIN CAPITAL LETTER L WITH CARON
246	166	A6	Ś	LATIN CAPITAL LETTER S WITH ACUTE
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	Š	LATIN CAPITAL LETTER S WITH CARON
252	170	AA	Ş	LATIN CAPITAL LETTER S WITH CEDILLA
253	171	AB	Ť	LATIN CAPITAL LETTER T WITH CARON
254	172	AC	Ź	LATIN CAPITAL LETTER Z WITH ACUTE
255	173	AD		SOFT HYPHEN
256	174	AE	Ž	LATIN CAPITAL LETTER Z WITH CARON
257	175	AF	Ż	LATIN CAPITAL LETTER Z WITH DOT ABOVE
260	176	B0	°	DEGREE SIGN
261	177	B1	ą	LATIN SMALL LETTER A WITH OGONEK
262	178	B2	˛	OGONEK
263	179	B3	ł	LATIN SMALL LETTER L WITH STROKE
264	180	B4	´	ACUTE ACCENT
265	181	B5	ľ	LATIN SMALL LETTER L WITH CARON
266	182	B6	ś	LATIN SMALL LETTER S WITH ACUTE
267	183	B7	ˇ	CARON
270	184	B8	¸	CEDILLA
271	185	B9	š	LATIN SMALL LETTER S WITH CARON
272	186	BA	ş	LATIN SMALL LETTER S WITH CEDILLA
273	187	BB	ť	LATIN SMALL LETTER T WITH CARON
274	188	BC	ź	LATIN SMALL LETTER Z WITH ACUTE

275	189	BD	˝	DOUBLE ACUTE ACCENT
276	190	BE	ž	LATIN SMALL LETTER Z WITH CARON
277	191	BF	ż	LATIN SMALL LETTER Z WITH DOT ABOVE
300	192	C0	Ŕ	LATIN CAPITAL LETTER R WITH ACUTE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ă	LATIN CAPITAL LETTER A WITH BREVE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Ĺ	LATIN CAPITAL LETTER L WITH ACUTE
306	198	C6	Ć	LATIN CAPITAL LETTER C WITH ACUTE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	Č	LATIN CAPITAL LETTER C WITH CARON
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ę	LATIN CAPITAL LETTER E WITH OGONEK
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ě	LATIN CAPITAL LETTER E WITH CARON
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ď	LATIN CAPITAL LETTER D WITH CARON
320	208	D0	Đ	LATIN CAPITAL LETTER D WITH STROKE
321	209	D1	Ń	LATIN CAPITAL LETTER N WITH ACUTE
322	210	D2	Ň	LATIN CAPITAL LETTER N WITH CARON
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Ő	LATIN CAPITAL LETTER O WITH DOUBLE ACUTE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ř	LATIN CAPITAL LETTER R WITH CARON
331	217	D9	Ů	LATIN CAPITAL LETTER U WITH RING ABOVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Ű	LATIN CAPITAL LETTER U WITH DOUBLE ACUTE
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Ţ	LATIN CAPITAL LETTER T WITH CEDILLA
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	ŕ	LATIN SMALL LETTER R WITH ACUTE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ă	LATIN SMALL LETTER A WITH BREVE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	ĺ	LATIN SMALL LETTER L WITH ACUTE
346	230	E6	ć	LATIN SMALL LETTER C WITH ACUTE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	č	LATIN SMALL LETTER C WITH CARON
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ę	LATIN SMALL LETTER E WITH OGONEK
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ě	LATIN SMALL LETTER E WITH CARON
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ď	LATIN SMALL LETTER D WITH CARON
360	240	F0	đ	LATIN SMALL LETTER D WITH STROKE
361	241	F1	ń	LATIN SMALL LETTER N WITH ACUTE
362	242	F2	ň	LATIN SMALL LETTER N WITH CARON
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	ő	LATIN SMALL LETTER O WITH DOUBLE ACUTE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS

367	247	F7	÷	DIVISION SIGN
370	248	F8	ř	LATIN SMALL LETTER R WITH CARON
371	249	F9	ů	LATIN SMALL LETTER U WITH RING ABOVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	ű	LATIN SMALL LETTER U WITH DOUBLE ACUTE
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	ţ	LATIN SMALL LETTER T WITH CEDILLA
377	255	FF	˙	DOT ABOVE

NOTES

ISO/IEC 8859-2 is also known as Latin-2.

SEE ALSO

ascii(7), *charsets*(7), *iso_8859-1*(7), *iso_8859-16*(7), *utf-8*(7)

NAME

iso_8859-3 − ISO/IEC 8859-3 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-3 encodes the characters used in certain Southeast European languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-3 characters

The following table displays the characters in ISO/IEC 8859-3 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ħ	LATIN CAPITAL LETTER H WITH STROKE
242	162	A2	˘	BREVE
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
246	166	A6	Ĥ	LATIN CAPITAL LETTER H WITH CIRCUMFLEX
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	İ	LATIN CAPITAL LETTER I WITH DOT ABOVE
252	170	AA	Ş	LATIN CAPITAL LETTER S WITH CEDILLA
253	171	AB	Ğ	LATIN CAPITAL LETTER G WITH BREVE
254	172	AC	Ĵ	LATIN CAPITAL LETTER J WITH CIRCUMFLEX
255	173	AD		SOFT HYPHEN
257	175	AF	Ż	LATIN CAPITAL LETTER Z WITH DOT ABOVE
260	176	B0	°	DEGREE SIGN
261	177	B1	ħ	LATIN SMALL LETTER H WITH STROKE
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	ĥ	LATIN SMALL LETTER H WITH CIRCUMFLEX
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	ı	LATIN SMALL LETTER DOTLESS I
272	186	BA	ş	LATIN SMALL LETTER S WITH CEDILLA
273	187	BB	ğ	LATIN SMALL LETTER G WITH BREVE
274	188	BC	ĵ	LATIN SMALL LETTER J WITH CIRCUMFLEX
275	189	BD	½	VULGAR FRACTION ONE HALF
277	191	BF	ż	LATIN SMALL LETTER Z WITH DOT ABOVE

300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Ċ	LATIN CAPITAL LETTER C WITH DOT ABOVE
306	198	C6	Ĉ	LATIN CAPITAL LETTER C WITH CIRCUMFLEX
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Ġ	LATIN CAPITAL LETTER G WITH DOT ABOVE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ĝ	LATIN CAPITAL LETTER G WITH CIRCUMFLEX
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ŭ	LATIN CAPITAL LETTER U WITH BREVE
336	222	DE	Ŝ	LATIN CAPITAL LETTER S WITH CIRCUMFLEX
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	ċ	LATIN SMALL LETTER C WITH DOT ABOVE
346	230	E6	ĉ	LATIN SMALL LETTER C WITH CIRCUMFLEX
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	ġ	LATIN SMALL LETTER G WITH DOT ABOVE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN
370	248	F8	ĝ	LATIN SMALL LETTER G WITH CIRCUMFLEX
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ŭ	LATIN SMALL LETTER U WITH BREVE

376	254	FE	ŝ	LATIN SMALL LETTER S WITH CIRCUMFLEX
377	255	FF	˙	DOT ABOVE

NOTES

ISO/IEC 8859-3 is also known as Latin-3.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-4 − ISO/IEC 8859-4 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-4 encodes the characters used in Scandinavian and Baltic languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-4 characters

The following table displays the characters in ISO/IEC 8859-4 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ą	LATIN CAPITAL LETTER A WITH OGONEK
242	162	A2	ĸ	LATIN SMALL LETTER KRA (Greenlandic)
243	163	A3	Ŗ	LATIN CAPITAL LETTER R WITH CEDILLA
244	164	A4	¤	CURRENCY SIGN
245	165	A5	Ĩ	LATIN CAPITAL LETTER I WITH TILDE
246	166	A6	Ļ	LATIN CAPITAL LETTER L WITH CEDILLA
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	Š	LATIN CAPITAL LETTER S WITH CARON
252	170	AA	Ē	LATIN CAPITAL LETTER E WITH MACRON
253	171	AB	Ģ	LATIN CAPITAL LETTER G WITH CEDILLA
254	172	AC	Ŧ	LATIN CAPITAL LETTER T WITH STROKE
255	173	AD		SOFT HYPHEN
256	174	AE	Ž	LATIN CAPITAL LETTER Z WITH CARON
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	ą	LATIN SMALL LETTER A WITH OGONEK
262	178	B2	˛	OGONEK
263	179	B3	ŗ	LATIN SMALL LETTER R WITH CEDILLA
264	180	B4	´	ACUTE ACCENT
265	181	B5	ĩ	LATIN SMALL LETTER I WITH TILDE
266	182	B6	ļ	LATIN SMALL LETTER L WITH CEDILLA
267	183	B7	ˇ	CARON
270	184	B8	¸	CEDILLA
271	185	B9	š	LATIN SMALL LETTER S WITH CARON
272	186	BA	ē	LATIN SMALL LETTER E WITH MACRON
273	187	BB	ģ	LATIN SMALL LETTER G WITH CEDILLA
274	188	BC	ŧ	LATIN SMALL LETTER T WITH STROKE

275	189	BD	Ŋ	LATIN CAPITAL LETTER ENG
276	190	BE	ž	LATIN SMALL LETTER Z WITH CARON
277	191	BF	ŋ	LATIN SMALL LETTER ENG
300	192	C0	Ā	LATIN CAPITAL LETTER A WITH MACRON
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Į	LATIN CAPITAL LETTER I WITH OGONEK
310	200	C8	Č	LATIN CAPITAL LETTER C WITH CARON
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ę	LATIN CAPITAL LETTER E WITH OGONEK
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ė	LATIN CAPITAL LETTER E WITH DOT ABOVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ī	LATIN CAPITAL LETTER I WITH MACRON
320	208	D0	Đ	LATIN CAPITAL LETTER D WITH STROKE
321	209	D1	Ņ	LATIN CAPITAL LETTER N WITH CEDILLA
322	210	D2	Ō	LATIN CAPITAL LETTER O WITH MACRON
323	211	D3	Ķ	LATIN CAPITAL LETTER K WITH CEDILLA
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ų	LATIN CAPITAL LETTER U WITH OGONEK
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ũ	LATIN CAPITAL LETTER U WITH TILDE
336	222	DE	Ū	LATIN CAPITAL LETTER U WITH MACRON
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	ā	LATIN SMALL LETTER A WITH MACRON
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	į	LATIN SMALL LETTER I WITH OGONEK
350	232	E8	č	LATIN SMALL LETTER C WITH CARON
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ę	LATIN SMALL LETTER E WITH OGONEK
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ė	LATIN SMALL LETTER E WITH DOT ABOVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ī	LATIN SMALL LETTER I WITH MACRON
360	240	F0	đ	LATIN SMALL LETTER D WITH STROKE
361	241	F1	ņ	LATIN SMALL LETTER N WITH CEDILLA
362	242	F2	ō	LATIN SMALL LETTER O WITH MACRON
363	243	F3	ķ	LATIN SMALL LETTER K WITH CEDILLA
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS

367	247	F7	÷	DIVISION SIGN
370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ų	LATIN SMALL LETTER U WITH OGONEK
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ũ	LATIN SMALL LETTER U WITH TILDE
376	254	FE	ū	LATIN SMALL LETTER U WITH MACRON
377	255	FF	˙	DOT ABOVE

NOTES

ISO/IEC 8859-4 is also known as Latin-4.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-5 − ISO/IEC 8859-5 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-5 encodes the Cyrillic characters used in many East European languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-5 characters

The following table displays the characters in ISO/IEC 8859-5 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ё	CYRILLIC CAPITAL LETTER IO
242	162	A2	Ђ	CYRILLIC CAPITAL LETTER DJE
243	163	A3	Ѓ	CYRILLIC CAPITAL LETTER GJE
244	164	A4	Є	CYRILLIC CAPITAL LETTER UKRAINIAN IE
245	165	A5	S	CYRILLIC CAPITAL LETTER DZE
246	166	A6	I	CYRILLIC CAPITAL LETTER BYELORUSSIAN-UKRAINIAN I
247	167	A7	Ї	CYRILLIC CAPITAL LETTER YI
250	168	A8	J	CYRILLIC CAPITAL LETTER JE
251	169	A9	Љ	CYRILLIC CAPITAL LETTER LJE
252	170	AA	Њ	CYRILLIC CAPITAL LETTER NJE
253	171	AB	Ћ	CYRILLIC CAPITAL LETTER TSHE
254	172	AC	Ќ	CYRILLIC CAPITAL LETTER KJE
255	173	AD		SOFT HYPHEN
256	174	AE	Ў	CYRILLIC CAPITAL LETTER SHORT U
257	175	AF	Џ	CYRILLIC CAPITAL LETTER DZHE
260	176	B0	А	CYRILLIC CAPITAL LETTER A
261	177	B1	Б	CYRILLIC CAPITAL LETTER BE
262	178	B2	В	CYRILLIC CAPITAL LETTER VE
263	179	B3	Г	CYRILLIC CAPITAL LETTER GHE
264	180	B4	Д	CYRILLIC CAPITAL LETTER DE
265	181	B5	Е	CYRILLIC CAPITAL LETTER IE
266	182	B6	Ж	CYRILLIC CAPITAL LETTER ZHE
267	183	B7	З	CYRILLIC CAPITAL LETTER ZE
270	184	B8	И	CYRILLIC CAPITAL LETTER I
271	185	B9	Й	CYRILLIC CAPITAL LETTER SHORT I
272	186	BA	К	CYRILLIC CAPITAL LETTER KA
273	187	BB	Л	CYRILLIC CAPITAL LETTER EL

274	188	BC	М	CYRILLIC CAPITAL LETTER EM
275	189	BD	Н	CYRILLIC CAPITAL LETTER EN
276	190	BE	О	CYRILLIC CAPITAL LETTER O
277	191	BF	П	CYRILLIC CAPITAL LETTER PE
300	192	C0	Р	CYRILLIC CAPITAL LETTER ER
301	193	C1	С	CYRILLIC CAPITAL LETTER ES
302	194	C2	Т	CYRILLIC CAPITAL LETTER TE
303	195	C3	У	CYRILLIC CAPITAL LETTER U
304	196	C4	Ф	CYRILLIC CAPITAL LETTER EF
305	197	C5	Х	CYRILLIC CAPITAL LETTER HA
306	198	C6	Ц	CYRILLIC CAPITAL LETTER TSE
307	199	C7	Ч	CYRILLIC CAPITAL LETTER CHE
310	200	C8	Ш	CYRILLIC CAPITAL LETTER SHA
311	201	C9	Щ	CYRILLIC CAPITAL LETTER SHCHA
312	202	CA	Ъ	CYRILLIC CAPITAL LETTER HARD SIGN
313	203	CB	Ы	CYRILLIC CAPITAL LETTER YERU
314	204	CC	Ь	CYRILLIC CAPITAL LETTER SOFT SIGN
315	205	CD	Э	CYRILLIC CAPITAL LETTER E
316	206	CE	Ю	CYRILLIC CAPITAL LETTER YU
317	207	CF	Я	CYRILLIC CAPITAL LETTER YA
320	208	D0	а	CYRILLIC SMALL LETTER A
321	209	D1	б	CYRILLIC SMALL LETTER BE
322	210	D2	в	CYRILLIC SMALL LETTER VE
323	211	D3	г	CYRILLIC SMALL LETTER GHE
324	212	D4	д	CYRILLIC SMALL LETTER DE
325	213	D5	е	CYRILLIC SMALL LETTER IE
326	214	D6	ж	CYRILLIC SMALL LETTER ZHE
327	215	D7	з	CYRILLIC SMALL LETTER ZE
330	216	D8	и	CYRILLIC SMALL LETTER I
331	217	D9	й	CYRILLIC SMALL LETTER SHORT I
332	218	DA	к	CYRILLIC SMALL LETTER KA
333	219	DB	л	CYRILLIC SMALL LETTER EL
334	220	DC	м	CYRILLIC SMALL LETTER EM
335	221	DD	н	CYRILLIC SMALL LETTER EN
336	222	DE	о	CYRILLIC SMALL LETTER O
337	223	DF	п	CYRILLIC SMALL LETTER PE
340	224	E0	р	CYRILLIC SMALL LETTER ER
341	225	E1	с	CYRILLIC SMALL LETTER ES
342	226	E2	т	CYRILLIC SMALL LETTER TE
343	227	E3	у	CYRILLIC SMALL LETTER U
344	228	E4	ф	CYRILLIC SMALL LETTER EF
345	229	E5	х	CYRILLIC SMALL LETTER HA
346	230	E6	ц	CYRILLIC SMALL LETTER TSE
347	231	E7	ч	CYRILLIC SMALL LETTER CHE
350	232	E8	ш	CYRILLIC SMALL LETTER SHA
351	233	E9	щ	CYRILLIC SMALL LETTER SHCHA
352	234	EA	ъ	CYRILLIC SMALL LETTER HARD SIGN
353	235	EB	ы	CYRILLIC SMALL LETTER YERU
354	236	EC	ь	CYRILLIC SMALL LETTER SOFT SIGN
355	237	ED	э	CYRILLIC SMALL LETTER E
356	238	EE	ю	CYRILLIC SMALL LETTER YU
357	239	EF	я	CYRILLIC SMALL LETTER YA
360	240	F0	№	NUMERO SIGN
361	241	F1	ё	CYRILLIC SMALL LETTER IO
362	242	F2	ђ	CYRILLIC SMALL LETTER DJE
363	243	F3	ѓ	CYRILLIC SMALL LETTER GJE
364	244	F4	є	CYRILLIC SMALL LETTER UKRAINIAN IE
365	245	F5	s	CYRILLIC SMALL LETTER DZE

366	246	F6	і	CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I
367	247	F7	ї	CYRILLIC SMALL LETTER YI
370	248	F8	ј	CYRILLIC SMALL LETTER JE
371	249	F9	љ	CYRILLIC SMALL LETTER LJE
372	250	FA	њ	CYRILLIC SMALL LETTER NJE
373	251	FB	ћ	CYRILLIC SMALL LETTER TSHE
374	252	FC	ќ	CYRILLIC SMALL LETTER KJE
375	253	FD	§	SECTION SIGN
376	254	FE	ў	CYRILLIC SMALL LETTER SHORT U
377	255	FF	џ	CYRILLIC SMALL LETTER DZHE

SEE ALSO

ascii(7), *charsets*(7), *cp1251*(7), *koi8-r*(7), *koi8-u*(7), *utf-8*(7)

NAME

iso_8859-6 − ISO/IEC 8859-6 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-6 encodes the characters used in the Arabic language.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-6 characters

The following table displays the characters in ISO/IEC 8859-6 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
244	164	A4	¤	CURRENCY SIGN
254	172	AC		ARABIC COMMA
255	173	AD		SOFT HYPHEN
273	187	BB		ARABIC SEMICOLON
277	191	BF		ARABIC QUESTION MARK
301	193	C1		ARABIC LETTER HAMZA
302	194	C2		ARABIC LETTER ALEF WITH MADDA ABOVE
303	195	C3		ARABIC LETTER ALEF WITH HAMZA ABOVE
304	196	C4		ARABIC LETTER WAW WITH HAMZA ABOVE
305	197	C5		ARABIC LETTER ALEF WITH HAMZA BELOW
306	198	C6		ARABIC LETTER YEH WITH HAMZA ABOVE
307	199	C7		ARABIC LETTER ALEF
310	200	C8		ARABIC LETTER BEH
311	201	C9		ARABIC LETTER TEH MARBUTA
312	202	CA		ARABIC LETTER TEH
313	203	CB		ARABIC LETTER THEH
314	204	CC		ARABIC LETTER JEEM
315	205	CD		ARABIC LETTER HAH
316	206	CE		ARABIC LETTER KHAH
317	207	CF		ARABIC LETTER DAL
320	208	D0		ARABIC LETTER THAL
321	209	D1		ARABIC LETTER REH
322	210	D2		ARABIC LETTER ZAIN
323	211	D3		ARABIC LETTER SEEN
324	212	D4		ARABIC LETTER SHEEN
325	213	D5		ARABIC LETTER SAD
326	214	D6		ARABIC LETTER DAD
327	215	D7		ARABIC LETTER TAH
330	216	D8		ARABIC LETTER ZAH

331	217	D9	ARABIC LETTER AIN
332	218	DA	ARABIC LETTER GHAIN
340	224	E0	ARABIC TATWEEL
341	225	E1	ARABIC LETTER FEH
342	226	E2	ARABIC LETTER QAF
343	227	E3	ARABIC LETTER KAF
344	228	E4	ARABIC LETTER LAM
345	229	E5	ARABIC LETTER MEEM
346	230	E6	ARABIC LETTER NOON
347	231	E7	ARABIC LETTER HEH
350	232	E8	ARABIC LETTER WAW
351	233	E9	ARABIC LETTER ALEF MAKSURA
352	234	EA	ARABIC LETTER YEH
353	235	EB	ARABIC FATHATAN
354	236	EC	ARABIC DAMMATAN
355	237	ED	ARABIC KASRATAN
356	238	EE	ARABIC FATHA
357	239	EF	ARABIC DAMMA
360	240	F0	ARABIC KASRA
361	241	F1	ARABIC SHADDA
362	242	F2	ARABIC SUKUN

NOTES

ISO/IEC 8859-6 lacks the glyphs required for many related languages, such as Urdu and Persian (Farsi).

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-7 − ISO/IEC 8859-7 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-7 encodes the characters used in modern monotonic Greek.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-7 characters

The following table displays the characters in ISO/IEC 8859-7 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	‘	LEFT SINGLE QUOTATION MARK
242	162	A2	’	RIGHT SINGLE QUOTATION MARK
243	163	A3	£	POUND SIGN
244	164	A4	€	EURO SIGN
245	165	A5	₯	DRACHMA SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ͺ	GREEK YPOGEGRAMMENI
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
257	175	AF	―	HORIZONTAL BAR
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	΄	GREEK TONOS
265	181	B5	΅	GREEK DIALYTIKA TONOS
266	182	B6	Ά	GREEK CAPITAL LETTER ALPHA WITH TONOS
267	183	B7	·	MIDDLE DOT
270	184	B8	Έ	GREEK CAPITAL LETTER EPSILON WITH TONOS
271	185	B9	Ή	GREEK CAPITAL LETTER ETA WITH TONOS
272	186	BA	Ί	GREEK CAPITAL LETTER IOTA WITH TONOS
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	Ό	GREEK CAPITAL LETTER OMICRON WITH TONOS
275	189	BD	½	VULGAR FRACTION ONE HALF
276	190	BE	Ύ	GREEK CAPITAL LETTER UPSILON WITH TONOS

277	191	BF	Ώ	GREEK CAPITAL LETTER OMEGA WITH TONOS
300	192	C0	ΐ	GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS
301	193	C1	Α	GREEK CAPITAL LETTER ALPHA
302	194	C2	Β	GREEK CAPITAL LETTER BETA
303	195	C3	Γ	GREEK CAPITAL LETTER GAMMA
304	196	C4	Δ	GREEK CAPITAL LETTER DELTA
305	197	C5	Ε	GREEK CAPITAL LETTER EPSILON
306	198	C6	Ζ	GREEK CAPITAL LETTER ZETA
307	199	C7	Η	GREEK CAPITAL LETTER ETA
310	200	C8	Θ	GREEK CAPITAL LETTER THETA
311	201	C9	Ι	GREEK CAPITAL LETTER IOTA
312	202	CA	Κ	GREEK CAPITAL LETTER KAPPA
313	203	CB	Λ	GREEK CAPITAL LETTER LAMBDA
314	204	CC	Μ	GREEK CAPITAL LETTER MU
315	205	CD	Ν	GREEK CAPITAL LETTER NU
316	206	CE	Ξ	GREEK CAPITAL LETTER XI
317	207	CF	Ο	GREEK CAPITAL LETTER OMICRON
320	208	D0	Π	GREEK CAPITAL LETTER PI
321	209	D1	Ρ	GREEK CAPITAL LETTER RHO
323	211	D3	Σ	GREEK CAPITAL LETTER SIGMA
324	212	D4	Τ	GREEK CAPITAL LETTER TAU
325	213	D5	Υ	GREEK CAPITAL LETTER UPSILON
326	214	D6	Φ	GREEK CAPITAL LETTER PHI
327	215	D7	Χ	GREEK CAPITAL LETTER CHI
330	216	D8	Ψ	GREEK CAPITAL LETTER PSI
331	217	D9	Ω	GREEK CAPITAL LETTER OMEGA
332	218	DA	Ϊ	GREEK CAPITAL LETTER IOTA WITH DIALYTIKA
333	219	DB	Ϋ	GREEK CAPITAL LETTER UPSILON WITH DIALYTIKA
334	220	DC	ά	GREEK SMALL LETTER ALPHA WITH TONOS
335	221	DD	έ	GREEK SMALL LETTER EPSILON WITH TONOS
336	222	DE	ή	GREEK SMALL LETTER ETA WITH TONOS
337	223	DF	ί	GREEK SMALL LETTER IOTA WITH TONOS
340	224	E0	ΰ	GREEK SMALL LETTER UPSILON WITH DIALYTIKA AND TONOS
341	225	E1	*α*	GREEK SMALL LETTER ALPHA
342	226	E2	*β*	GREEK SMALL LETTER BETA
343	227	E3	*γ*	GREEK SMALL LETTER GAMMA
344	228	E4	*δ*	GREEK SMALL LETTER DELTA
345	229	E5	*ε*	GREEK SMALL LETTER EPSILON
346	230	E6	*ζ*	GREEK SMALL LETTER ZETA
347	231	E7	*η*	GREEK SMALL LETTER ETA
350	232	E8	*θ*	GREEK SMALL LETTER THETA
351	233	E9	*ι*	GREEK SMALL LETTER IOTA
352	234	EA	*κ*	GREEK SMALL LETTER KAPPA
353	235	EB	*λ*	GREEK SMALL LETTER LAMBDA
354	236	EC	μ	GREEK SMALL LETTER MU
355	237	ED	*ν*	GREEK SMALL LETTER NU
356	238	EE	*ξ*	GREEK SMALL LETTER XI
357	239	EF	*ο*	GREEK SMALL LETTER OMICRON
360	240	F0	*π*	GREEK SMALL LETTER PI
361	241	F1	*ρ*	GREEK SMALL LETTER RHO
362	242	F2	*ς*	GREEK SMALL LETTER FINAL SIGMA
363	243	F3	*σ*	GREEK SMALL LETTER SIGMA
364	244	F4	*τ*	GREEK SMALL LETTER TAU
365	245	F5	*υ*	GREEK SMALL LETTER UPSILON
366	246	F6	*ϕ*	GREEK SMALL LETTER PHI
367	247	F7	*χ*	GREEK SMALL LETTER CHI

370	248	F8	ψ	GREEK SMALL LETTER PSI
371	249	F9	ω	GREEK SMALL LETTER OMEGA
372	250	FA	ϊ	GREEK SMALL LETTER IOTA WITH DIALYTIKA
373	251	FB	ϋ	GREEK SMALL LETTER UPSILON WITH DIALYTIKA
374	252	FC	ό	GREEK SMALL LETTER OMICRON WITH TONOS
375	253	FD	ύ	GREEK SMALL LETTER UPSILON WITH TONOS
376	254	FE	ώ	GREEK SMALL LETTER OMEGA WITH TONOS

NOTES

ISO/IEC 8859-7 was formerly known as ELOT-928 or ECMA-118:1986.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-8 − ISO/IEC 8859-8 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-8 encodes the characters used in Modern Hebrew.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-8 characters

The following table displays the characters in ISO/IEC 8859-8 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	×	MULTIPLICATION SIGN
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	÷	DIVISION SIGN
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF
276	190	BE	¾	VULGAR FRACTION THREE QUARTERS

337	223	DF	‗	DOUBLE LOW LINE
340	224	E0	א	HEBREW LETTER ALEF
341	225	E1	ב	HEBREW LETTER BET
342	226	E2	ג	HEBREW LETTER GIMEL
343	227	E3	ד	HEBREW LETTER DALET
344	228	E4	ה	HEBREW LETTER HE
345	229	E5	ו	HEBREW LETTER VAV
346	230	E6	ז	HEBREW LETTER ZAYIN
347	231	E7	ח	HEBREW LETTER HET
350	232	E8	ט	HEBREW LETTER TET
351	233	E9	י	HEBREW LETTER YOD
352	234	EA	ך	HEBREW LETTER FINAL KAF
353	235	EB	כ	HEBREW LETTER KAF
354	236	EC	ל	HEBREW LETTER LAMED
355	237	ED	ם	HEBREW LETTER FINAL MEM
356	238	EE	מ	HEBREW LETTER MEM
357	239	EF	ן	HEBREW LETTER FINAL NUN
360	240	F0	נ	HEBREW LETTER NUN
361	241	F1	ס	HEBREW LETTER SAMEKH
362	242	F2	ע	HEBREW LETTER AYIN
363	243	F3	ף	HEBREW LETTER FINAL PE
364	244	F4	פ	HEBREW LETTER PE
365	245	F5	ץ	HEBREW LETTER FINAL TSADI
366	246	F6	צ	HEBREW LETTER TSADI
367	247	F7	ק	HEBREW LETTER QOF
370	248	F8	ר	HEBREW LETTER RESH
371	249	F9	ש	HEBREW LETTER SHIN
372	250	FA	ת	HEBREW LETTER TAV
375	253	FD	‎	LEFT-TO-RIGHT MARK
376	254	FE	‏	RIGHT-TO-LEFT MARK

NOTES

ISO/IEC 8859-8 was also known as ISO-IR-138. ISO/IEC 8859-8 includes neither short vowels nor diacritical marks, and Yiddish is not provided for.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-9 − ISO/IEC 8859-9 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-9 encodes the characters used in Turkish.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-9 characters

The following table displays the characters in ISO/IEC 8859-9 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF

276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ğ	LATIN CAPITAL LETTER G WITH BREVE
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	İ	LATIN CAPITAL LETTER I WITH DOT ABOVE
336	222	DE	Ş	LATIN CAPITAL LETTER S WITH CEDILLA
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ğ	LATIN SMALL LETTER G WITH BREVE
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN

370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ı	LATIN SMALL LETTER DOTLESS I
376	254	FE	ş	LATIN SMALL LETTER S WITH CEDILLA
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

ISO/IEC 8859-9 is also known as Latin-5.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-10 − ISO/IEC 8859-10 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-10 encodes the characters used in Nordic languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-10 characters

The following table displays the characters in ISO/IEC 8859-10 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ą	LATIN CAPITAL LETTER A WITH OGONEK
242	162	A2	Ē	LATIN CAPITAL LETTER E WITH MACRON
243	163	A3	Ģ	LATIN CAPITAL LETTER G WITH CEDILLA
244	164	A4	Ī	LATIN CAPITAL LETTER I WITH MACRON
245	165	A5	Ĩ	LATIN CAPITAL LETTER I WITH TILDE
246	166	A6	Ķ	LATIN CAPITAL LETTER K WITH CEDILLA
247	167	A7	§	SECTION SIGN
250	168	A8	Ļ	LATIN CAPITAL LETTER L WITH CEDILLA
251	169	A9	Đ	LATIN CAPITAL LETTER D WITH STROKE
252	170	AA	Š	LATIN CAPITAL LETTER S WITH CARON
253	171	AB	Ŧ	LATIN CAPITAL LETTER T WITH STROKE
254	172	AC	Ž	LATIN CAPITAL LETTER Z WITH CARON
255	173	AD		SOFT HYPHEN
256	174	AE	Ū	LATIN CAPITAL LETTER U WITH MACRON
257	175	AF	Ŋ	LATIN CAPITAL LETTER ENG
260	176	B0	°	DEGREE SIGN
261	177	B1	ą	LATIN SMALL LETTER A WITH OGONEK
262	178	B2	ē	LATIN SMALL LETTER E WITH MACRON
263	179	B3	ģ	LATIN SMALL LETTER G WITH CEDILLA
264	180	B4	ī	LATIN SMALL LETTER I WITH MACRON
265	181	B5	ĩ	LATIN SMALL LETTER I WITH TILDE
266	182	B6	ķ	LATIN SMALL LETTER K WITH CEDILLA
267	183	B7	·	MIDDLE DOT
270	184	B8	ļ	LATIN SMALL LETTER L WITH CEDILLA
271	185	B9	đ	LATIN SMALL LETTER D WITH STROKE
272	186	BA	š	LATIN SMALL LETTER S WITH CARON
273	187	BB	ŧ	LATIN SMALL LETTER T WITH STROKE
274	188	BC	ž	LATIN SMALL LETTER Z WITH CARON
275	189	BD	―	HORIZONTAL BAR

276	190	BE	ū	LATIN SMALL LETTER U WITH MACRON
277	191	BF	ŋ	LATIN SMALL LETTER ENG
300	192	C0	Ā	LATIN CAPITAL LETTER A WITH MACRON
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Į	LATIN CAPITAL LETTER I WITH OGONEK
310	200	C8	Č	LATIN CAPITAL LETTER C WITH CARON
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ę	LATIN CAPITAL LETTER E WITH OGONEK
312	202	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ė	LATIN CAPITAL LETTER E WITH DOT ABOVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Đ	LATIN CAPITAL LETTER ETH
321	209	D1	Ņ	LATIN CAPITAL LETTER N WITH CEDILLA
322	210	D2	Ō	LATIN CAPITAL LETTER O WITH MACRON
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	Ũ	LATIN CAPITAL LETTER U WITH TILDE
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ų	LATIN CAPITAL LETTER U WITH OGONEK
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	ā	LATIN SMALL LETTER A WITH MACRON
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	į	LATIN SMALL LETTER I WITH OGONEK
350	232	E8	č	LATIN SMALL LETTER C WITH CARON
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ę	LATIN SMALL LETTER E WITH OGONEK
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ė	LATIN SMALL LETTER E WITH DOT ABOVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ņ	LATIN SMALL LETTER N WITH CEDILLA
362	242	F2	ō	LATIN SMALL LETTER O WITH MACRON
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	ũ	LATIN SMALL LETTER U WITH TILDE

370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ų	LATIN SMALL LETTER U WITH OGONEK
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ĸ	LATIN SMALL LETTER KRA

NOTES

ISO/IEC 8859-10 is also known as Latin-6.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-11 − ISO/IEC 8859-11 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-11 encodes the characters used in the Thai language.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-11 characters

The following table displays the characters in ISO/IEC 8859-11 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1		THAI CHARACTER KO KAI
242	162	A2		THAI CHARACTER KHO KHAI
243	163	A3		THAI CHARACTER KHO KHUAT
244	164	A4		THAI CHARACTER KHO KHWAI
245	165	A5		THAI CHARACTER KHO KHON
246	166	A6		THAI CHARACTER KHO RAKHANG
247	167	A7		THAI CHARACTER NGO NGU
250	168	A8		THAI CHARACTER CHO CHAN
251	169	A9		THAI CHARACTER CHO CHING
252	170	AA		THAI CHARACTER CHO CHANG
253	171	AB		THAI CHARACTER SO SO
254	172	AC		THAI CHARACTER CHO CHOE
255	173	AD		THAI CHARACTER YO YING
256	174	AE		THAI CHARACTER DO CHADA
257	175	AF		THAI CHARACTER TO PATAK
260	176	B0		THAI CHARACTER THO THAN
261	177	B1		THAI CHARACTER THO NANGMONTHO
262	178	B2		THAI CHARACTER THO PHUTHAO
263	179	B3		THAI CHARACTER NO NEN
264	180	B4		THAI CHARACTER DO DEK
265	181	B5		THAI CHARACTER TO TAO
266	182	B6		THAI CHARACTER THO THUNG
267	183	B7		THAI CHARACTER THO THAHAN
270	184	B8		THAI CHARACTER THO THONG
271	185	B9		THAI CHARACTER NO NU
272	186	BA		THAI CHARACTER BO BAIMAI
273	187	BB		THAI CHARACTER PO PLA
274	188	BC		THAI CHARACTER PHO PHUNG
275	189	BD		THAI CHARACTER FO FA

276	190	BE	THAI CHARACTER PHO PHAN
277	191	BF	THAI CHARACTER FO FAN
300	192	C0	THAI CHARACTER PHO SAMPHAO
301	193	C1	THAI CHARACTER MO MA
302	194	C2	THAI CHARACTER YO YAK
303	195	C3	THAI CHARACTER RO RUA
304	196	C4	THAI CHARACTER RU
305	197	C5	THAI CHARACTER LO LING
306	198	C6	THAI CHARACTER LU
307	199	C7	THAI CHARACTER WO WAEN
310	200	C8	THAI CHARACTER SO SALA
311	201	C9	THAI CHARACTER SO RUSI
312	202	CA	THAI CHARACTER SO SUA
313	203	CB	THAI CHARACTER HO HIP
314	204	CC	THAI CHARACTER LO CHULA
315	205	CD	THAI CHARACTER O ANG
316	206	CE	THAI CHARACTER HO NOKHUK
317	207	CF	THAI CHARACTER PAIYANNOI
320	208	D0	THAI CHARACTER SARA A
321	209	D1	THAI CHARACTER MAI HAN-AKAT
322	210	D2	THAI CHARACTER SARA AA
323	211	D3	THAI CHARACTER SARA AM
324	212	D4	THAI CHARACTER SARA I
325	213	D5	THAI CHARACTER SARA II
326	214	D6	THAI CHARACTER SARA UE
327	215	D7	THAI CHARACTER SARA UEE
330	216	D8	THAI CHARACTER SARA U
331	217	D9	THAI CHARACTER SARA UU
332	218	DA	THAI CHARACTER PHINTHU
337	223	DF	THAI CURRENCY SYMBOL BAHT
340	224	E0	THAI CHARACTER SARA E
341	225	E1	THAI CHARACTER SARA AE
342	226	E2	THAI CHARACTER SARA O
343	227	E3	THAI CHARACTER SARA AI MAIMUAN
344	228	E4	THAI CHARACTER SARA AI MAIMALAI
345	229	E5	THAI CHARACTER LAKKHANGYAO
346	230	E6	THAI CHARACTER MAIYAMOK
347	231	E7	THAI CHARACTER MAITAIKHU
350	232	E8	THAI CHARACTER MAI EK
351	233	E9	THAI CHARACTER MAI THO
352	234	EA	THAI CHARACTER MAI TRI
353	235	EB	THAI CHARACTER MAI CHATTAWA
354	236	EC	THAI CHARACTER THANTHAKHAT
355	237	ED	THAI CHARACTER NIKHAHIT
356	238	EE	THAI CHARACTER YAMAKKAN
357	239	EF	THAI CHARACTER FONGMAN
360	240	F0	THAI DIGIT ZERO
361	241	F1	THAI DIGIT ONE
362	242	F2	THAI DIGIT TWO
363	243	F3	THAI DIGIT THREE
364	244	F4	THAI DIGIT FOUR
365	245	F5	THAI DIGIT FIVE
366	246	F6	THAI DIGIT SIX
367	247	F7	THAI DIGIT SEVEN
370	248	F8	THAI DIGIT EIGHT
371	249	F9	THAI DIGIT NINE
372	250	FA	THAI CHARACTER ANGKHANKHU
373	251	FB	THAI CHARACTER KHOMUT

NOTES

ISO/IEC 8859-11 is the same as TIS (Thai Industrial Standard) 620-2253, commonly known as TIS-620, except for the character in position A0: ISO/IEC 8859-11 defines this as NO-BREAK SPACE, while TIS-620 leaves it undefined.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-13 − ISO/IEC 8859-13 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-13 encodes the characters used in Baltic Rim languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-13 characters

The following table displays the characters in ISO/IEC 8859-13 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	”	RIGHT DOUBLE QUOTATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	„	DOUBLE LOW-9 QUOTATION MARK
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	Ø	LATIN CAPITAL LETTER O WITH STROKE
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	Ŗ	LATIN CAPITAL LETTER R WITH CEDILLA
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	Æ	LATIN CAPITAL LETTER AE
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	“	LEFT DOUBLE QUOTATION MARK
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	ø	LATIN SMALL LETTER O WITH STROKE
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	ŗ	LATIN SMALL LETTER R WITH CEDILLA
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF

276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	æ	LATIN SMALL LETTER AE
300	192	C0	Ą	LATIN CAPITAL LETTER A WITH OGONEK
301	193	C1	Į	LATIN CAPITAL LETTER I WITH OGONEK
302	194	C2	Ā	LATIN CAPITAL LETTER A WITH MACRON
303	195	C3	Ć	LATIN CAPITAL LETTER C WITH ACUTE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Ę	LATIN CAPITAL LETTER E WITH OGONEK
307	199	C7	Ē	LATIN CAPITAL LETTER E WITH MACRON
310	200	C8	Č	LATIN CAPITAL LETTER C WITH CARON
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ź	LATIN CAPITAL LETTER Z WITH ACUTE
313	203	CB	Ė	LATIN CAPITAL LETTER E WITH DOT ABOVE
314	204	CC	Ģ	LATIN CAPITAL LETTER G WITH CEDILLA
315	205	CD	Ķ	LATIN CAPITAL LETTER K WITH CEDILLA
316	206	CE	Ī	LATIN CAPITAL LETTER I WITH MACRON
317	207	CF	Ļ	LATIN CAPITAL LETTER L WITH CEDILLA
320	208	D0	Š	LATIN CAPITAL LETTER S WITH CARON
321	209	D1	Ń	LATIN CAPITAL LETTER N WITH ACUTE
322	210	D2	Ņ	LATIN CAPITAL LETTER N WITH CEDILLA
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ō	LATIN CAPITAL LETTER O WITH MACRON
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ų	LATIN CAPITAL LETTER U WITH OGONEK
331	217	D9	Ł	LATIN CAPITAL LETTER L WITH STROKE
332	218	DA	Ś	LATIN CAPITAL LETTER S WITH ACUTE
333	219	DB	Ū	LATIN CAPITAL LETTER U WITH MACRON
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ż	LATIN CAPITAL LETTER Z WITH DOT ABOVE
336	222	DE	Ž	LATIN CAPITAL LETTER Z WITH CARON
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	ą	LATIN SMALL LETTER A WITH OGONEK
341	225	E1	į	LATIN SMALL LETTER I WITH OGONEK
342	226	E2	ā	LATIN SMALL LETTER A WITH MACRON
343	227	E3	ć	LATIN SMALL LETTER C WITH ACUTE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	ę	LATIN SMALL LETTER E WITH OGONEK
347	231	E7	ē	LATIN SMALL LETTER E WITH MACRON
350	232	E8	č	LATIN SMALL LETTER C WITH CARON
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ź	LATIN SMALL LETTER Z WITH ACUTE
353	235	EB	ė	LATIN SMALL LETTER E WITH DOT ABOVE
354	236	EC	ģ	LATIN SMALL LETTER G WITH CEDILLA
355	237	ED	ķ	LATIN SMALL LETTER K WITH CEDILLA
356	238	EE	ī	LATIN SMALL LETTER I WITH MACRON
357	239	EF	ļ	LATIN SMALL LETTER L WITH CEDILLA
360	240	F0	š	LATIN SMALL LETTER S WITH CARON
361	241	F1	ń	LATIN SMALL LETTER N WITH ACUTE
362	242	F2	ņ	LATIN SMALL LETTER N WITH CEDILLA
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ō	LATIN SMALL LETTER O WITH MACRON
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN

370	248	F8	ų	LATIN SMALL LETTER U WITH OGONEK
371	249	F9	ł	LATIN SMALL LETTER L WITH STROKE
372	250	FA	ś	LATIN SMALL LETTER S WITH ACUTE
373	251	FB	ū	LATIN SMALL LETTER U WITH MACRON
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ż	LATIN SMALL LETTER Z WITH DOT ABOVE
376	254	FE	ž	LATIN SMALL LETTER Z WITH CARON
377	255	FF	’	RIGHT SINGLE QUOTATION MARK

NOTES

ISO/IEC 8859-13 is also known as Latin-7.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-14 − ISO/IEC 8859-14 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-14 encodes the characters used in Celtic languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-14 characters

The following table displays the characters in ISO/IEC 8859-14 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ḃ	LATIN CAPITAL LETTER B WITH DOT ABOVE
242	162	A2	ḃ	LATIN SMALL LETTER B WITH DOT ABOVE
243	163	A3	£	POUND SIGN
244	164	A4	Ċ	LATIN CAPITAL LETTER C WITH DOT ABOVE
245	165	A5	ċ	LATIN SMALL LETTER C WITH DOT ABOVE
246	166	A6	Ḋ	LATIN CAPITAL LETTER D WITH DOT ABOVE
247	167	A7	§	SECTION SIGN
250	168	A8	Ẁ	LATIN CAPITAL LETTER W WITH GRAVE
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	Ẃ	LATIN CAPITAL LETTER W WITH ACUTE
253	171	AB	ḋ	LATIN SMALL LETTER D WITH DOT ABOVE
254	172	AC	Ỳ	LATIN CAPITAL LETTER Y WITH GRAVE
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	Ÿ	LATIN CAPITAL LETTER Y WITH DIAERESIS
260	176	B0	Ḟ	LATIN CAPITAL LETTER F WITH DOT ABOVE
261	177	B1	ḟ	LATIN SMALL LETTER F WITH DOT ABOVE
262	178	B2	Ġ	LATIN CAPITAL LETTER G WITH DOT ABOVE
263	179	B3	ġ	LATIN SMALL LETTER G WITH DOT ABOVE
264	180	B4	Ṁ	LATIN CAPITAL LETTER M WITH DOT ABOVE
265	181	B5	ṁ	LATIN SMALL LETTER M WITH DOT ABOVE
266	182	B6	¶	PILCROW SIGN
267	183	B7	Ṗ	LATIN CAPITAL LETTER P WITH DOT ABOVE
270	184	B8	ẁ	LATIN SMALL LETTER W WITH GRAVE
271	185	B9	ṗ	LATIN SMALL LETTER P WITH DOT ABOVE
272	186	BA	ẃ	LATIN SMALL LETTER W WITH ACUTE
273	187	BB	Ṡ	LATIN CAPITAL LETTER S WITH DOT ABOVE
274	188	BC	ỳ	LATIN SMALL LETTER Y WITH GRAVE
275	189	BD	Ẅ	LATIN CAPITAL LETTER W WITH DIAERESIS

276	190	BE	ẅ	LATIN SMALL LETTER W WITH DIAERESIS
277	191	BF	ṡ	LATIN SMALL LETTER S WITH DOT ABOVE
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ŵ	LATIN CAPITAL LETTER W WITH CIRCUMFLEX
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	Ṫ	LATIN CAPITAL LETTER T WITH DOT ABOVE
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Ŷ	LATIN CAPITAL LETTER Y WITH CIRCUMFLEX
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ŵ	LATIN SMALL LETTER W WITH CIRCUMFLEX
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	ṫ	LATIN SMALL LETTER T WITH DOT ABOVE

370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	ŷ	LATIN SMALL LETTER Y WITH CIRCUMFLEX
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

ISO/IEC 8859-14 is also known as Latin-8.

SEE ALSO

ascii(7), *charsets*(7), *utf-8*(7)

NAME

iso_8859-15 − ISO/IEC 8859-15 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-15 encodes the characters used in many West European languages and adds the Euro sign.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-15 characters

The following table displays the characters in ISO/IEC 8859-15 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	€	EURO SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	Š	LATIN CAPITAL LETTER S WITH CARON
247	167	A7	§	SECTION SIGN
250	168	A8	š	LATIN SMALL LETTER S WITH CARON
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD		SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	Ž	LATIN CAPITAL LETTER Z WITH CARON
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	ž	LATIN SMALL LETTER Z WITH CARON
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	Œ	LATIN CAPITAL LIGATURE OE

275	189	BD	œ	LATIN SMALL LIGATURE OE
276	190	BE	Ÿ	LATIN CAPITAL LETTER Y WITH DIAERESIS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Đ	LATIN CAPITAL LETTER ETH
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS

367	247	F7	÷	DIVISION SIGN
370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

ISO/IEC 8859-15 is also known as Latin-9 (or sometimes as Latin-0).

SEE ALSO

ascii(7), *charsets*(7), *cp1252*(7), *iso_8859-1*(7), *utf-8*(7)

NAME

iso_8859-16 − ISO/IEC 8859-16 character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

The ISO/IEC 8859 standard includes several 8-bit extensions to the ASCII character set (also known as ISO/IEC 646-IRV). ISO/IEC 8859-16 encodes the Latin characters used in Southeast European languages.

ISO/IEC 8859 alphabets

The full set of ISO/IEC 8859 alphabets includes:

ISO/IEC 8859-1	West European languages (Latin-1)
ISO/IEC 8859-2	Central and East European languages (Latin-2)
ISO/IEC 8859-3	Southeast European and miscellaneous languages (Latin-3)
ISO/IEC 8859-4	Scandinavian/Baltic languages (Latin-4)
ISO/IEC 8859-5	Latin/Cyrillic
ISO/IEC 8859-6	Latin/Arabic
ISO/IEC 8859-7	Latin/Greek
ISO/IEC 8859-8	Latin/Hebrew
ISO/IEC 8859-9	Latin-1 modification for Turkish (Latin-5)
ISO/IEC 8859-10	Lappish/Nordic/Eskimo languages (Latin-6)
ISO/IEC 8859-11	Latin/Thai
ISO/IEC 8859-13	Baltic Rim languages (Latin-7)
ISO/IEC 8859-14	Celtic (Latin-8)
ISO/IEC 8859-15	West European languages (Latin-9)
ISO/IEC 8859-16	Romanian (Latin-10)

ISO/IEC 8859-16 characters

The following table displays the characters in ISO/IEC 8859-16 that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	Ą	LATIN CAPITAL LETTER A WITH OGONEK
242	162	A2	ą	LATIN SMALL LETTER A WITH OGONEK
243	163	A3	Ł	LATIN CAPITAL LETTER L WITH STROKE
244	164	A4	€	EURO SIGN
245	165	A5	„	DOUBLE LOW-9 QUOTATION MARK
246	166	A6	Š	LATIN CAPITAL LETTER S WITH CARON
247	167	A7	§	SECTION SIGN
250	168	A8	š	LATIN SMALL LETTER S WITH CARON
251	169	A9	©	COPYRIGHT SIGN
252	170	AA		LATIN CAPITAL LETTER S WITH COMMA BELOW
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	Ź	LATIN CAPITAL LETTER Z WITH ACUTE
255	173	AD		SOFT HYPHEN
256	174	AE	ź	LATIN SMALL LETTER Z WITH ACUTE
257	175	AF	Ż	LATIN CAPITAL LETTER Z WITH DOT ABOVE
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	Č	LATIN CAPITAL LETTER C WITH CARON
263	179	B3	ł	LATIN SMALL LETTER L WITH STROKE
264	180	B4	Ž	LATIN CAPITAL LETTER Z WITH CARON
265	181	B5	”	LEFT DOUBLE QUOTATION MARK
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	ž	LATIN SMALL LETTER Z WITH CARON
271	185	B9	č	LATIN SMALL LETTER C WITH CARON
272	186	BA		LATIN SMALL LETTER S WITH COMMA BELOW
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	Œ	LATIN CAPITAL LIGATURE OE

275	189	BD	œ	LATIN SMALL LIGATURE OE
276	190	BE	Ÿ	LATIN CAPITAL LETTER Y WITH DIAERESIS
277	191	BF	ż	LATIN SMALL LETTER Z WITH DOT ABOVE
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ă	LATIN CAPITAL LETTER A WITH BREVE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Ć	LATIN CAPITAL LETTER C WITH ACUTE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Đ	LATIN CAPITAL LETTER D WITH STROKE
321	209	D1	Ń	LATIN CAPITAL LETTER N WITH ACUTE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Ő	LATIN CAPITAL LETTER O WITH DOUBLE ACUTE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	Ś	LATIN CAPITAL LETTER S WITH ACUTE
330	216	D8	Ű	LATIN CAPITAL LETTER U WITH DOUBLE ACUTE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ę	LATIN CAPITAL LETTER E WITH OGONEK
336	222	DE		LATIN CAPITAL LETTER T WITH COMMA BELOW
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ă	LATIN SMALL LETTER A WITH BREVE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	ć	LATIN SMALL LETTER C WITH ACUTE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	đ	LATIN SMALL LETTER D WITH STROKE
361	241	F1	ń	LATIN SMALL LETTER N WITH ACUTE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	ő	LATIN SMALL LETTER O WITH DOUBLE ACUTE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS

367	247	F7	ś	LATIN SMALL LETTER S WITH ACUTE
370	248	F8	ű	LATIN SMALL LETTER U WITH DOUBLE ACUTE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ę	LATIN SMALL LETTER E WITH OGONEK
376	254	FE		LATIN SMALL LETTER T WITH COMMA BELOW
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

NOTES

ISO/IEC 8859-16 is also known as Latin-10.

SEE ALSO

ascii(7), *charsets*(7), *iso_8859-3*(7), *utf-8*(7)

NAME

kernel_lockdown − kernel image access prevention feature

DESCRIPTION

The Kernel Lockdown feature is designed to prevent both direct and indirect access to a running kernel image, attempting to protect against unauthorized modification of the kernel image and to prevent access to security and cryptographic data located in kernel memory, whilst still permitting driver modules to be loaded.

If a prohibited or restricted feature is accessed or used, the kernel will emit a message that looks like:

```
Lockdown: X: Y is restricted, see man kernel_lockdown.7
```

where X indicates the process name and Y indicates what is restricted.

On an EFI-enabled x86 or arm64 machine, lockdown will be automatically enabled if the system boots in EFI Secure Boot mode.

Coverage

When lockdown is in effect, a number of features are disabled or have their use restricted. This includes special device files and kernel services that allow direct access of the kernel image:

/dev/mem
/dev/kmem
/dev/kcore
/dev/ioports
BPF
kprobes

and the ability to directly configure and control devices, so as to prevent the use of a device to access or modify a kernel image:

- The use of module parameters that directly specify hardware parameters to drivers through the kernel command line or when loading a module.
- The use of direct PCI BAR access.
- The use of the ioperm and iopl instructions on x86.
- The use of the KD*IO console ioctls.
- The use of the TIOCSSERIAL serial ioctl.
- The alteration of MSR registers on x86.
- The replacement of the PCMCIA CIS.
- The overriding of ACPI tables.
- The use of ACPI error injection.
- The specification of the ACPI RDSP address.
- The use of ACPI custom methods.

Certain facilities are restricted:

- Only validly signed modules may be loaded (waived if the module file being loaded is vouched for by IMA appraisal).
- Only validly signed binaries may be kexec’d (waived if the binary image file to be executed is vouched for by IMA appraisal).
- Unencrypted hibernation/suspend to swap are disallowed as the kernel image is saved to a medium that can then be accessed.
- Use of debugfs is not permitted as this allows a whole range of actions including direct configuration of, access to and driving of hardware.
- IMA requires the addition of the "secure_boot" rules to the policy, whether or not they are specified on the command line, for both the built-in and custom policies in secure boot lockdown mode.

VERSIONS

The Kernel Lockdown feature was added in Linux 5.4.

NOTES

The Kernel Lockdown feature is enabled by CONFIG_SECURITY_LOCKDOWN_LSM. The *lsm=lsm1,...,lsmN* command line parameter controls the sequence of the initialization of Linux Security Modules. It must contain the string *lockdown* to enable the Kernel Lockdown feature. If the command line parameter is not specified, the initialization falls back to the value of the deprecated *security=* command line parameter and further to the value of CONFIG_LSM.

NAME

keyrings − in-kernel key management and retention facility

DESCRIPTION

The Linux key-management facility is primarily a way for various kernel components to retain or cache security data, authentication keys, encryption keys, and other data in the kernel.

System call interfaces are provided so that user-space programs can manage those objects and also use the facility for their own purposes; see *add_key*(2), *request_key*(2), and *keyctl*(2).

A library and some user-space utilities are provided to allow access to the facility. See *keyctl*(1), *keyctl*(3), and *keyutils*(7) for more information.

Keys

A key has the following attributes:

Serial number (ID)
: This is a unique integer handle by which a key is referred to in system calls. The serial number is sometimes synonymously referred as the key ID. Programmatically, key serial numbers are represented using the type *key_serial_t*.

Type
: A key's type defines what sort of data can be held in the key, how the proposed content of the key will be parsed, and how the payload will be used.

  There are a number of general-purpose types available, plus some specialist types defined by specific kernel components.

Description (name)
: The key description is a printable string that is used as the search term for the key (in conjunction with the key type) as well as a display name. During searches, the description may be partially matched or exactly matched.

Payload (data)
: The payload is the actual content of a key. This is usually set when a key is created, but it is possible for the kernel to upcall to user space to finish the instantiation of a key if that key wasn't already known to the kernel when it was requested. For further details, see *request_key*(2).

  A key's payload can be read and updated if the key type supports it and if suitable permission is granted to the caller.

Access rights
: Much as files do, each key has an owning user ID, an owning group ID, and a security label. Each key also has a set of permissions, though there are more than for a normal UNIX file, and there is an additional category—possessor—beyond the usual user, group, and other (see *Possession*, below).

  Note that keys are quota controlled, since they require unswappable kernel memory. The owning user ID specifies whose quota is to be debited.

Expiration time
: Each key can have an expiration time set. When that time is reached, the key is marked as being expired and accesses to it fail with the error **EKEYEXPIRED**. If not deleted, updated, or replaced, then, after a set amount of time, an expired key is automatically removed (garbage collected) along with all links to it, and attempts to access the key fail with the error **ENOKEY**.

Reference count
: Each key has a reference count. Keys are referenced by keyrings, by currently active users, and by a process's credentials. When the reference count reaches zero, the key is scheduled for garbage collection.

Key types

The kernel provides several basic types of key:

"keyring"
: Keyrings are special keys which store a set of links to other keys (including other keyrings), analogous to a directory holding links to files. The main purpose of a keyring is to prevent

other keys from being garbage collected because nothing refers to them.

Keyrings with descriptions (names) that begin with a period ('.') are reserved to the implementation.

"user" This is a general-purpose key type. The key is kept entirely within kernel memory. The payload may be read and updated by user-space applications.

The payload for keys of this type is a blob of arbitrary data of up to 32,767 bytes.

The description may be any valid string, though it is preferred that it start with a colon-delimited prefix representing the service to which the key is of interest (for instance *"afs:mykey"*).

"logon" (since Linux 3.3)
This key type is essentially the same as *"user"*, but it does not provide reading (i.e., the *keyctl*(2) **KEYCTL_READ** operation), meaning that the key payload is never visible from user space. This is suitable for storing username-password pairs that should not be readable from user space.

The description of a *"logon"* key *must* start with a non-empty colon-delimited prefix whose purpose is to identify the service to which the key belongs. (Note that this differs from keys of the *"user"* type, where the inclusion of a prefix is recommended but is not enforced.)

"big_key" (since Linux 3.13)
This key type is similar to the *"user"* key type, but it may hold a payload of up to 1 MiB in size. This key type is useful for purposes such as holding Kerberos ticket caches.

The payload data may be stored in a tmpfs filesystem, rather than in kernel memory, if the data size exceeds the overhead of storing the data in the filesystem. (Storing the data in a filesystem requires filesystem structures to be allocated in the kernel. The size of these structures determines the size threshold above which the tmpfs storage method is used.) Since Linux 4.8, the payload data is encrypted when stored in tmpfs, thereby preventing it from being written unencrypted into swap space.

There are more specialized key types available also, but they aren't discussed here because they aren't intended for normal user-space use.

Key type names that begin with a period ('.') are reserved to the implementation.

Keyrings

As previously mentioned, keyrings are a special type of key that contain links to other keys (which may include other keyrings). Keys may be linked to by multiple keyrings. Keyrings may be considered as analogous to UNIX directories where each directory contains a set of hard links to files.

Various operations (system calls) may be applied only to keyrings:

Adding A key may be added to a keyring by system calls that create keys. This prevents the new key from being immediately deleted when the system call releases its last reference to the key.

Linking
A link may be added to a keyring pointing to a key that is already known, provided this does not create a self-referential cycle.

Unlinking
A link may be removed from a keyring. When the last link to a key is removed, that key will be scheduled for deletion by the garbage collector.

Clearing
All the links may be removed from a keyring.

Searching
A keyring may be considered the root of a tree or subtree in which keyrings form the branches and non-keyrings the leaves. This tree may be searched for a key matching a particular type and description.

See *keyctl_clear*(3), *keyctl_link*(3), *keyctl_search*(3), and *keyctl_unlink*(3) for more information.

Anchoring keys

To prevent a key from being garbage collected, it must be anchored to keep its reference count elevated when it is not in active use by the kernel.

Keyrings are used to anchor other keys: each link is a reference on a key. Note that keyrings themselves are just keys and are also subject to the same anchoring requirement to prevent them being garbage collected.

The kernel makes available a number of anchor keyrings. Note that some of these keyrings will be created only when first accessed.

Process keyrings
: Process credentials themselves reference keyrings with specific semantics. These keyrings are pinned as long as the set of credentials exists, which is usually as long as the process exists.

  There are three keyrings with different inheritance/sharing rules: the *session-keyring*(7) (inherited and shared by all child processes), the *process-keyring*(7) (shared by all threads in a process) and the *thread-keyring*(7) (specific to a particular thread).

  As an alternative to using the actual keyring IDs, in calls to *add_key*(2), *keyctl*(2), and *request_key*(2), the special keyring values **KEY_SPEC_SESSION_KEYRING**, **KEY_SPEC_PROCESS_KEYRING**, and **KEY_SPEC_THREAD_KEYRING** can be used to refer to the caller's own instances of these keyrings.

User keyrings
: Each UID known to the kernel has a record that contains two keyrings: the *user-keyring*(7) and the *user-session-keyring*(7). These exist for as long as the UID record in the kernel exists.

  As an alternative to using the actual keyring IDs, in calls to *add_key*(2), *keyctl*(2), and *request_key*(2), the special keyring values **KEY_SPEC_USER_KEYRING** and **KEY_SPEC_USER_SESSION_KEYRING** can be used to refer to the caller's own instances of these keyrings.

  A link to the user keyring is placed in a new session keyring by *pam_keyinit*(8) when a new login session is initiated.

Persistent keyrings
: There is a *persistent-keyring*(7) available to each UID known to the system. It may persist beyond the life of the UID record previously mentioned, but has an expiration time set such that it is automatically cleaned up after a set time. The persistent keyring permits, for example, *cron*(8) scripts to use credentials that are left in the persistent keyring after the user logs out.

  Note that the expiration time of the persistent keyring is reset every time the persistent key is requested.

Special keyrings
: There are special keyrings owned by the kernel that can anchor keys for special purposes. An example of this is the *system keyring* used for holding encryption keys for module signature verification.

  These special keyrings are usually closed to direct alteration by user space.

An originally planned "group keyring", for storing keys associated with each GID known to the kernel, is not so far implemented, is unlikely to be implemented. Nevertheless, the constant **KEY_SPEC_GROUP_KEYRING** has been defined for this keyring.

Possession

The concept of possession is important to understanding the keyrings security model. Whether a thread possesses a key is determined by the following rules:

(1) Any key or keyring that does not grant *search* permission to the caller is ignored in all the following rules.

(2) A thread possesses its *session-keyring*(7), *process-keyring*(7), and *thread-keyring*(7) directly because those keyrings are referred to by its credentials.

(3) If a keyring is possessed, then any key it links to is also possessed.

(4) If any key a keyring links to is itself a keyring, then rule (3) applies recursively.

(5) If a process is upcalled from the kernel to instantiate a key (see *request_key*(2)), then it also possesses the requester's keyrings as in rule (1) as if it were the requester.

Note that possession is not a fundamental property of a key, but must rather be calculated each time the

key is needed.

Possession is designed to allow set-user-ID programs run from, say a user's shell to access the user's keys. Granting permissions to the key possessor while denying them to the key owner and group allows the prevention of access to keys on the basis of UID and GID matches.

When it creates the session keyring, *pam_keyinit*(8) adds a link to the *user-keyring*(7), thus making the user keyring and anything it contains possessed by default.

Access rights

Each key has the following security-related attributes:

- The owning user ID
- The ID of a group that is permitted to access the key
- A security label
- A permissions mask

The permissions mask contains four sets of rights. The first three sets are mutually exclusive. One and only one will be in force for a particular access check. In order of descending priority, these three sets are:

user The set specifies the rights granted if the key's user ID matches the caller's filesystem user ID.

group The set specifies the rights granted if the user ID didn't match and the key's group ID matches the caller's filesystem GID or one of the caller's supplementary group IDs.

other The set specifies the rights granted if neither the key's user ID nor group ID matched.

The fourth set of rights is:

possessor
The set specifies the rights granted if a key is determined to be possessed by the caller.

The complete set of rights for a key is the union of whichever of the first three sets is applicable plus the fourth set if the key is possessed.

The set of rights that may be granted in each of the four masks is as follows:

view The attributes of the key may be read. This includes the type, description, and access rights (excluding the security label).

read For a key: the payload of the key may be read. For a keyring: the list of serial numbers (keys) to which the keyring has links may be read.

write The payload of the key may be updated and the key may be revoked. For a keyring, links may be added to or removed from the keyring, and the keyring may be cleared completely (all links are removed),

search For a key (or a keyring): the key may be found by a search. For a keyring: keys and keyrings that are linked to by the keyring may be searched.

link Links may be created from keyrings to the key. The initial link to a key that is established when the key is created doesn't require this permission.

setattr The ownership details and security label of the key may be changed, the key's expiration time may be set, and the key may be revoked.

In addition to access rights, any active Linux Security Module (LSM) may prevent access to a key if its policy so dictates. A key may be given a security label or other attribute by the LSM; this label is retrievable via *keyctl_get_security*(3)

See *keyctl_chown*(3), *keyctl_describe*(3), *keyctl_get_security*(3), *keyctl_setperm*(3), and *selinux*(8) for more information.

Searching for keys

One of the key features of the Linux key-management facility is the ability to find a key that a process is retaining. The *request_key*(2) system call is the primary point of access for user-space applications to find a key. (Internally, the kernel has something similar available for use by internal components that make use of keys.)

The search algorithm works as follows:

(1) The process keyrings are searched in the following order: the *thread-keyring*(7) if it exists, the *process-keyring*(7) if it exists, and then either the *session-keyring*(7) if it exists or the *user-session-keyring*(7) if that exists.

(2) If the caller was a process that was invoked by the *request_key*(2) upcall mechanism, then the keyrings of the original caller of *request_key*(2) will be searched as well.

(3) The search of a keyring tree is in breadth-first order: each keyring is searched first for a match, then the keyrings referred to by that keyring are searched.

(4) If a matching key is found that is valid, then the search terminates and that key is returned.

(5) If a matching key is found that has an error state attached, that error state is noted and the search continues.

(6) If no valid matching key is found, then the first noted error state is returned; otherwise, an **ENOKEY** error is returned.

It is also possible to search a specific keyring, in which case only steps (3) to (6) apply.

See *request_key*(2) and *keyctl_search*(3) for more information.

On-demand key creation

If a key cannot be found, *request_key*(2) will, if given a *callout_info* argument, create a new key and then upcall to user space to instantiate the key. This allows keys to be created on an as-needed basis.

Typically, this will involve the kernel creating a new process that executes the *request-key*(8) program, which will then execute the appropriate handler based on its configuration.

The handler is passed a special authorization key that allows it and only it to instantiate the new key. This is also used to permit searches performed by the handler program to also search the requester's keyrings.

See *request_key*(2), *keyctl_assume_authority*(3), *keyctl_instantiate*(3), *keyctl_negate*(3), *keyctl_reject*(3), *request-key*(8), and *request-key.conf* (5) for more information.

/proc files

The kernel provides various */proc* files that expose information about keys or define limits on key usage.

/proc/keys (since Linux 2.6.10)

This file exposes a list of the keys for which the reading thread has *view* permission, providing various information about each key. The thread need not possess the key for it to be visible in this file.

The only keys included in the list are those that grant *view* permission to the reading process (regardless of whether or not it possesses them). LSM security checks are still performed, and may filter out further keys that the process is not authorized to view.

An example of the data that one might see in this file (with the columns numbered for easy reference below) is the following:

```
  (1)     (2)     (3)(4)    (5)     (6)   (7)   (8)        (9)
009a2028 I--Q---   1 perm 3f010000  1000  1000 user     krb_ccache:primary: 12
1806c4ba I--Q---   1 perm 3f010000  1000  1000 keyring  _pid: 2
25d3a08f I--Q---   1 perm 1f3f0000  1000 65534 keyring  _uid_ses.1000: 1
28576bd8 I--Q---   3 perm 3f010000  1000  1000 keyring  _krb: 1
2c546d21 I--Q--- 190 perm 3f030000  1000  1000 keyring  _ses: 2
30a4e0be I------   4   2d 1f030000  1000 65534 keyring  _persistent.1000: 1
32100fab I--Q---   4 perm 1f3f0000  1000 65534 keyring  _uid.1000: 2
32a387ea I--Q---   1 perm 3f010000  1000  1000 keyring  _pid: 2
3ce56aea I--Q---   5 perm 3f030000  1000  1000 keyring  _ses: 1
```

The fields shown in each line of this file are as follows:

ID (1) The ID (serial number) of the key, expressed in hexadecimal.

Flags (2)

A set of flags describing the state of the key:

I The key has been instantiated.

R The key has been revoked.

D The key is dead (i.e., the key type has been unregistered). (A key may be briefly in this state during garbage collection.)

Q The key contributes to the user's quota.

U The key is under construction via a callback to user space; see *request-key*(2)

N The key is negatively instantiated.

i The key has been invalidated.

Usage (3)
: This is a count of the number of kernel credential structures that are pinning the key (approximately: the number of threads and open file references that refer to this key).

Timeout (4)
: The amount of time until the key will expire, expressed in human-readable form (weeks, days, hours, minutes, and seconds). The string *perm* here means that the key is permanent (no timeout). The string *expd* means that the key has already expired, but has not yet been garbage collected.

Permissions (5)
: The key permissions, expressed as four hexadecimal bytes containing, from left to right, the possessor, user, group, and other permissions. Within each byte, the permission bits are as follows:

0x01	*view*
0x02	*read*
0x04	*write*
0x08	*search*
0x10	*link*
0x20	*setattr*

UID (6)
: The user ID of the key owner.

GID (7)
: The group ID of the key. The value −1 here means that the key has no group ID; this can occur in certain circumstances for keys created by the kernel.

Type (8)
: The key type (user, keyring, etc.)

Description (9)
: The key description (name). This field contains descriptive information about the key. For most key types, it has the form

```
name[: extra-info]
```

The *name* subfield is the key's description (name). The optional *extra−info* field provides some further information about the key. The information that appears here depends on the key type, as follows:

"user" and *"logon"*
: The size in bytes of the key payload (expressed in decimal).

"keyring"
: The number of keys linked to the keyring, or the string *empty* if there are no keys linked to the keyring.

"big_key"
: The payload size in bytes, followed either by the string *[file]*, if the key payload exceeds the threshold that means that the payload is stored in a (swappable) *tmpfs*(5) filesystem, or otherwise the string *[buff]*, indicating that the key is small enough to reside in kernel memory.

For the *".request_key_auth"* key type (authorization key; see *request_key*(2)), the description field has the form shown in the following example:

```
key:c9a9b19 pid:28880 ci:10
```

The three subfields are as follows:

key The hexadecimal ID of the key being instantiated in the requesting program.

pid The PID of the requesting program.

ci The length of the callout data with which the requested key should be instantiated (i.e., the length of the payload associated with the authorization key).

/proc/key−users (since Linux 2.6.10)
This file lists various information for each user ID that has at least one key on the system. An example of the data that one might see in this file is the following:

```
    0:    10 9/9 2/1000000 22/25000000
   42:     9 9/9 8/200 106/20000
 1000:    11 11/11 10/200 271/20000
```

The fields shown in each line are as follows:

uid The user ID.

usage This is a kernel-internal usage count for the kernel structure used to record key users.

nkeys/*nikeys*
The total number of keys owned by the user, and the number of those keys that have been instantiated.

qnkeys/*maxkeys*
The number of keys owned by the user, and the maximum number of keys that the user may own.

qnbytes/*maxbytes*
The number of bytes consumed in payloads of the keys owned by this user, and the upper limit on the number of bytes in key payloads for that user.

/proc/sys/kernel/keys/gc_delay (since Linux 2.6.32)
The value in this file specifies the interval, in seconds, after which revoked and expired keys will be garbage collected. The purpose of having such an interval is so that there is a window of time where user space can see an error (respectively **EKEYREVOKED** and **EKEYEXPIRED**) that indicates what happened to the key.

The default value in this file is 300 (i.e., 5 minutes).

/proc/sys/kernel/keys/persistent_keyring_expiry (since Linux 3.13)
This file defines an interval, in seconds, to which the persistent keyring's expiration timer is reset each time the keyring is accessed (via *keyctl_get_persistent*(3) or the *keyctl*(2) **KEYCTL_GET_PERSISTENT** operation.)

The default value in this file is 259200 (i.e., 3 days).

The following files (which are writable by privileged processes) are used to enforce quotas on the number of keys and number of bytes of data that can be stored in key payloads:

/proc/sys/kernel/keys/maxbytes (since Linux 2.6.26)
This is the maximum number of bytes of data that a nonroot user can hold in the payloads of the keys owned by the user.

The default value in this file is 20,000.

/proc/sys/kernel/keys/maxkeys (since Linux 2.6.26)
This is the maximum number of keys that a nonroot user may own.

The default value in this file is 200.

/proc/sys/kernel/keys/root_maxbytes (since Linux 2.6.26)
This is the maximum number of bytes of data that the root user (UID 0 in the root user namespace) can hold in the payloads of the keys owned by root.

The default value in this file is 25,000,000 (20,000 before Linux 3.17).

/proc/sys/kernel/keys/root_maxkeys (since Linux 2.6.26)
This is the maximum number of keys that the root user (UID 0 in the root user namespace) may own.

The default value in this file is 1,000,000 (200 before Linux 3.17).

With respect to keyrings, note that each link in a keyring consumes 4 bytes of the keyring payload.

Users

The Linux key-management facility has a number of users and usages, but is not limited to those that already exist.

In-kernel users of this facility include:

Network filesystems - DNS
The kernel uses the upcall mechanism provided by the keys to upcall to user space to do DNS lookups and then to cache the results.

AF_RXRPC and kAFS - Authentication
The AF_RXRPC network protocol and the in-kernel AFS filesystem use keys to store the ticket needed to do secured or encrypted traffic. These are then looked up by network operations on AF_RXRPC and filesystem operations on kAFS.

NFS - User ID mapping
The NFS filesystem uses keys to store mappings of foreign user IDs to local user IDs.

CIFS - Password
The CIFS filesystem uses keys to store passwords for accessing remote shares.

Module verification
The kernel build process can be made to cryptographically sign modules. That signature is then checked when a module is loaded.

User-space users of this facility include:

Kerberos key storage
The MIT Kerberos 5 facility (libkrb5) can use keys to store authentication tokens which can be made to be automatically cleaned up a set time after the user last uses them, but until then permits them to hang around after the user has logged out so that *cron*(8) scripts can use them.

SEE ALSO

keyctl(1), *add_key*(2), *keyctl*(2), *request_key*(2), *keyctl*(3), *keyutils*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7), *pam_keyinit*(8), *request-key*(8)

The kernel source files *Documentation/crypto/asymmetric−keys.txt* and under *Documentation/security/keys* (or, before Linux 4.13, in the file *Documentation/security/keys.txt*).

NAME

koi8-r − Russian character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

RFC 1489 defines an 8-bit character set, KOI8-R. KOI8-R encodes the characters used in Russian.

KOI8-R characters

The following table displays the characters in KOI8-R that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
200	128	80	─	BOX DRAWINGS LIGHT HORIZONTAL
201	129	81	│	BOX DRAWINGS LIGHT VERTICAL
202	130	82	┌	BOX DRAWINGS LIGHT DOWN AND RIGHT
203	131	83	┐	BOX DRAWINGS LIGHT DOWN AND LEFT
204	132	84	└	BOX DRAWINGS LIGHT UP AND RIGHT
205	133	85	┘	BOX DRAWINGS LIGHT UP AND LEFT
206	134	86	├	BOX DRAWINGS LIGHT VERTICAL AND RIGHT
207	135	87	┤	BOX DRAWINGS LIGHT VERTICAL AND LEFT
210	136	88	┬	BOX DRAWINGS LIGHT DOWN AND HORIZONTAL
211	137	89	┴	BOX DRAWINGS LIGHT UP AND HORIZONTAL
212	138	8A	┼	BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL
213	139	8B	▀	UPPER HALF BLOCK
214	140	8C	▄	LOWER HALF BLOCK
215	141	8D	█	FULL BLOCK
216	142	8E	▌	LEFT HALF BLOCK
217	143	8F	▐	RIGHT HALF BLOCK
220	144	90	░	LIGHT SHADE
221	145	91	▒	MEDIUM SHADE
222	146	92	▓	DARK SHADE
223	147	93	⌠	TOP HALF INTEGRAL
224	148	94	■	BLACK SQUARE
225	149	95		BULLET OPERATOR
226	150	96	√	SQUARE ROOT
227	151	97	≈	ALMOST EQUAL TO
230	152	98	≤	LESS-THAN OR EQUAL TO
231	153	99	≥	GREATER-THAN OR EQUAL TO
232	154	9A		NO-BREAK SPACE
233	155	9B	⌡	BOTTOM HALF INTEGRAL
234	156	9C	°	DEGREE SIGN
235	157	9D	²	SUPERSCRIPT TWO
236	158	9E	·	MIDDLE DOT
237	159	9F	÷	DIVISION SIGN
240	160	A0	═	BOX DRAWINGS DOUBLE HORIZONTAL
241	161	A1	║	BOX DRAWINGS DOUBLE VERTICAL
242	162	A2	╒	BOX DRAWINGS DOWN SINGLE AND RIGHT DOUBLE
243	163	A3	ё	CYRILLIC SMALL LETTER IO
244	164	A4	╓	BOX DRAWINGS DOWN DOUBLE AND RIGHT SINGLE
245	165	A5	╔	BOX DRAWINGS DOUBLE DOWN AND RIGHT
246	166	A6	╕	BOX DRAWINGS DOWN SINGLE AND LEFT DOUBLE
247	167	A7	╖	BOX DRAWINGS DOWN DOUBLE AND LEFT SINGLE
250	168	A8	╗	BOX DRAWINGS DOUBLE DOWN AND LEFT
251	169	A9	╘	BOX DRAWINGS UP SINGLE AND RIGHT DOUBLE
252	170	AA	╙	BOX DRAWINGS UP DOUBLE AND RIGHT SINGLE
253	171	AB	╚	BOX DRAWINGS DOUBLE UP AND RIGHT
254	172	AC	╛	BOX DRAWINGS UP SINGLE AND LEFT DOUBLE
255	173	AD	╜	BOX DRAWINGS UP DOUBLE AND LEFT SINGLE
256	174	AE	╝	BOX DRAWINGS DOUBLE UP AND LEFT
257	175	AF	╞	BOX DRAWINGS VERTICAL SINGLE AND RIGHT DOUBLE
260	176	B0	╟	BOX DRAWINGS VERTICAL DOUBLE AND RIGHT SINGLE

261	177	B1	╠	BOX DRAWINGS DOUBLE VERTICAL AND RIGHT
262	178	B2	╡	BOX DRAWINGS VERTICAL SINGLE AND LEFT DOUBLE
263	179	B3	Ё	CYRILLIC CAPITAL LETTER IO
264	180	B4	╢	BOX DRAWINGS VERTICAL DOUBLE AND LEFT SINGLE
265	181	B5	╣	BOX DRAWINGS DOUBLE VERTICAL AND LEFT
266	182	B6	╤	BOX DRAWINGS DOWN SINGLE AND HORIZONTAL DOUBLE
267	183	B7	╥	BOX DRAWINGS DOWN DOUBLE AND HORIZONTAL SINGLE
270	184	B8	╦	BOX DRAWINGS DOUBLE DOWN AND HORIZONTAL
271	185	B9	╧	BOX DRAWINGS UP SINGLE AND HORIZONTAL DOUBLE
272	186	BA	╨	BOX DRAWINGS UP DOUBLE AND HORIZONTAL SINGLE
273	187	BB	╩	BOX DRAWINGS DOUBLE UP AND HORIZONTAL
274	188	BC	╪	BOX DRAWINGS VERTICAL SINGLE AND HORIZONTAL DOUBLE
275	189	BD	╫	BOX DRAWINGS VERTICAL DOUBLE AND HORIZONTAL SINGLE
276	190	BE	╬	BOX DRAWINGS DOUBLE VERTICAL AND HORIZONTAL
277	191	BF	©	COPYRIGHT SIGN
300	192	C0	ю	CYRILLIC SMALL LETTER YU
301	193	C1	а	CYRILLIC SMALL LETTER A
302	194	C2	б	CYRILLIC SMALL LETTER BE
303	195	C3	ц	CYRILLIC SMALL LETTER TSE
304	196	C4	д	CYRILLIC SMALL LETTER DE
305	197	C5	е	CYRILLIC SMALL LETTER IE
306	198	C6	ф	CYRILLIC SMALL LETTER EF
307	199	C7	г	CYRILLIC SMALL LETTER GHE
310	200	C8	х	CYRILLIC SMALL LETTER HA
311	201	C9	и	CYRILLIC SMALL LETTER I
312	202	CA	й	CYRILLIC SMALL LETTER SHORT I
313	203	CB	к	CYRILLIC SMALL LETTER KA
314	204	CC	л	CYRILLIC SMALL LETTER EL
315	205	CD	м	CYRILLIC SMALL LETTER EM
316	206	CE	н	CYRILLIC SMALL LETTER EN
317	207	CF	о	CYRILLIC SMALL LETTER O
320	208	D0	п	CYRILLIC SMALL LETTER PE
321	209	D1	я	CYRILLIC SMALL LETTER YA
322	210	D2	р	CYRILLIC SMALL LETTER ER
323	211	D3	с	CYRILLIC SMALL LETTER ES
324	212	D4	т	CYRILLIC SMALL LETTER TE
325	213	D5	у	CYRILLIC SMALL LETTER U
326	214	D6	ж	CYRILLIC SMALL LETTER ZHE
327	215	D7	в	CYRILLIC SMALL LETTER VE
330	216	D8	ь	CYRILLIC SMALL LETTER SOFT SIGN
331	217	D9	ы	CYRILLIC SMALL LETTER YERU
332	218	DA	з	CYRILLIC SMALL LETTER ZE
333	219	DB	ш	CYRILLIC SMALL LETTER SHA
334	220	DC	э	CYRILLIC SMALL LETTER E
335	221	DD	щ	CYRILLIC SMALL LETTER SHCHA
336	222	DE	ч	CYRILLIC SMALL LETTER CHE
337	223	DF	ъ	CYRILLIC SMALL LETTER HARD SIGN
340	224	E0	Ю	CYRILLIC CAPITAL LETTER YU
341	225	E1	А	CYRILLIC CAPITAL LETTER A
342	226	E2	Б	CYRILLIC CAPITAL LETTER BE
343	227	E3	Ц	CYRILLIC CAPITAL LETTER TSE
344	228	E4	Д	CYRILLIC CAPITAL LETTER DE
345	229	E5	Е	CYRILLIC CAPITAL LETTER IE
346	230	E6	Ф	CYRILLIC CAPITAL LETTER EF
347	231	E7	Г	CYRILLIC CAPITAL LETTER GHE
350	232	E8	Х	CYRILLIC CAPITAL LETTER HA

351	233	E9	И	CYRILLIC CAPITAL LETTER I
352	234	EA	Й	CYRILLIC CAPITAL LETTER SHORT I
353	235	EB	К	CYRILLIC CAPITAL LETTER KA
354	236	EC	Л	CYRILLIC CAPITAL LETTER EL
355	237	ED	М	CYRILLIC CAPITAL LETTER EM
356	238	EE	Н	CYRILLIC CAPITAL LETTER EN
357	239	EF	О	CYRILLIC CAPITAL LETTER O
360	240	F0	П	CYRILLIC CAPITAL LETTER PE
361	241	F1	Я	CYRILLIC CAPITAL LETTER YA
362	242	F2	Р	CYRILLIC CAPITAL LETTER ER
363	243	F3	С	CYRILLIC CAPITAL LETTER ES
364	244	F4	Т	CYRILLIC CAPITAL LETTER TE
365	245	F5	У	CYRILLIC CAPITAL LETTER U
366	246	F6	Ж	CYRILLIC CAPITAL LETTER ZHE
367	247	F7	В	CYRILLIC CAPITAL LETTER VE
370	248	F8	Ь	CYRILLIC CAPITAL LETTER SOFT SIGN
371	249	F9	Ы	CYRILLIC CAPITAL LETTER YERU
372	250	FA	З	CYRILLIC CAPITAL LETTER ZE
373	251	FB	Ш	CYRILLIC CAPITAL LETTER SHA
374	252	FC	Э	CYRILLIC CAPITAL LETTER E
375	253	FD	Щ	CYRILLIC CAPITAL LETTER SHCHA
376	254	FE	Ч	CYRILLIC CAPITAL LETTER CHE
377	255	FF	Ъ	CYRILLIC CAPITAL LETTER HARD SIGN

NOTES

The differences with KOI8-U are in the hex positions A4, A6, A7, AD, B4, B6, B7, and BD.

SEE ALSO

ascii(7), *charsets*(7), *cp1251*(7), *iso_8859-5*(7), *koi8-u*(7), *utf-8*(7)

NAME

koi8-u − Ukrainian character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

RFC 2310 defines an 8-bit character set, KOI8-U. KOI8-U encodes the characters used in Ukrainian and Byelorussian.

KOI8-U characters

The following table displays the characters in KOI8-U that are printable and unlisted in the *ascii*(7) manual page.

Oct	Dec	Hex	Char	Description
200	128	80	─	BOX DRAWINGS LIGHT HORIZONTAL
201	129	81	│	BOX DRAWINGS LIGHT VERTICAL
202	130	82	┌	BOX DRAWINGS LIGHT DOWN AND RIGHT
203	131	83	┐	BOX DRAWINGS LIGHT DOWN AND LEFT
204	132	84	└	BOX DRAWINGS LIGHT UP AND RIGHT
205	133	85	┘	BOX DRAWINGS LIGHT UP AND LEFT
206	134	86	├	BOX DRAWINGS LIGHT VERTICAL AND RIGHT
207	135	87	┤	BOX DRAWINGS LIGHT VERTICAL AND LEFT
210	136	88	┬	BOX DRAWINGS LIGHT DOWN AND HORIZONTAL
211	137	89	┴	BOX DRAWINGS LIGHT UP AND HORIZONTAL
212	138	8A	┼	BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL
213	139	8B	▀	UPPER HALF BLOCK
214	140	8C	▄	LOWER HALF BLOCK
215	141	8D	█	FULL BLOCK
216	142	8E	▌	LEFT HALF BLOCK
217	143	8F	▐	RIGHT HALF BLOCK
220	144	90	░	LIGHT SHADE
221	145	91	▒	MEDIUM SHADE
222	146	92	▓	DARK SHADE
223	147	93	⌠	TOP HALF INTEGRAL
224	148	94	■	BLACK SQUARE
225	149	95		BULLET OPERATOR
226	150	96	√	SQUARE ROOT
227	151	97	≈	ALMOST EQUAL TO
230	152	98	≤	LESS-THAN OR EQUAL TO
231	153	99	≥	GREATER-THAN OR EQUAL TO
232	154	9A		NO-BREAK SPACE
233	155	9B	⌡	BOTTOM HALF INTEGRAL
234	156	9C	°	DEGREE SIGN
235	157	9D	²	SUPERSCRIPT TWO
236	158	9E	·	MIDDLE DOT
237	159	9F	÷	DIVISION SIGN
240	160	A0	═	BOX DRAWINGS DOUBLE HORIZONTAL
241	161	A1	║	BOX DRAWINGS DOUBLE VERTICAL
242	162	A2	╒	BOX DRAWINGS DOWN SINGLE AND RIGHT DOUBLE
243	163	A3	ё	CYRILLIC SMALL LETTER IO
244	164	A4	є	CYRILLIC SMALL LETTER UKRAINIAN IE
245	165	A5	╔	BOX DRAWINGS DOUBLE DOWN AND RIGHT
246	166	A6	і	CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I
247	167	A7	ї	CYRILLIC SMALL LETTER YI (Ukrainian)
250	168	A8	╗	BOX DRAWINGS DOUBLE DOWN AND LEFT
251	169	A9	╘	BOX DRAWINGS UP SINGLE AND RIGHT DOUBLE
252	170	AA	╚	BOX DRAWINGS UP DOUBLE AND RIGHT SINGLE
253	171	AB	╝	BOX DRAWINGS DOUBLE UP AND RIGHT
254	172	AC	╛	BOX DRAWINGS UP SINGLE AND LEFT DOUBLE
255	173	AD	ґ	CYRILLIC SMALL LETTER GHE WITH UPTURN
256	174	AE	╩	BOX DRAWINGS DOUBLE UP AND LEFT

257	175	AF	╞	BOX DRAWINGS VERTICAL SINGLE AND RIGHT DOUBLE
260	176	B0	╠	BOX DRAWINGS VERTICAL DOUBLE AND RIGHT SINGLE
261	177	B1	╡	BOX DRAWINGS DOUBLE VERTICAL AND RIGHT
262	178	B2	╢	BOX DRAWINGS VERTICAL SINGLE AND LEFT DOUBLE
263	179	B3	Ё	CYRILLIC CAPITAL LETTER IO
264	180	B4	Є	CYRILLIC CAPITAL LETTER UKRAINIAN IE
265	181	B5	╣	BOX DRAWINGS DOUBLE VERTICAL AND LEFT
266	182	B6	І	CYRILLIC CAPITAL LETTER BYELORUSSIAN-UKRAINIAN I
267	183	B7	Ї	CYRILLIC CAPITAL LETTER YI (Ukrainian)
270	184	B8	╦	BOX DRAWINGS DOUBLE DOWN AND HORIZONTAL
271	185	B9	╧	BOX DRAWINGS UP SINGLE AND HORIZONTAL DOUBLE
272	186	BA	╨	BOX DRAWINGS UP DOUBLE AND HORIZONTAL SINGLE
273	187	BB	╩	BOX DRAWINGS DOUBLE UP AND HORIZONTAL
274	188	BC	╪	BOX DRAWINGS VERTICAL SINGLE AND HORIZONTAL DOUBLE
275	189	BD	Ґ	CYRILLIC CAPITAL LETTER GHE WITH UPTURN
276	190	BE	╬	BOX DRAWINGS DOUBLE VERTICAL AND HORIZONTAL
277	191	BF	©	COPYRIGHT SIGN
300	192	C0	ю	CYRILLIC SMALL LETTER YU
301	193	C1	а	CYRILLIC SMALL LETTER A
302	194	C2	б	CYRILLIC SMALL LETTER BE
303	195	C3	ц	CYRILLIC SMALL LETTER TSE
304	196	C4	д	CYRILLIC SMALL LETTER DE
305	197	C5	е	CYRILLIC SMALL LETTER IE
306	198	C6	ф	CYRILLIC SMALL LETTER EF
307	199	C7	г	CYRILLIC SMALL LETTER GHE
310	200	C8	х	CYRILLIC SMALL LETTER HA
311	201	C9	и	CYRILLIC SMALL LETTER I
312	202	CA	й	CYRILLIC SMALL LETTER SHORT I
313	203	CB	к	CYRILLIC SMALL LETTER KA
314	204	CC	л	CYRILLIC SMALL LETTER EL
315	205	CD	м	CYRILLIC SMALL LETTER EM
316	206	CE	н	CYRILLIC SMALL LETTER EN
317	207	CF	о	CYRILLIC SMALL LETTER O
320	208	D0	п	CYRILLIC SMALL LETTER PE
321	209	D1	я	CYRILLIC SMALL LETTER YA
322	210	D2	р	CYRILLIC SMALL LETTER ER
323	211	D3	с	CYRILLIC SMALL LETTER ES
324	212	D4	т	CYRILLIC SMALL LETTER TE
325	213	D5	у	CYRILLIC SMALL LETTER U
326	214	D6	ж	CYRILLIC SMALL LETTER ZHE
327	215	D7	в	CYRILLIC SMALL LETTER VE
330	216	D8	ь	CYRILLIC SMALL LETTER SOFT SIGN
331	217	D9	ы	CYRILLIC SMALL LETTER YERU
332	218	DA	з	CYRILLIC SMALL LETTER ZE
333	219	DB	ш	CYRILLIC SMALL LETTER SHA
334	220	DC	э	CYRILLIC SMALL LETTER E
335	221	DD	щ	CYRILLIC SMALL LETTER SHCHA
336	222	DE	ч	CYRILLIC SMALL LETTER CHE
337	223	DF	ъ	CYRILLIC SMALL LETTER HARD SIGN
340	224	E0	Ю	CYRILLIC CAPITAL LETTER YU
341	225	E1	А	CYRILLIC CAPITAL LETTER A
342	226	E2	Б	CYRILLIC CAPITAL LETTER BE
343	227	E3	Ц	CYRILLIC CAPITAL LETTER TSE
344	228	E4	Д	CYRILLIC CAPITAL LETTER DE
345	229	E5	Е	CYRILLIC CAPITAL LETTER IE
346	230	E6	Ф	CYRILLIC CAPITAL LETTER EF

347	231	E7	Г	CYRILLIC CAPITAL LETTER GHE
350	232	E8	Х	CYRILLIC CAPITAL LETTER HA
351	233	E9	И	CYRILLIC CAPITAL LETTER I
352	234	EA	Й	CYRILLIC CAPITAL LETTER SHORT I
353	235	EB	К	CYRILLIC CAPITAL LETTER KA
354	236	EC	Л	CYRILLIC CAPITAL LETTER EL
355	237	ED	М	CYRILLIC CAPITAL LETTER EM
356	238	EE	Н	CYRILLIC CAPITAL LETTER EN
357	239	EF	О	CYRILLIC CAPITAL LETTER O
360	240	F0	П	CYRILLIC CAPITAL LETTER PE
361	241	F1	Я	CYRILLIC CAPITAL LETTER YA
362	242	F2	Р	CYRILLIC CAPITAL LETTER ER
363	243	F3	С	CYRILLIC CAPITAL LETTER ES
364	244	F4	Т	CYRILLIC CAPITAL LETTER TE
365	245	F5	У	CYRILLIC CAPITAL LETTER U
366	246	F6	Ж	CYRILLIC CAPITAL LETTER ZHE
367	247	F7	В	CYRILLIC CAPITAL LETTER VE
370	248	F8	Ь	CYRILLIC CAPITAL LETTER SOFT SIGN
371	249	F9	Ы	CYRILLIC CAPITAL LETTER YERU
372	250	FA	З	CYRILLIC CAPITAL LETTER ZE
373	251	FB	Ш	CYRILLIC CAPITAL LETTER SHA
374	252	FC	Э	CYRILLIC CAPITAL LETTER E
375	253	FD	Щ	CYRILLIC CAPITAL LETTER SHCHA
376	254	FE	Ч	CYRILLIC CAPITAL LETTER CHE
377	255	FF	Ъ	CYRILLIC CAPITAL LETTER HARD SIGN

NOTES

The differences from KOI8-R are in the hex positions A4, A6, A7, AD, B4, B6, B7, and BD.

SEE ALSO

ascii(7), *charsets*(7), *cp1251*(7), *iso_8859-5*(7), *koi8-r*(7), *utf-8*(7)

NAME

Landlock − unprivileged access-control

DESCRIPTION

Landlock is an access-control system that enables any processes to securely restrict themselves and their future children. Because Landlock is a stackable Linux Security Module (LSM), it makes it possible to create safe security sandboxes as new security layers in addition to the existing system-wide access-controls. This kind of sandbox is expected to help mitigate the security impact of bugs, and unexpected or malicious behaviors in applications.

A Landlock security policy is a set of access rights (e.g., open a file in read-only, make a directory, etc.) tied to a file hierarchy. Such policy can be configured and enforced by processes for themselves using three system calls:

- *landlock_create_ruleset*(2) creates a new ruleset;
- *landlock_add_rule*(2) adds a new rule to a ruleset;
- *landlock_restrict_self*(2) enforces a ruleset on the calling thread.

To be able to use these system calls, the running kernel must support Landlock and it must be enabled at boot time.

Landlock rules

A Landlock rule describes an action on an object. An object is currently a file hierarchy, and the related filesystem actions are defined with access rights (see *landlock_add_rule*(2)). A set of rules is aggregated in a ruleset, which can then restrict the thread enforcing it, and its future children.

Filesystem actions

These flags enable to restrict a sandboxed process to a set of actions on files and directories. Files or directories opened before the sandboxing are not subject to these restrictions. See *landlock_add_rule*(2) and *landlock_create_ruleset*(2) for more context.

A file can only receive these access rights:

LANDLOCK_ACCESS_FS_EXECUTE
: Execute a file.

LANDLOCK_ACCESS_FS_WRITE_FILE
: Open a file with write access.

  When opening files for writing, you will often additionally need the **LANDLOCK_ACCESS_FS_TRUNCATE** right. In many cases, these system calls truncate existing files when overwriting them (e.g., *creat*(2)).

LANDLOCK_ACCESS_FS_READ_FILE
: Open a file with read access.

LANDLOCK_ACCESS_FS_TRUNCATE
: Truncate a file with *truncate*(2), *ftruncate*(2), *creat*(2), or *open*(2) with **O_TRUNC**. Whether an opened file can be truncated with *ftruncate*(2) is determined during *open*(2), in the same way as read and write permissions are checked during *open*(2) using **LANDLOCK_ACCESS_FS_READ_FILE** and **LANDLOCK_ACCESS_FS_WRITE_FILE**. This access right is available since the third version of the Landlock ABI.

A directory can receive access rights related to files or directories. The following access right is applied to the directory itself, and the directories beneath it:

LANDLOCK_ACCESS_FS_READ_DIR
: Open a directory or list its content.

However, the following access rights only apply to the content of a directory, not the directory itself:

LANDLOCK_ACCESS_FS_REMOVE_DIR
: Remove an empty directory or rename one.

LANDLOCK_ACCESS_FS_REMOVE_FILE
: Unlink (or rename) a file.

LANDLOCK_ACCESS_FS_MAKE_CHAR
Create (or rename or link) a character device.

LANDLOCK_ACCESS_FS_MAKE_DIR
Create (or rename) a directory.

LANDLOCK_ACCESS_FS_MAKE_REG
Create (or rename or link) a regular file.

LANDLOCK_ACCESS_FS_MAKE_SOCK
Create (or rename or link) a UNIX domain socket.

LANDLOCK_ACCESS_FS_MAKE_FIFO
Create (or rename or link) a named pipe.

LANDLOCK_ACCESS_FS_MAKE_BLOCK
Create (or rename or link) a block device.

LANDLOCK_ACCESS_FS_MAKE_SYM
Create (or rename or link) a symbolic link.

LANDLOCK_ACCESS_FS_REFER
Link or rename a file from or to a different directory (i.e., reparent a file hierarchy).

This access right is available since the second version of the Landlock ABI.

This is the only access right which is denied by default by any ruleset, even if the right is not specified as handled at ruleset creation time. The only way to make a ruleset grant this right is to explicitly allow it for a specific directory by adding a matching rule to the ruleset.

In particular, when using the first Landlock ABI version, Landlock will always deny attempts to reparent files between different directories.

In addition to the source and destination directories having the **LANDLOCK_ACCESS_FS_REFER** access right, the attempted link or rename operation must meet the following constraints:

- The reparented file may not gain more access rights in the destination directory than it previously had in the source directory. If this is attempted, the operation results in an **EXDEV** error.
- When linking or renaming, the **LANDLOCK_ACCESS_FS_MAKE_*** right for the respective file type must be granted for the destination directory. Otherwise, the operation results in an **EACCES** error.
- When renaming, the **LANDLOCK_ACCESS_FS_REMOVE_*** right for the respective file type must be granted for the source directory. Otherwise, the operation results in an **EACCES** error.

If multiple requirements are not met, the **EACCES** error code takes precedence over **EXDEV**.

Layers of file path access rights
Each time a thread enforces a ruleset on itself, it updates its Landlock domain with a new layer of policy. Indeed, this complementary policy is composed with the potentially other rulesets already restricting this thread. A sandboxed thread can then safely add more constraints to itself with a new enforced ruleset.

One policy layer grants access to a file path if at least one of its rules encountered on the path grants the access. A sandboxed thread can only access a file path if all its enforced policy layers grant the access as well as all the other system access controls (e.g., filesystem DAC, other LSM policies, etc.).

Bind mounts and OverlayFS
Landlock enables restricting access to file hierarchies, which means that these access rights can be propagated with bind mounts (cf. *mount_namespaces*(7)) but not with OverlayFS.

A bind mount mirrors a source file hierarchy to a destination. The destination hierarchy is then composed of the exact same files, on which Landlock rules can be tied, either via the source or the destination path. These rules restrict access when they are encountered on a path, which means that they can restrict access to multiple file hierarchies at the same time, whether these hierarchies are the result of bind mounts or not.

An OverlayFS mount point consists of upper and lower layers. These layers are combined in a merge directory, result of the mount point. This merge hierarchy may include files from the upper and lower layers, but modifications performed on the merge hierarchy only reflect on the upper layer. From a Landlock policy point of view, each of the OverlayFS layers and merge hierarchies is standalone and contains its own set of files and directories, which is different from a bind mount. A policy restricting an OverlayFS layer will not restrict the resulted merged hierarchy, and vice versa. Landlock users should then only think about file hierarchies they want to allow access to, regardless of the underlying filesystem.

Inheritance

Every new thread resulting from a *clone*(2) inherits Landlock domain restrictions from its parent. This is similar to the *seccomp*(2) inheritance or any other LSM dealing with tasks' *credentials*(7). For instance, one process's thread may apply Landlock rules to itself, but they will not be automatically applied to other sibling threads (unlike POSIX thread credential changes, cf. *nptl*(7)).

When a thread sandboxes itself, we have the guarantee that the related security policy will stay enforced on all this thread's descendants. This allows creating standalone and modular security policies per application, which will automatically be composed between themselves according to their run-time parent policies.

Ptrace restrictions

A sandboxed process has less privileges than a non-sandboxed process and must then be subject to additional restrictions when manipulating another process. To be allowed to use *ptrace*(2) and related syscalls on a target process, a sandboxed process should have a subset of the target process rules, which means the tracee must be in a sub-domain of the tracer.

Truncating files

The operations covered by **LANDLOCK_ACCESS_FS_WRITE_FILE** and **LANDLOCK_ACCESS_FS_TRUNCATE** both change the contents of a file and sometimes overlap in non-intuitive ways. It is recommended to always specify both of these together.

A particularly surprising example is *creat*(2). The name suggests that this system call requires the rights to create and write files. However, it also requires the truncate right if an existing file under the same name is already present.

It should also be noted that truncating files does not require the **LANDLOCK_ACCESS_FS_WRITE_FILE** right. Apart from the *truncate*(2) system call, this can also be done through *open*(2) with the flags *O_RDONLY* | *O_TRUNC*.

When opening a file, the availability of the **LANDLOCK_ACCESS_FS_TRUNCATE** right is associated with the newly created file descriptor and will be used for subsequent truncation attempts using *ftruncate*(2). The behavior is similar to opening a file for reading or writing, where permissions are checked during *open*(2), but not during the subsequent *read*(2) and *write*(2) calls.

As a consequence, it is possible to have multiple open file descriptors for the same file, where one grants the right to truncate the file and the other does not. It is also possible to pass such file descriptors between processes, keeping their Landlock properties, even when these processes do not have an enforced Landlock ruleset.

VERSIONS

Landlock was introduced in Linux 5.13.

To determine which Landlock features are available, users should query the Landlock ABI version:

ABI	Kernel	Newly introduced access rights
1	5.13	**LANDLOCK_ACCESS_FS_EXECUTE** **LANDLOCK_ACCESS_FS_WRITE_FILE** **LANDLOCK_ACCESS_FS_READ_FILE** **LANDLOCK_ACCESS_FS_READ_DIR** **LANDLOCK_ACCESS_FS_REMOVE_DIR** **LANDLOCK_ACCESS_FS_REMOVE_FILE** **LANDLOCK_ACCESS_FS_MAKE_CHAR** **LANDLOCK_ACCESS_FS_MAKE_DIR** **LANDLOCK_ACCESS_FS_MAKE_REG** **LANDLOCK_ACCESS_FS_MAKE_SOCK** **LANDLOCK_ACCESS_FS_MAKE_FIFO** **LANDLOCK_ACCESS_FS_MAKE_BLOCK** **LANDLOCK_ACCESS_FS_MAKE_SYM**
2	5.19	**LANDLOCK_ACCESS_FS_REFER**
3	6.2	**LANDLOCK_ACCESS_FS_TRUNCATE**

Users should use the Landlock ABI version rather than the kernel version to determine which features are available. The mainline kernel versions listed here are only included for orientation. Kernels from other sources may contain backported features, and their version numbers may not match.

To query the running kernel's Landlock ABI version, programs may pass the **LANDLOCK_CREATE_RULESET_VERSION** flag to *landlock_create_ruleset*(2).

When building fallback mechanisms for compatibility with older kernels, users are advised to consider the special semantics of the **LANDLOCK_ACCESS_FS_REFER** access right: In ABI v1, linking and moving of files between different directories is always forbidden, so programs relying on such operations are only compatible with Landlock ABI v2 and higher.

NOTES

Landlock is enabled by **CONFIG_SECURITY_LANDLOCK**. The *lsm=lsm1,...,lsmN* command line parameter controls the sequence of the initialization of Linux Security Modules. It must contain the string *landlock* to enable Landlock. If the command line parameter is not specified, the initialization falls back to the value of the deprecated *security=* command line parameter and further to the value of **CONFIG_LSM**. We can check that Landlock is enabled by looking for *landlock: Up and running.* in kernel logs.

CAVEATS

It is currently not possible to restrict some file-related actions accessible through these system call families: *chdir*(2), *stat*(2), *flock*(2), *chmod*(2), *chown*(2), *setxattr*(2), *utime*(2), *ioctl*(2), *fcntl*(2), *access*(2). Future Landlock evolutions will enable to restrict them.

EXAMPLES

We first need to create the ruleset that will contain our rules.

For this example, the ruleset will contain rules that only allow read actions, but write actions will be denied. The ruleset then needs to handle both of these kinds of actions. See the **DESCRIPTION** section for the description of filesystem actions.

```
struct landlock_ruleset_attr attr = {0};
int ruleset_fd;

attr.handled_access_fs =
        LANDLOCK_ACCESS_FS_EXECUTE |
        LANDLOCK_ACCESS_FS_WRITE_FILE |
        LANDLOCK_ACCESS_FS_READ_FILE |
        LANDLOCK_ACCESS_FS_READ_DIR |
        LANDLOCK_ACCESS_FS_REMOVE_DIR |
        LANDLOCK_ACCESS_FS_REMOVE_FILE |
        LANDLOCK_ACCESS_FS_MAKE_CHAR |
        LANDLOCK_ACCESS_FS_MAKE_DIR |
        LANDLOCK_ACCESS_FS_MAKE_REG |
        LANDLOCK_ACCESS_FS_MAKE_SOCK |
```

```
            LANDLOCK_ACCESS_FS_MAKE_FIFO |
            LANDLOCK_ACCESS_FS_MAKE_BLOCK |
            LANDLOCK_ACCESS_FS_MAKE_SYM |
            LANDLOCK_ACCESS_FS_REFER |
            LANDLOCK_ACCESS_FS_TRUNCATE;
```

To be compatible with older Linux versions, we detect the available Landlock ABI version, and only use the available subset of access rights:

```
/*
 * Table of available file system access rights by ABI version,
 * numbers hardcoded to keep the example short.
 */
__u64 landlock_fs_access_rights[] = {
    (LANDLOCK_ACCESS_FS_MAKE_SYM << 1) - 1,  /* v1                */
    (LANDLOCK_ACCESS_FS_REFER    << 1) - 1,  /* v2: add "refer"   */
    (LANDLOCK_ACCESS_FS_TRUNCATE << 1) - 1,  /* v3: add "truncate" */
};

int abi = landlock_create_ruleset(NULL, 0,
                                  LANDLOCK_CREATE_RULESET_VERSION);
if (abi == -1) {
    /*
     * Kernel too old, not compiled with Landlock,
     * or Landlock was not enabled at boot time.
     */
    perror("Unable to use Landlock");
    return;  /* Graceful fallback: Do nothing. */
}
abi = MIN(abi, 3);

/* Only use the available rights in the ruleset. */
attr.handled_access_fs &= landlock_fs_access_rights[abi - 1];
```

The available access rights for each ABI version are listed in the **VERSIONS** section.

If our program needed to create hard links or rename files between different directories (**LANDLOCK_ACCESS_FS_REFER**), we would require the following change to the backwards compatibility logic: Directory reparenting is not possible in a process restricted with Landlock ABI version 1. Therefore, if the program needed to do file reparenting, and if only Landlock ABI version 1 was available, we could not restrict the process.

Now that the ruleset attributes are determined, we create the Landlock ruleset and acquire a file descriptor as a handle to it, using *landlock_create_ruleset*(2):

```
ruleset_fd = landlock_create_ruleset(&attr, sizeof(attr), 0);
if (ruleset_fd == -1) {
    perror("Failed to create a ruleset");
    exit(EXIT_FAILURE);
}
```

We can now add a new rule to the ruleset through the ruleset's file descriptor. The requested access rights must be a subset of the access rights which were specified in *attr.handled_access_fs* at ruleset creation time.

In this example, the rule will only allow reading the file hierarchy */usr*. Without another rule, write actions would then be denied by the ruleset. To add */usr* to the ruleset, we open it with the *O_PATH* flag and fill the *struct landlock_path_beneath_attr* with this file descriptor.

```
struct landlock_path_beneath_attr path_beneath = {0};
int err;

path_beneath.allowed_access =
        LANDLOCK_ACCESS_FS_EXECUTE |
```

```
            LANDLOCK_ACCESS_FS_READ_FILE |
            LANDLOCK_ACCESS_FS_READ_DIR;

    path_beneath.parent_fd = open("/usr", O_PATH | O_CLOEXEC);
    if (path_beneath.parent_fd == -1) {
        perror("Failed to open file");
        close(ruleset_fd);
        exit(EXIT_FAILURE);
    }
    err = landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,
                            &path_beneath, 0);
    close(path_beneath.parent_fd);
    if (err) {
        perror("Failed to update ruleset");
        close(ruleset_fd);
        exit(EXIT_FAILURE);
    }
```

We now have a ruleset with one rule allowing read access to */usr* while denying all other handled accesses for the filesystem. The next step is to restrict the current thread from gaining more privileges (e.g., thanks to a set-user-ID binary).

```
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("Failed to restrict privileges");
        close(ruleset_fd);
        exit(EXIT_FAILURE);
    }
```

The current thread is now ready to sandbox itself with the ruleset.

```
    if (landlock_restrict_self(ruleset_fd, 0)) {
        perror("Failed to enforce ruleset");
        close(ruleset_fd);
        exit(EXIT_FAILURE);
    }
    close(ruleset_fd);
```

If the *landlock_restrict_self*(2) system call succeeds, the current thread is now restricted and this policy will be enforced on all its subsequently created children as well. Once a thread is landlocked, there is no way to remove its security policy; only adding more restrictions is allowed. These threads are now in a new Landlock domain, merge of their parent one (if any) with the new ruleset.

Full working code can be found in

SEE ALSO

landlock_create_ruleset(2), *landlock_add_rule*(2), *landlock_restrict_self*(2)

NAME

libc − overview of standard C libraries on Linux

DESCRIPTION

The term "libc" is commonly used as a shorthand for the "standard C library" a library of standard functions that can be used by all C programs (and sometimes by programs in other languages). Because of some history (see below), use of the term "libc" to refer to the standard C library is somewhat ambiguous on Linux.

glibc

By far the most widely used C library on Linux is the GNU C Library, often referred to as *glibc*. This is the C library that is nowadays used in all major Linux distributions. It is also the C library whose details are documented in the relevant pages of the *man-pages* project (primarily in Section 3 of the manual). Documentation of glibc is also available in the glibc manual, available via the command *info libc*. Release 1.0 of glibc was made in September 1992. (There were earlier 0.x releases.) The next major release of glibc was 2.0, at the beginning of 1997.

The pathname */lib/libc.so.6* (or something similar) is normally a symbolic link that points to the location of the glibc library, and executing this pathname will cause glibc to display various information about the version installed on your system.

Linux libc

In the early to mid 1990s, there was for a while *Linux libc*, a fork of glibc 1.x created by Linux developers who felt that glibc development at the time was not sufficing for the needs of Linux. Often, this library was referred to (ambiguously) as just "libc". Linux libc released major versions 2, 3, 4, and 5, as well as many minor versions of those releases. Linux libc4 was the last version to use the a.out binary format, and the first version to provide (primitive) shared library support. Linux libc 5 was the first version to support the ELF binary format; this version used the shared library soname *libc.so.5*. For a while, Linux libc was the standard C library in many Linux distributions.

However, notwithstanding the original motivations of the Linux libc effort, by the time glibc 2.0 was released (in 1997), it was clearly superior to Linux libc, and all major Linux distributions that had been using Linux libc soon switched back to glibc. To avoid any confusion with Linux libc versions, glibc 2.0 and later used the shared library soname *libc.so.6*.

Since the switch from Linux libc to glibc 2.0 occurred long ago, *man-pages* no longer takes care to document Linux libc details. Nevertheless, the history is visible in vestiges of information about Linux libc that remain in a few manual pages, in particular, references to *libc4* and *libc5*.

Other C libraries

There are various other less widely used C libraries for Linux. These libraries are generally smaller than glibc, both in terms of features and memory footprint, and often intended for building small binaries, perhaps targeted at development for embedded Linux systems. Among such libraries are *uClibc*, *dietlibc*, and *musl libc*. Details of these libraries are covered by the *man-pages* project, where they are known.

SEE ALSO

syscalls(2), *getauxval*(3), *proc*(5), *feature_test_macros*(7), *man-pages*(7), *standards*(7), *vdso*(7)

NAME

locale − description of multilanguage support

SYNOPSIS

#include <locale.h>

DESCRIPTION

A locale is a set of language and cultural rules. These cover aspects such as language for messages, different character sets, lexicographic conventions, and so on. A program needs to be able to determine its locale and act accordingly to be portable to different cultures.

The header <*locale.h*> declares data types, functions, and macros which are useful in this task.

The functions it declares are *setlocale*(3) to set the current locale, and *localeconv*(3) to get information about number formatting.

There are different categories for locale information a program might need; they are declared as macros. Using them as the first argument to the *setlocale*(3) function, it is possible to set one of these to the desired locale:

LC_ADDRESS (GNU extension, since glibc 2.2)
: Change settings that describe the formats (e.g., postal addresses) used to describe locations and geography-related items. Applications that need this information can use *nl_langinfo*(3) to retrieve nonstandard elements, such as **_NL_ADDRESS_COUNTRY_NAME** (country name, in the language of the locale) and **_NL_ADDRESS_LANG_NAME** (language name, in the language of the locale), which return strings such as "Deutschland" and "Deutsch" (for German-language locales). (Other element names are listed in <*langinfo.h*>.)

LC_COLLATE
: This category governs the collation rules used for sorting and regular expressions, including character equivalence classes and multicharacter collating elements. This locale category changes the behavior of the functions *strcoll*(3) and *strxfrm*(3), which are used to compare strings in the local alphabet. For example, the German sharp s is sorted as "ss".

LC_CTYPE
: This category determines the interpretation of byte sequences as characters (e.g., single versus multibyte characters), character classifications (e.g., alphabetic or digit), and the behavior of character classes. On glibc systems, this category also determines the character transliteration rules for *iconv*(1) and *iconv*(3). It changes the behavior of the character handling and classification functions, such as *isupper*(3) and *toupper*(3), and the multibyte character functions such as *mblen*(3) or *wctomb*(3).

LC_IDENTIFICATION (GNU extension, since glibc 2.2)
: Change settings that relate to the metadata for the locale. Applications that need this information can use *nl_langinfo*(3) to retrieve nonstandard elements, such as **_NL_IDENTIFICATION_TITLE** (title of this locale document) and **_NL_IDENTIFICATION_TERRITORY** (geographical territory to which this locale document applies), which might return strings such as "English locale for the USA" and "USA". (Other element names are listed in <*langinfo.h*>.)

LC_MONETARY
: This category determines the formatting used for monetary-related numeric values. This changes the information returned by *localeconv*(3), which describes the way numbers are usually printed, with details such as decimal point versus decimal comma. This information is internally used by the function *strfmon*(3).

LC_MESSAGES
: This category affects the language in which messages are displayed and what an affirmative or negative answer looks like. The GNU C library contains the *gettext*(3), *ngettext*(3), and *rpmatch*(3) functions to ease the use of this information. The GNU gettext family of functions also obey the environment variable **LANGUAGE** (containing a colon-separated list of locales) if the category is set to a valid locale other than **"C"**. This category also affects the behavior of *catopen*(3).

LC_MEASUREMENT (GNU extension, since glibc 2.2)
: Change the settings relating to the measurement system in the locale (i.e., metric versus US customary units). Applications can use *nl_langinfo*(3) to retrieve the nonstandard **_NL_MEASUREMENT_MEASUREMENT** element, which returns a pointer to a character that has the value 1 (metric) or 2 (US customary units).

LC_NAME (GNU extension, since glibc 2.2)
: Change settings that describe the formats used to address persons. Applications that need this information can use *nl_langinfo*(3) to retrieve nonstandard elements, such as **_NL_NAME_NAME_MR** (general salutation for men) and **_NL_NAME_NAME_MS** (general salutation for women) elements, which return strings such as "Herr" and "Frau" (for German-language locales). (Other element names are listed in <*langinfo.h*>.)

LC_NUMERIC
: This category determines the formatting rules used for nonmonetary numeric values—for example, the thousands separator and the radix character (a period in most English-speaking countries, but a comma in many other regions). It affects functions such as *printf*(3), *scanf*(3), and *strtod*(3). This information can also be read with the *localeconv*(3) function.

LC_PAPER (GNU extension, since glibc 2.2)
: Change the settings relating to the dimensions of the standard paper size (e.g., US letter versus A4). Applications that need the dimensions can obtain them by using *nl_langinfo*(3) to retrieve the nonstandard **_NL_PAPER_WIDTH** and **_NL_PAPER_HEIGHT** elements, which return *int* values specifying the dimensions in millimeters.

LC_TELEPHONE (GNU extension, since glibc 2.2)
: Change settings that describe the formats to be used with telephone services. Applications that need this information can use *nl_langinfo*(3) to retrieve nonstandard elements, such as **_NL_TELEPHONE_INT_PREFIX** (international prefix used to call numbers in this locale), which returns a string such as "49" (for Germany). (Other element names are listed in <*langinfo.h*>.)

LC_TIME
: This category governs the formatting used for date and time values. For example, most of Europe uses a 24-hour clock versus the 12-hour clock used in the United States. The setting of this category affects the behavior of functions such as *strftime*(3) and *strptime*(3).

LC_ALL
: All of the above.

If the second argument to *setlocale*(3) is an empty string, "", for the default locale, it is determined using the following steps:

(1) If there is a non-null environment variable **LC_ALL**, the value of **LC_ALL** is used.

(2) If an environment variable with the same name as one of the categories above exists and is non-null, its value is used for that category.

(3) If there is a non-null environment variable **LANG**, the value of **LANG** is used.

Values about local numeric formatting is made available in a *struct lconv* returned by the *localeconv*(3) function, which has the following declaration:

```
struct lconv {

    /* Numeric (nonmonetary) information */

    char *decimal_point;     /* Radix character */
    char *thousands_sep;     /* Separator for digit groups to left
                                of radix character */
    char *grouping;     /* Each element is the number of digits in
                           a group; elements with higher indices
                           are further left.  An element with value
                           CHAR_MAX means that no further grouping
                           is done.  An element with value 0 means
                           that the previous element is used for
```

```
                                all groups further left. */

            /* Remaining fields are for monetary information */

            char *int_curr_symbol;   /* First three chars are a currency
                                        symbol from ISO 4217.  Fourth char
                                        is the separator.  Fifth char
                                        is '\0'. */
            char *currency_symbol;   /* Local currency symbol */
            char *mon_decimal_point; /* Radix character */
            char *mon_thousands_sep; /* Like thousands_sep above */
            char *mon_grouping;      /* Like grouping above */
            char *positive_sign;     /* Sign for positive values */
            char *negative_sign;     /* Sign for negative values */
            char  int_frac_digits;   /* International fractional digits */
            char  frac_digits;       /* Local fractional digits */
            char  p_cs_precedes;     /* 1 if currency_symbol precedes a
                                        positive value, 0 if succeeds */
            char  p_sep_by_space;    /* 1 if a space separates
                                        currency_symbol from a positive
                                        value */
            char  n_cs_precedes;     /* 1 if currency_symbol precedes a
                                        negative value, 0 if succeeds */
            char  n_sep_by_space;    /* 1 if a space separates
                                        currency_symbol from a negative
                                        value */
            /* Positive and negative sign positions:
               0 Parentheses surround the quantity and currency_symbol.
               1 The sign string precedes the quantity and currency_symbol.
               2 The sign string succeeds the quantity and currency_symbol.
               3 The sign string immediately precedes the currency_symbol.
               4 The sign string immediately succeeds the currency_symbol. */
            char  p_sign_posn;
            char  n_sign_posn;
        };
```

POSIX.1-2008 extensions to the locale API

POSIX.1-2008 standardized a number of extensions to the locale API, based on implementations that first appeared in glibc 2.3. These extensions are designed to address the problem that the traditional locale APIs do not mix well with multithreaded applications and with applications that must deal with multiple locales.

The extensions take the form of new functions for creating and manipulating locale objects (**newlocale**(3), *freelocale*(3), *duplocale*(3), and *uselocale*(3)) and various new library functions with the suffix "_l" (e.g., *toupper_l*(3)) that extend the traditional locale-dependent APIs (e.g., *toupper*(3)) to allow the specification of a locale object that should apply when executing the function.

ENVIRONMENT

The following environment variable is used by *newlocale*(3) and *setlocale*(3), and thus affects all unprivileged localized programs:

LOCPATH

A list of pathnames, separated by colons (':'), that should be used to find locale data. If this variable is set, only the individual compiled locale data files from **LOCPATH** and the system default locale data path are used; any available locale archives are not used (see *localedef*(1)). The individual compiled locale data files are searched for under subdirectories which depend on the currently used locale. For example, when *en_GB.UTF−8* is used for a category, the following subdirectories are searched for, in this order: *en_GB.UTF−8*, *en_GB.utf8*, *en_GB*, *en.UTF−8*, *en.utf8*, and *en*.

FILES

/usr/lib/locale/locale−archive
: Usual default locale archive location.

/usr/lib/locale
: Usual default path for compiled individual locale files.

STANDARDS

POSIX.1-2001.

SEE ALSO

iconv(1), *locale*(1), *localedef*(1), *catopen*(3), *gettext*(3), *iconv*(3), *localeconv*(3), *mbstowcs*(3), *newlocale*(3), *ngettext*(3), *nl_langinfo*(3), *rpmatch*(3), *setlocale*(3), *strcoll*(3), *strfmon*(3), *strftime*(3), *strxfrm*(3), *uselocale*(3), *wcstombs*(3), *locale*(5), *charsets*(7), *unicode*(7), *utf-8*(7)

NAME

mailaddr − mail addressing description

DESCRIPTION

This manual page gives a brief introduction to SMTP mail addresses, as used on the Internet. These addresses are in the general format

```
user@domain
```

where a domain is a hierarchical dot-separated list of subdomains. These examples are valid forms of the same address:

```
john.doe@monet.example.com
John Doe <john.doe@monet.example.com>
john.doe@monet.example.com (John Doe)
```

The domain part ("monet.example.com") is a mail-accepting domain. It can be a host and in the past it usually was, but it doesn't have to be. The domain part is not case sensitive.

The local part ("john.doe") is often a username, but its meaning is defined by the local software. Sometimes it is case sensitive, although that is unusual. If you see a local-part that looks like garbage, it is usually because of a gateway between an internal e-mail system and the net, here are some examples:

```
"surname/admd=telemail/c=us/o=hp/prmd=hp"@some.where
USER%SOMETHING@some.where
machine!machine!name@some.where
I2461572@some.where
```

(These are, respectively, an X.400 gateway, a gateway to an arbitrary internal mail system that lacks proper internet support, an UUCP gateway, and the last one is just boring username policy.)

The real-name part ("John Doe") can either be placed before <>, or in () at the end. (Strictly speaking the two aren't the same, but the difference is beyond the scope of this page.) The name may have to be quoted using "", for example, if it contains ".":

```
"John Q. Doe" <john.doe@monet.example.com>
```

Abbreviation

Some mail systems let users abbreviate the domain name. For instance, users at example.com may get away with "john.doe@monet" to send mail to John Doe. *This behavior is deprecated.* Sometimes it works, but you should not depend on it.

Route-addrs

In the past, sometimes one had to route a message through several hosts to get it to its final destination. Addresses which show these relays are termed "route-addrs". These use the syntax:

```
<@hosta,@hostb:user@hostc>
```

This specifies that the message should be sent to hosta, from there to hostb, and finally to hostc. Many hosts disregard route-addrs and send directly to hostc.

Route-addrs are very unusual now. They occur sometimes in old mail archives. It is generally possible to ignore all but the "user@hostc" part of the address to determine the actual address.

Postmaster

Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed. The "postmaster" address is not case sensitive.

FILES

/etc/aliases
~/.forward

SEE ALSO

mail(1), *aliases*(5), *forward*(5), *sendmail*(8)

IETF RFC 5322

NAME

man-pages − conventions for writing Linux man pages

SYNOPSIS

man [*section*] *title*

DESCRIPTION

This page describes the conventions that should be employed when writing man pages for the Linux *man-pages* project, which documents the user-space API provided by the Linux kernel and the GNU C library. The project thus provides most of the pages in Section 2, many of the pages that appear in Sections 3, 4, and 7, and a few of the pages that appear in Sections 1, 5, and 8 of the man pages on a Linux system. The conventions described on this page may also be useful for authors writing man pages for other projects.

Sections of the manual pages

The manual Sections are traditionally defined as follows:

1 User commands (Programs)
Commands that can be executed by the user from within a shell.

2 System calls
Functions which wrap operations performed by the kernel.

3 Library calls
All library functions excluding the system call wrappers (Most of the *libc* functions).

4 Special files (devices)
Files found in */dev* which allow to access to devices through the kernel.

5 File formats and configuration files
Describes various human-readable file formats and configuration files.

6 Games
Games and funny little programs available on the system.

7 Overview, conventions, and miscellaneous
Overviews or descriptions of various topics, conventions, and protocols, character set standards, the standard filesystem layout, and miscellaneous other things.

8 System management commands
Commands like *mount*(8), many of which only root can execute.

Macro package

New manual pages should be marked up using the **groff an.tmac** package described in *man*(7). This choice is mainly for consistency: the vast majority of existing Linux manual pages are marked up using these macros.

Conventions for source file layout

Please limit source code line length to no more than about 75 characters wherever possible. This helps avoid line-wrapping in some mail clients when patches are submitted inline.

Title line

The first command in a man page should be a **TH** command:

.TH *title section date source manual-section*

The arguments of the command are as follows:

title The title of the man page, written in all caps (e.g., *MAN-PAGES*).

section The section number in which the man page should be placed (e.g., *7*).

date The date of the last nontrivial change that was made to the man page. (Within the *man-pages* project, the necessary updates to these timestamps are handled automatically by scripts, so there is no need to manually update them as part of a patch.) Dates should be written in the form YYYY-MM-DD.

source The name and version of the project that provides the manual page (not necessarily the package that provides the functionality).

manual-section
Normally, this should be empty, since the default value will be good.

Sections within a manual page
The list below shows conventional or suggested sections. Most manual pages should include at least the **highlighted** sections. Arrange a new manual page so that sections are placed in the order shown in the list.

NAME	
LIBRARY	[Normally only in Sections 2, 3]
SYNOPSIS	
CONFIGURATION	[Normally only in Section 4]
DESCRIPTION	
OPTIONS	[Normally only in Sections 1, 8]
EXIT STATUS	[Normally only in Sections 1, 8]
RETURN VALUE	[Normally only in Sections 2, 3]
ERRORS	[Typically only in Sections 2, 3]
ENVIRONMENT	
FILES	
ATTRIBUTES	[Normally only in Sections 2, 3]
VERSIONS	[Normally only in Sections 2, 3]
STANDARDS	
HISTORY	
NOTES	
CAVEATS	
BUGS	
EXAMPLES	
AUTHORS	[Discouraged]
REPORTING BUGS	[Not used in man-pages]
COPYRIGHT	[Not used in man-pages]
SEE ALSO	

Where a traditional heading would apply, please use it; this kind of consistency can make the information easier to understand. If you must, you can create your own headings if they make things easier to understand (this can be especially useful for pages in Sections 4 and 5). However, before doing this, consider whether you could use the traditional headings, with some subsections (*.SS*) within those sections.

The following list elaborates on the contents of each of the above sections.

NAME The name of this manual page.

See *man*(7) for important details of the line(s) that should follow the **.SH NAME** command. All words in this line (including the word immediately following the "\−") should be in lowercase, except where English or technical terminological convention dictates otherwise.

LIBRARY
The library providing a symbol.

It shows the common name of the library, and in parentheses, the name of the library file and, if needed, the linker flag needed to link a program against it: (*libfoo*[, −*lfoo*]).

SYNOPSIS
A brief summary of the command or function's interface.

For commands, this shows the syntax of the command and its arguments (including options); boldface is used for as-is text and italics are used to indicate replaceable arguments. Brackets ([]) surround optional arguments, vertical bars (|) separate choices, and ellipses (...) can be repeated. For functions, it shows any required data declarations or **#include** directives, followed by the function declaration.

Where a feature test macro must be defined in order to obtain the declaration of a function (or a variable) from a header file, then the SYNOPSIS should indicate this, as described in *feature_test_macros*(7).

CONFIGURATION
Configuration details for a device.

This section normally appears only in Section 4 pages.

DESCRIPTION
An explanation of what the program, function, or format does.

Discuss how it interacts with files and standard input, and what it produces on standard output or standard error. Omit internals and implementation details unless they're critical for understanding the interface. Describe the usual case; for information on command-line options of a program use the **OPTIONS** section.

When describing new behavior or new flags for a system call or library function, be careful to note the kernel or C library version that introduced the change. The preferred method of noting this information for flags is as part of a **.TP** list, in the following form (here, for a new system call flag):

XYZ_FLAG (since Linux 3.7)
Description of flag...

Including version information is especially useful to users who are constrained to using older kernel or C library versions (which is typical in embedded systems, for example).

OPTIONS
A description of the command-line options accepted by a program and how they change its behavior.

This section should appear only for Section 1 and 8 manual pages.

EXIT STATUS
A list of the possible exit status values of a program and the conditions that cause these values to be returned.

This section should appear only for Section 1 and 8 manual pages.

RETURN VALUE
For Section 2 and 3 pages, this section gives a list of the values the library routine will return to the caller and the conditions that cause these values to be returned.

ERRORS
For Section 2 and 3 manual pages, this is a list of the values that may be placed in *errno* in the event of an error, along with information about the cause of the errors.

Where several different conditions produce the same error, the preferred approach is to create separate list entries (with duplicate error names) for each of the conditions. This makes the separate conditions clear, may make the list easier to read, and allows metainformation (e.g., kernel version number where the condition first became applicable) to be more easily marked for each condition.

The error list should be in alphabetical order.

ENVIRONMENT
A list of all environment variables that affect the program or function and how they affect it.

FILES A list of the files the program or function uses, such as configuration files, startup files, and files the program directly operates on.

Give the full pathname of these files, and use the installation process to modify the directory part to match user preferences. For many programs, the default installation location is in */usr/local*, so your base manual page should use */usr/local* as the base.

ATTRIBUTES
A summary of various attributes of the function(s) documented on this page. See *attributes*(7) for further details.

VERSIONS
A summary of systems where the API performs differently, or where there's a similar API.

STANDARDS

A description of any standards or conventions that relate to the function or command described by the manual page.

The preferred terms to use for the various standards are listed as headings in *standards*(7).

This section should note the current standards to which the API conforms to.

If the API is not governed by any standards but commonly exists on other systems, note them. If the call is Linux-specific or GNU-specific, note this. If it's available in the BSDs, note that.

If this section consists of just a list of standards (which it commonly does), terminate the list with a period ('.').

HISTORY

A brief summary of the Linux kernel or glibc versions where a system call or library function appeared, or changed significantly in its operation.

As a general rule, every new interface should include a HISTORY section in its manual page. Unfortunately, many existing manual pages don't include this information (since there was no policy to do so when they were written). Patches to remedy this are welcome, but, from the perspective of programmers writing new code, this information probably matters only in the case of kernel interfaces that have been added in Linux 2.4 or later (i.e., changes since Linux 2.2), and library functions that have been added to glibc since glibc 2.1 (i.e., changes since glibc 2.0).

The *syscalls*(2) manual page also provides information about kernel versions in which various system calls first appeared.

Old versions of standards should be mentioned here, rather than in STANDARDS, for example, SUS, SUSv2, and XPG, or the SVr4 and 4.xBSD implementation standards.

NOTES

Miscellaneous notes.

For Section 2 and 3 man pages you may find it useful to include subsections (**SS**) named *Linux Notes* and *glibc Notes*.

In Section 2, use the heading *C library/kernel differences* to mark off notes that describe the differences (if any) between the C library wrapper function for a system call and the raw system call interface provided by the kernel.

CAVEATS

Warnings about typical user misuse of an API, that don't constitute an API bug or design defect.

BUGS A list of limitations, known defects or inconveniences, and other questionable activities.

EXAMPLES

One or more examples demonstrating how this function, file, or command is used.

For details on writing example programs, see *Example programs* below.

AUTHORS

A list of authors of the documentation or program.

Use of an AUTHORS section is strongly discouraged. Generally, it is better not to clutter every page with a list of (over time potentially numerous) authors; if you write or significantly amend a page, add a copyright notice as a comment in the source file. If you are the author of a device driver and want to include an address for reporting bugs, place this under the BUGS section.

REPORTING BUGS

The *man-pages* project doesn't use a REPORTING BUGS section in manual pages. Information on reporting bugs is instead supplied in the script-generated COLOPHON section. However, various projects do use a REPORTING BUGS section. It is recommended to place it near the foot of the page.

COPYRIGHT

The *man-pages* project doesn't use a COPYRIGHT section in manual pages. Copyright information is instead maintained in the page source. In pages where this section is present, it is recommended to place it near the foot of the page, just above SEE ALSO.

SEE ALSO

A comma-separated list of related man pages, possibly followed by other related pages or documents.

The list should be ordered by section number and then alphabetically by name. Do not terminate this list with a period.

Where the SEE ALSO list contains many long manual page names, to improve the visual result of the output, it may be useful to employ the *.ad l* (don't right justify) and *.nh* (don't hyphenate) directives. Hyphenation of individual page names can be prevented by preceding words with the string "\%".

Given the distributed, autonomous nature of FOSS projects and their documentation, it is sometimes necessary—and in many cases desirable—that the SEE ALSO section includes references to manual pages provided by other projects.

FORMATTING AND WORDING CONVENTIONS

The following subsections note some details for preferred formatting and wording conventions in various sections of the pages in the *man-pages* project.

SYNOPSIS

Wrap the function prototype(s) in a *.nf*/*.fi* pair to prevent filling.

In general, where more than one function prototype is shown in the SYNOPSIS, the prototypes should *not* be separated by blank lines. However, blank lines (achieved using *.P*) may be added in the following cases:

- to separate long lists of function prototypes into related groups (see for example *list*(3));
- in other cases that may improve readability.

In the SYNOPSIS, a long function prototype may need to be continued over to the next line. The continuation line is indented according to the following rules:

(1) If there is a single such prototype that needs to be continued, then align the continuation line so that when the page is rendered on a fixed-width font device (e.g., on an xterm) the continuation line starts just below the start of the argument list in the line above. (Exception: the indentation may be adjusted if necessary to prevent a very long continuation line or a further continuation line where the function prototype is very long.) As an example:

```
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

(2) But, where multiple functions in the SYNOPSIS require continuation lines, and the function names have different lengths, then align all continuation lines to start in the same column. This provides a nicer rendering in PDF output (because the SYNOPSIS uses a variable width font where spaces render narrower than most characters). As an example:

```
int getopt(int argc, char * const argv[],
           const char *optstring);
int getopt_long(int argc, char * const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

RETURN VALUE

The preferred wording to describe how *errno* is set is "*errno* is set to indicate the error" or similar. This wording is consistent with the wording used in both POSIX.1 and FreeBSD.

ATTRIBUTES

Note the following:

- Wrap the table in this section in a *.ad l*/*.ad* pair to disable text filling and a *.nh*/*.hy* pair to disable hyphenation.

- Ensure that the table occupies the full page width through the use of an *lbx* description for one of the columns (usually the first column, though in some cases the last column if it contains a lot of text).
- Make free use of *T{/T}* macro pairs to allow table cells to be broken over multiple lines (also bearing in mind that pages may sometimes be rendered to a width of less than 80 columns).

For examples of all of the above, see the source code of various pages.

STYLE GUIDE

The following subsections describe the preferred style for the *man-pages* project. For details not covered below, the Chicago Manual of Style is usually a good source; try also grepping for preexisting usage in the project source tree.

Use of gender-neutral language

As far as possible, use gender-neutral language in the text of man pages. Use of "they" ("them", "themself", "their") as a gender-neutral singular pronoun is acceptable.

Formatting conventions for manual pages describing commands

For manual pages that describe a command (typically in Sections 1 and 8), the arguments are always specified using italics, *even in the SYNOPSIS section*.

The name of the command, and its options, should always be formatted in bold.

Formatting conventions for manual pages describing functions

For manual pages that describe functions (typically in Sections 2 and 3), the arguments are always specified using italics, *even in the SYNOPSIS section*, where the rest of the function is specified in bold:

int myfunction(int *argc***, char *****argv***);**

Variable names should, like argument names, be specified in italics.

Any reference to the subject of the current manual page should be written with the name in bold followed by a pair of parentheses in Roman (normal) font. For example, in the *fcntl*(2) man page, references to the subject of the page would be written as: **fcntl**(). The preferred way to write this in the source file is:

```
.BR fcntl ()
```

(Using this format, rather than the use of "\fB...\fP()" makes it easier to write tools that parse man page source files.)

Use semantic newlines

In the source of a manual page, new sentences should be started on new lines, long sentences should be split into lines at clause breaks (commas, semicolons, colons, and so on), and long clauses should be split at phrase boundaries. This convention, sometimes known as "semantic newlines", makes it easier to see the effect of patches, which often operate at the level of individual sentences, clauses, or phrases.

Lists

There are different kinds of lists:

Tagged paragraphs

These are used for a list of tags and their descriptions. When the tags are constants (either macros or numbers) they are in bold. Use the **.TP** macro.

An example is this "Tagged paragraphs" subsection is itself.

Ordered lists

Elements are preceded by a number in parentheses (1), (2). These represent a set of steps that have an order.

When there are substeps, they will be numbered like (4.2).

Positional lists

Elements are preceded by a number (index) in square brackets [4], [5]. These represent fields in a set. The first index will be:

0 When it represents fields of a C data structure, to be consistent with arrays.
1 When it represents fields of a file, to be consistent with tools like *cut*(1)

Alternatives list
: Elements are preceded by a letter in parentheses (a), (b). These represent a set of (normally) exclusive alternatives.

Bullet lists
: Elements are preceded by bullet symbols (**\[bu]**). Anything that doesn't fit elsewhere is usually covered by this type of list.

Numbered notes
: Not really a list, but the syntax is identical to "positional lists".

There should always be exactly 2 spaces between the list symbol and the elements. This doesn't apply to "tagged paragraphs", which use the default indentation rules.

Formatting conventions (general)

Paragraphs should be separated by suitable markers (usually either *.P* or *.IP*). Do *not* separate paragraphs using blank lines, as this results in poor rendering in some output formats (such as PostScript and PDF).

Filenames (whether pathnames, or references to header files) are always in italics (e.g., <*stdio.h*>), except in the SYNOPSIS section, where included files are in bold (e.g., **#include <stdio.h>**). When referring to a standard header file include, specify the header file surrounded by angle brackets, in the usual C way (e.g., <*stdio.h*>).

Special macros, which are usually in uppercase, are in bold (e.g., **MAXINT**). Exception: don't boldface NULL.

When enumerating a list of error codes, the codes are in bold (this list usually uses the **.TP** macro).

Complete commands should, if long, be written as an indented line on their own, with a blank line before and after the command, for example

```
man 7 man-pages
```

If the command is short, then it can be included inline in the text, in italic format, for example, *man 7 man-pages*. In this case, it may be worth using nonbreaking spaces (\[ti]) at suitable places in the command. Command options should be written in italics (e.g., *-l*).

Expressions, if not written on a separate indented line, should be specified in italics. Again, the use of nonbreaking spaces may be appropriate if the expression is inlined with normal text.

When showing example shell sessions, user input should be formatted in bold, for example

```
$ date
Thu Jul  7 13:01:27 CEST 2016
```

Any reference to another man page should be written with the name in bold, *always* followed by the section number, formatted in Roman (normal) font, without any separating spaces (e.g., *intro*(2)). The preferred way to write this in the source file is:

```
.BR intro (2)
```

(Including the section number in cross references lets tools like *man2html*(1) create properly hyperlinked pages.)

Control characters should be written in bold face, with no quotes; for example, **^X**.

Spelling

Starting with release 2.59, *man-pages* follows American spelling conventions (previously, there was a random mix of British and American spellings); please write all new pages and patches according to these conventions.

Aside from the well-known spelling differences, there are a few other subtleties to watch for:

- American English tends to use the forms "backward", "upward", "toward", and so on rather than the British forms "backwards", "upwards", "towards", and so on.
- Opinions are divided on "acknowledgement" vs "acknowledgment". The latter is predominant, but not universal usage in American English. POSIX and the BSD license use the former spelling. In the Linux man-pages project, we use "acknowledgement".

BSD version numbers

The classical scheme for writing BSD version numbers is *x.yBSD*, where *x.y* is the version number (e.g., 4.2BSD). Avoid forms such as *BSD 4.3*.

Capitalization

In subsection ("SS") headings, capitalize the first word in the heading, but otherwise use lowercase, except where English usage (e.g., proper nouns) or programming language requirements (e.g., identifier names) dictate otherwise. For example:

```
.SS Unicode under Linux
```

Indentation of structure definitions, shell session logs, and so on

When structure definitions, shell session logs, and so on are included in running text, indent them by 4 spaces (i.e., a block enclosed by *.in +4n* and *.in*), format them using the *.EX* and *.EE* macros, and surround them with suitable paragraph markers (either *.P* or *.IP*). For example:

```
.P
.in +4n
.EX
int
main(int argc, char *argv[])
{
    return 0;
}
.EE
.in
.P
```

Preferred terms

The following table lists some preferred terms to use in man pages, mainly to ensure consistency across pages.

Term	Avoid using	Notes
built-in	builtin	
Epoch	epoch	For the UNIX Epoch (00:00:00, 1 Jan 1970 UTC)
filename	file name	
filesystem	file system	
hostname	host name	
inode	i-node	
lowercase	lower case, lower-case	
nonzero	non-zero	
pathname	path name	
pseudoterminal	pseudo-terminal	
privileged port	reserved port, system port	
real-time	realtime, real time	
run time	runtime	
saved set-group-ID	saved group ID, saved set-GID	
saved set-user-ID	saved user ID, saved set-UID	
set-group-ID	set-GID, setgid	
set-user-ID	set-UID, setuid	
superuser	super user, super-user	
superblock	super block, super-block	
symbolic link	symlink	
timestamp	time stamp	
timezone	time zone	

uppercase	upper case, upper-case	
usable	useable	
user space	userspace	
username	user name	
x86-64	x86_64	Except if referring to result of "uname −m" or similar
zeros	zeroes	

See also the discussion *Hyphenation of attributive compounds* below.

Terms to avoid

The following table lists some terms to avoid using in man pages, along with some suggested alternatives, mainly to ensure consistency across pages.

Avoid	Use instead	Notes
32bit	32-bit	same for 8-bit, 16-bit, etc.
current process	calling process	A common mistake made by kernel programmers when writing man pages
manpage	man page, manual page	
minus infinity	negative infinity	
non-root	unprivileged user	
non-superuser	unprivileged user	
nonprivileged	unprivileged	
OS	operating system	
plus infinity	positive infinity	
pty	pseudoterminal	
tty	terminal	
Unices	UNIX systems	
Unixes	UNIX systems	

Trademarks

Use the correct spelling and case for trademarks. The following is a list of the correct spellings of various relevant trademarks that are sometimes misspelled:

DG/UX
HP-UX
UNIX
UnixWare

NULL, NUL, null pointer, and null byte

A *null pointer* is a pointer that points to nothing, and is normally indicated by the constant *NULL*. On the other hand, *NUL* is the *null byte*, a byte with the value 0, represented in C via the character constant *'\0'*.

The preferred term for the pointer is "null pointer" or simply "NULL"; avoid writing "NULL pointer".

The preferred term for the byte is "null byte". Avoid writing "NUL", since it is too easily confused with "NULL". Avoid also the terms "zero byte" and "null character". The byte that terminates a C string should be described as "the terminating null byte"; strings may be described as "null-terminated", but avoid the use of "NUL-terminated".

Hyperlinks

For hyperlinks, use the *.UR*/*.UE* macro pair (see *groff_man*(7)). This produces proper hyperlinks that can be used in a web browser, when rendering a page with, say:

```
BROWSER=firefox man -H pagename
```

Use of e.g., i.e., etc., a.k.a., and similar

In general, the use of abbreviations such as "e.g.", "i.e.", "etc.", "cf.", and "a.k.a." should be avoided, in favor of suitable full wordings ("for example", "that is", "and so on", "compare to", "also known as").

The only place where such abbreviations may be acceptable is in *short* parenthetical asides (e.g., like this one).

Always include periods in such abbreviations, as shown here. In addition, "e.g." and "i.e." should

always be followed by a comma.

Em-dashes

The way to write an em-dash—the glyph that appears at either end of this subphrase—in *roff is with the macro "\[em]". (On an ASCII terminal, an em-dash typically renders as two hyphens, but in other typographical contexts it renders as a long dash.) Em-dashes should be written *without* surrounding spaces.

Hyphenation of attributive compounds

Compound terms should be hyphenated when used attributively (i.e., to qualify a following noun). Some examples:

32-bit value
command-line argument
floating-point number
run-time check
user-space function
wide-character string

Hyphenation with multi, non, pre, re, sub, and so on

The general tendency in modern English is not to hyphenate after prefixes such as "multi", "non", "pre", "re", "sub", and so on. Manual pages should generally follow this rule when these prefixes are used in natural English constructions with simple suffixes. The following list gives some examples of the preferred forms:

interprocess
multithreaded
multiprocess
nonblocking
nondefault
nonempty
noninteractive
nonnegative
nonportable
nonzero
preallocated
precreate
prerecorded
reestablished
reinitialize
rearm
reread
subcomponent
subdirectory
subsystem

Hyphens should be retained when the prefixes are used in nonstandard English words, with trademarks, proper nouns, acronyms, or compound terms. Some examples:

non-ASCII
non-English
non-NULL
non-real-time

Finally, note that "re-create" and "recreate" are two different verbs, and the former is probably what you want.

Generating optimal glyphs

Where a real minus character is required (e.g., for numbers such as −1, for man page cross references such as *utf-8*(7), or when writing options that have a leading dash, such as in *ls −l*), use the following form in the man page source:

```
\-
```

This guideline applies also to code examples.

The use of real minus signs serves the following purposes:

- To provide better renderings on various targets other than ASCII terminals, notably in PDF and on Unicode/UTF−8-capable terminals.
- To generate glyphs that when copied from rendered pages will produce real minus signs when pasted into a terminal.

To produce unslanted single quotes that render well in ASCII, UTF-8, and PDF, use "\[aq]" ("apostrophe quote"); for example

```
\[aq]C\[aq]
```

where *C* is the quoted character. This guideline applies also to character constants used in code examples.

Where a proper caret (^) that renders well in both a terminal and PDF is required, use "\[ha]". This is especially necessary in code samples, to get a nicely rendered caret when rendering to PDF.

Using a naked "~" character results in a poor rendering in PDF. Instead use "\[ti]". This is especially necessary in code samples, to get a nicely rendered tilde when rendering to PDF.

Example programs and shell sessions

Manual pages may include example programs demonstrating how to use a system call or library function. However, note the following:

- Example programs should be written in C.
- An example program is necessary and useful only if it demonstrates something beyond what can easily be provided in a textual description of the interface. An example program that does nothing other than call an interface usually serves little purpose.
- Example programs should ideally be short (e.g., a good example can often be provided in less than 100 lines of code), though in some cases longer programs may be necessary to properly illustrate the use of an API.
- Expressive code is appreciated.
- Comments should included where helpful. Complete sentences in free-standing comments should be terminated by a period. Periods should generally be omitted in "tag" comments (i.e., comments that are placed on the same line of code); such comments are in any case typically brief phrases rather than complete sentences.
- Example programs should do error checking after system calls and library function calls.
- Example programs should be complete, and compile without warnings when compiled with *cc −Wall*.
- Where possible and appropriate, example programs should allow experimentation, by varying their behavior based on inputs (ideally from command-line arguments, or alternatively, via input read by the program).
- Example programs should be laid out according to Kernighan and Ritchie style, with 4-space indents. (Avoid the use of TAB characters in source code!) The following command can be used to format your source code to something close to the preferred style:

  ```
  indent −npro −kr −i4 −ts4 −sob −l72 −ss −nut −psl prog.c
  ```

- For consistency, all example programs should terminate using either of:

  ```
  exit(EXIT_SUCCESS);
  exit(EXIT_FAILURE);
  ```

  Avoid using the following forms to terminate a program:

  ```
  exit(0);
  exit(1);
  return n;
  ```

- If there is extensive explanatory text before the program source code, mark off the source code with a subsection heading *Program source*, as in:

```
.SS Program source
```

Always do this if the explanatory text includes a shell session log.

If you include a shell session log demonstrating the use of a program or other system feature:

- Place the session log above the source code listing.
- Indent the session log by four spaces.
- Boldface the user input text, to distinguish it from output produced by the system.

For some examples of what example programs should look like, see *wait*(2) and *pipe*(2).

EXAMPLES

For canonical examples of how man pages in the *man-pages* package should look, see *pipe*(2) and *fcntl*(2).

SEE ALSO

man(1), *man2html*(1), *attributes*(7), *groff*(7), *groff_man*(7), *man*(7), *mdoc*(7)

NAME

math_error − detecting errors from mathematical functions

SYNOPSIS

#include <math.h>
#include <errno.h>
#include <fenv.h>

DESCRIPTION

When an error occurs, most library functions indicate this fact by returning a special value (e.g., −1 or NULL). Because they typically return a floating-point number, the mathematical functions declared in *<math.h>* indicate an error using other mechanisms. There are two error-reporting mechanisms: the older one sets *errno*; the newer one uses the floating-point exception mechanism (the use of *feclearexcept*(3) and *fetestexcept*(3), as outlined below) described in *fenv*(3).

A portable program that needs to check for an error from a mathematical function should set *errno* to zero, and make the following call

```
feclearexcept(FE_ALL_EXCEPT);
```

before calling a mathematical function.

Upon return from the mathematical function, if *errno* is nonzero, or the following call (see *fenv*(3)) returns nonzero

```
fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
             FE_UNDERFLOW);
```

then an error occurred in the mathematical function.

The error conditions that can occur for mathematical functions are described below.

Domain error

A *domain error* occurs when a mathematical function is supplied with an argument whose value falls outside the domain for which the function is defined (e.g., giving a negative argument to *log*(3)). When a domain error occurs, math functions commonly return a NaN (though some functions return a different value in this case); *errno* is set to **EDOM**, and an "invalid" (**FE_INVALID**) floating-point exception is raised.

Pole error

A *pole error* occurs when the mathematical result of a function is an exact infinity (e.g., the logarithm of 0 is negative infinity). When a pole error occurs, the function returns the (signed) value **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, depending on whether the function result type is *double*, *float*, or *long double*. The sign of the result is that which is mathematically correct for the function. *errno* is set to **ERANGE**, and a "divide-by-zero" (**FE_DIVBYZERO**) floating-point exception is raised.

Range error

A *range error* occurs when the magnitude of the function result means that it cannot be represented in the result type of the function. The return value of the function depends on whether the range error was an overflow or an underflow.

A floating result *overflows* if the result is finite, but is too large to represented in the result type. When an overflow occurs, the function returns the value **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, depending on whether the function result type is *double*, *float*, or *long double*. *errno* is set to **ERANGE**, and an "overflow" (**FE_OVERFLOW**) floating-point exception is raised.

A floating result *underflows* if the result is too small to be represented in the result type. If an underflow occurs, a mathematical function typically returns 0.0 (C99 says a function shall return "an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type"). *errno* may be set to **ERANGE**, and an "underflow" (**FE_UNDERFLOW**) floating-point exception may be raised.

Some functions deliver a range error if the supplied argument value, or the correct function result, would be *subnormal*. A subnormal value is one that is nonzero, but with a magnitude that is so small that it can't be presented in normalized form (i.e., with a 1 in the most significant bit of the significand). The representation of a subnormal number will contain one or more leading zeros in the significand.

NOTES

The *math_errhandling* identifier specified by C99 and POSIX.1 is not supported by glibc. This identifier is supposed to indicate which of the two error-notification mechanisms (*errno*, exceptions retrievable via *fetestexcept*(3)) is in use. The standards require that at least one be in use, but permit both to be available. The current (glibc 2.8) situation under glibc is messy. Most (but not all) functions raise exceptions on errors. Some also set *errno*. A few functions set *errno*, but don't raise an exception. A very few functions do neither. See the individual manual pages for details.

To avoid the complexities of using *errno* and *fetestexcept*(3) for error checking, it is often advised that one should instead check for bad argument values before each call. For example, the following code ensures that *log*(3)'s argument is not a NaN and is not zero (a pole error) or less than zero (a domain error):

```
double x, r;

if (isnan(x) || islessequal(x, 0)) {
    /* Deal with NaN / pole error / domain error */
}

r = log(x);
```

The discussion on this page does not apply to the complex mathematical functions (i.e., those declared by <*complex.h*>), which in general are not required to return errors by C99 and POSIX.1.

The *gcc*(1) *−fno−math−errno* option causes the executable to employ implementations of some mathematical functions that are faster than the standard implementations, but do not set *errno* on error. (The *gcc*(1) *−ffast−math* option also enables *−fno−math−errno*.) An error can still be tested for using *fetestexcept*(3).

SEE ALSO

gcc(1), *errno*(3), *fenv*(3), *fpclassify*(3), *INFINITY*(3), *isgreater*(3), *matherr*(3), *nan*(3)

info libc

NAME

mount_namespaces − overview of Linux mount namespaces

DESCRIPTION

For an overview of namespaces, see *namespaces*(7).

Mount namespaces provide isolation of the list of mounts seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies.

The views provided by the */proc/*pid*/mounts*, */proc/*pid*/mountinfo*, and */proc/*pid*/mountstats* files (all described in *proc*(5)) correspond to the mount namespace in which the process with the PID *pid* resides. (All of the processes that reside in the same mount namespace will see the same view in these files.)

A new mount namespace is created using either *clone*(2) or *unshare*(2) with the **CLONE_NEWNS** flag. When a new mount namespace is created, its mount list is initialized as follows:

- If the namespace is created using *clone*(2), the mount list of the child's namespace is a copy of the mount list in the parent process's mount namespace.
- If the namespace is created using *unshare*(2), the mount list of the new namespace is a copy of the mount list in the caller's previous mount namespace.

Subsequent modifications to the mount list (**mount**(2) and *umount*(2)) in either mount namespace will not (by default) affect the mount list seen in the other namespace (but see the following discussion of shared subtrees).

SHARED SUBTREES

After the implementation of mount namespaces was completed, experience showed that the isolation that they provided was, in some cases, too great. For example, in order to make a newly loaded optical disk available in all mount namespaces, a mount operation was required in each namespace. For this use case, and others, the shared subtree feature was introduced in Linux 2.6.15. This feature allows for automatic, controlled propagation of *mount*(2) and *umount*(2) *events* between namespaces (or, more precisely, between the mounts that are members of a *peer group* that are propagating events to one another).

Each mount is marked (via *mount*(2)) as having one of the following *propagation types*:

MS_SHARED
: This mount shares events with members of a peer group. *mount*(2) and *umount*(2) events immediately under this mount will propagate to the other mounts that are members of the peer group. *Propagation* here means that the same *mount*(2) or *umount*(2) will automatically occur under all of the other mounts in the peer group. Conversely, *mount*(2) and *umount*(2) events that take place under peer mounts will propagate to this mount.

MS_PRIVATE
: This mount is private; it does not have a peer group. *mount*(2) and *umount*(2) events do not propagate into or out of this mount.

MS_SLAVE
: *mount*(2) and *umount*(2) events propagate into this mount from a (master) shared peer group. *mount*(2) and *umount*(2) events under this mount do not propagate to any peer.

  Note that a mount can be the slave of another peer group while at the same time sharing *mount*(2) and *umount*(2) events with a peer group of which it is a member. (More precisely, one peer group can be the slave of another peer group.)

MS_UNBINDABLE
: This is like a private mount, and in addition this mount can't be bind mounted. Attempts to bind mount this mount (**mount**(2) with the **MS_BIND** flag) will fail.

  When a recursive bind mount (**mount**(2) with the **MS_BIND** and **MS_REC** flags) is performed on a directory subtree, any bind mounts within the subtree are automatically pruned (i.e., not replicated) when replicating that subtree to produce the target subtree.

For a discussion of the propagation type assigned to a new mount, see NOTES.

The propagation type is a per-mount-point setting; some mounts may be marked as shared (with each

shared mount being a member of a distinct peer group), while others are private (or slaved or unbindable).

Note that a mount's propagation type determines whether *mount*(2) and *umount*(2) of mounts *immediately under* the mount are propagated. Thus, the propagation type does not affect propagation of events for grandchildren and further removed descendant mounts. What happens if the mount itself is unmounted is determined by the propagation type that is in effect for the *parent* of the mount.

Members are added to a *peer group* when a mount is marked as shared and either:

(a) the mount is replicated during the creation of a new mount namespace; or

(b) a new bind mount is created from the mount.

In both of these cases, the new mount joins the peer group of which the existing mount is a member.

A new peer group is also created when a child mount is created under an existing mount that is marked as shared. In this case, the new child mount is also marked as shared and the resulting peer group consists of all the mounts that are replicated under the peers of parent mounts.

A mount ceases to be a member of a peer group when either the mount is explicitly unmounted, or when the mount is implicitly unmounted because a mount namespace is removed (because it has no more member processes).

The propagation type of the mounts in a mount namespace can be discovered via the "optional fields" exposed in */proc/*pid*/mountinfo*. (See *proc*(5) for details of this file.) The following tags can appear in the optional fields for a record in that file:

shared:X
: This mount is shared in peer group *X*. Each peer group has a unique ID that is automatically generated by the kernel, and all mounts in the same peer group will show the same ID. (These IDs are assigned starting from the value 1, and may be recycled when a peer group ceases to have any members.)

master:X
: This mount is a slave to shared peer group *X*.

propagate_from:X (since Linux 2.6.26)
: This mount is a slave and receives propagation from shared peer group *X*. This tag will always appear in conjunction with a *master:X* tag. Here, *X* is the closest dominant peer group under the process's root directory. If *X* is the immediate master of the mount, or if there is no dominant peer group under the same root, then only the *master:X* field is present and not the *propagate_from:X* field. For further details, see below.

unbindable
: This is an unbindable mount.

If none of the above tags is present, then this is a private mount.

MS_SHARED and MS_PRIVATE example

Suppose that on a terminal in the initial mount namespace, we mark one mount as shared and another as private, and then view the mounts in */proc/self/mountinfo*:

```
sh1# mount --make-shared /mntS
sh1# mount --make-private /mntP
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
77 61 8:17 / /mntS rw,relatime shared:1
83 61 8:15 / /mntP rw,relatime
```

From the */proc/self/mountinfo* output, we see that */mntS* is a shared mount in peer group 1, and that */mntP* has no optional tags, indicating that it is a private mount. The first two fields in each record in this file are the unique ID for this mount, and the mount ID of the parent mount. We can further inspect this file to see that the parent mount of */mntS* and */mntP* is the root directory, */*, which is mounted as private:

```
sh1# cat /proc/self/mountinfo | awk '$1 == 61' | sed 's/ - .*//'
61 0 8:2 / / rw,relatime
```

On a second terminal, we create a new mount namespace where we run a second shell and inspect the

mounts:

```
$ PS1='sh2# ' sudo unshare -m --propagation unchanged sh
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
222 145 8:17 / /mntS rw,relatime shared:1
225 145 8:15 / /mntP rw,relatime
```

The new mount namespace received a copy of the initial mount namespace's mounts. These new mounts maintain the same propagation types, but have unique mount IDs. (The *--propagation unchanged* option prevents *unshare*(1) from marking all mounts as private when creating a new mount namespace, which it does by default.)

In the second terminal, we then create submounts under each of */mntS* and */mntP* and inspect the set-up:

```
sh2# mkdir /mntS/a
sh2# mount /dev/sdb6 /mntS/a
sh2# mkdir /mntP/b
sh2# mount /dev/sdb7 /mntP/b
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
222 145 8:17 / /mntS rw,relatime shared:1
225 145 8:15 / /mntP rw,relatime
178 222 8:22 / /mntS/a rw,relatime shared:2
230 225 8:23 / /mntP/b rw,relatime
```

From the above, it can be seen that */mntS/a* was created as shared (inheriting this setting from its parent mount) and */mntP/b* was created as a private mount.

Returning to the first terminal and inspecting the set-up, we see that the new mount created under the shared mount */mntS* propagated to its peer mount (in the initial mount namespace), but the new mount created under the private mount */mntP* did not propagate:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
77 61 8:17 / /mntS rw,relatime shared:1
83 61 8:15 / /mntP rw,relatime
179 77 8:22 / /mntS/a rw,relatime shared:2
```

MS_SLAVE example

Making a mount a slave allows it to receive propagated *mount*(2) and *umount*(2) events from a master shared peer group, while preventing it from propagating events to that master. This is useful if we want to (say) receive a mount event when an optical disk is mounted in the master shared peer group (in another mount namespace), but want to prevent *mount*(2) and *umount*(2) events under the slave mount from having side effects in other namespaces.

We can demonstrate the effect of slaving by first marking two mounts as shared in the initial mount namespace:

```
sh1# mount --make-shared /mntX
sh1# mount --make-shared /mntY
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
```

On a second terminal, we create a new mount namespace and inspect the mounts:

```
sh2# unshare -m --propagation unchanged sh
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime shared:2
```

In the new mount namespace, we then mark one of the mounts as a slave:

```
sh2# mount --make-slave /mntY
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
```

From the above output, we see that */mntY* is now a slave mount that is receiving propagation events

from the shared peer group with the ID 2.

Continuing in the new namespace, we create submounts under each of */mntX* and */mntY*:

```
sh2# mkdir /mntX/a
sh2# mount /dev/sda3 /mntX/a
sh2# mkdir /mntY/b
sh2# mount /dev/sda5 /mntY/b
```

When we inspect the state of the mounts in the new mount namespace, we see that */mntX/a* was created as a new shared mount (inheriting the "shared" setting from its parent mount) and */mntY/b* was created as a private mount:

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
173 168 8:3 / /mntX/a rw,relatime shared:3
175 169 8:5 / /mntY/b rw,relatime
```

Returning to the first terminal (in the initial mount namespace), we see that the mount */mntX/a* propagated to the peer (the shared */mntX*), but the mount */mntY/b* was not propagated:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
174 132 8:3 / /mntX/a rw,relatime shared:3
```

Now we create a new mount under */mntY* in the first shell:

```
sh1# mkdir /mntY/c
sh1# mount /dev/sda1 /mntY/c
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
174 132 8:3 / /mntX/a rw,relatime shared:3
178 133 8:1 / /mntY/c rw,relatime shared:4
```

When we examine the mounts in the second mount namespace, we see that in this case the new mount has been propagated to the slave mount, and that the new mount is itself a slave mount (to peer group 4):

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
173 168 8:3 / /mntX/a rw,relatime shared:3
175 169 8:5 / /mntY/b rw,relatime
179 169 8:1 / /mntY/c rw,relatime master:4
```

MS_UNBINDABLE example

One of the primary purposes of unbindable mounts is to avoid the "mount explosion" problem when repeatedly performing bind mounts of a higher-level subtree at a lower-level mount. The problem is illustrated by the following shell session.

Suppose we have a system with the following mounts:

```
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
```

Suppose furthermore that we wish to recursively bind mount the root directory under several users' home directories. We do this for the first user, and inspect the mounts:

```
# mount --rbind / /home/cecilia/
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
```

```
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
```

When we repeat this operation for the second user, we start to see the explosion problem:

```
# mount --rbind / /home/henry
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sdb7 on /home/henry/mntY
/dev/sda1 on /home/henry/home/cecilia
/dev/sdb6 on /home/henry/home/cecilia/mntX
/dev/sdb7 on /home/henry/home/cecilia/mntY
```

Under */home/henry*, we have not only recursively added the */mntX* and */mntY* mounts, but also the recursive mounts of those directories under */home/cecilia* that were created in the previous step. Upon repeating the step for a third user, it becomes obvious that the explosion is exponential in nature:

```
# mount --rbind / /home/otto
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sdb7 on /home/henry/mntY
/dev/sda1 on /home/henry/home/cecilia
/dev/sdb6 on /home/henry/home/cecilia/mntX
/dev/sdb7 on /home/henry/home/cecilia/mntY
/dev/sda1 on /home/otto
/dev/sdb6 on /home/otto/mntX
/dev/sdb7 on /home/otto/mntY
/dev/sda1 on /home/otto/home/cecilia
/dev/sdb6 on /home/otto/home/cecilia/mntX
/dev/sdb7 on /home/otto/home/cecilia/mntY
/dev/sda1 on /home/otto/home/henry
/dev/sdb6 on /home/otto/home/henry/mntX
/dev/sdb7 on /home/otto/home/henry/mntY
/dev/sda1 on /home/otto/home/henry/home/cecilia
/dev/sdb6 on /home/otto/home/henry/home/cecilia/mntX
/dev/sdb7 on /home/otto/home/henry/home/cecilia/mntY
```

The mount explosion problem in the above scenario can be avoided by making each of the new mounts unbindable. The effect of doing this is that recursive mounts of the root directory will not replicate the unbindable mounts. We make such a mount for the first user:

```
# mount --rbind --make-unbindable / /home/cecilia
```

Before going further, we show that unbindable mounts are indeed unbindable:

```
# mkdir /mntZ
# mount --bind /home/cecilia /mntZ
mount: wrong fs type, bad option, bad superblock on /home/cecilia,
```

```
       missing codepage or helper program, or other error

       In some cases useful info is found in syslog - try
       dmesg | tail or so.
```

Now we create unbindable recursive bind mounts for the other two users:

```
# mount --rbind --make-unbindable / /home/henry
# mount --rbind --make-unbindable / /home/otto
```

Upon examining the list of mounts, we see there has been no explosion of mounts, because the unbindable mounts were not replicated under each user's directory:

```
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sdb7 on /home/henry/mntY
/dev/sda1 on /home/otto
/dev/sdb6 on /home/otto/mntX
/dev/sdb7 on /home/otto/mntY
```

Propagation type transitions

The following table shows the effect that applying a new propagation type (i.e., *mount --make-xxxx*) has on the existing propagation type of a mount. The rows correspond to existing propagation types, and the columns are the new propagation settings. For reasons of space, "private" is abbreviated as "priv" and "unbindable" as "unbind".

	make-shared	**make-slave**	**make-priv**	**make-unbind**
shared	shared	slave/priv [1]	priv	unbind
slave	slave+shared	slave [2]	priv	unbind
slave+shared	slave+shared	slave	priv	unbind
private	shared	priv [2]	priv	unbind
unbindable	shared	unbind [2]	priv	unbind

Note the following details to the table:

[1] If a shared mount is the only mount in its peer group, making it a slave automatically makes it private.

[2] Slaving a nonshared mount has no effect on the mount.

Bind (MS_BIND) semantics

Suppose that the following command is performed:

```
mount --bind A/a B/b
```

Here, *A* is the source mount, *B* is the destination mount, *a* is a subdirectory path under the mount point *A*, and *b* is a subdirectory path under the mount point *B*. The propagation type of the resulting mount, *B/b*, depends on the propagation types of the mounts *A* and *B*, and is summarized in the following table.

		source(A)			
		shared	**private**	**slave**	**unbind**
dest(B)	**shared**	shared	shared	slave+shared	invalid
	nonshared	shared	private	slave	invalid

Note that a recursive bind of a subtree follows the same semantics as for a bind operation on each mount in the subtree. (Unbindable mounts are automatically pruned at the target mount point.)

For further details, see *Documentation/filesystems/sharedsubtree.rst* in the kernel source tree.

Move (MS_MOVE) semantics

Suppose that the following command is performed:

```
mount --move A B/b
```

Here, *A* is the source mount, *B* is the destination mount, and *b* is a subdirectory path under the mount point *B*. The propagation type of the resulting mount, *B/b*, depends on the propagation types of the mounts *A* and *B*, and is summarized in the following table.

		source(A)			
		shared	**private**	**slave**	**unbind**
dest(B)	**shared**	shared	shared	slave+shared	invalid
	nonshared	shared	private	slave	unbindable

Note: moving a mount that resides under a shared mount is invalid.

For further details, see *Documentation/filesystems/sharedsubtree.rst* in the kernel source tree.

Mount semantics

Suppose that we use the following command to create a mount:

```
mount device B/b
```

Here, *B* is the destination mount, and *b* is a subdirectory path under the mount point *B*. The propagation type of the resulting mount, *B/b*, follows the same rules as for a bind mount, where the propagation type of the source mount is considered always to be private.

Unmount semantics

Suppose that we use the following command to tear down a mount:

```
umount A
```

Here, *A* is a mount on *B/b*, where *B* is the parent mount and *b* is a subdirectory path under the mount point *B*. If **B** is shared, then all most-recently-mounted mounts at *b* on mounts that receive propagation from mount *B* and do not have submounts under them are unmounted.

The /proc/ pid /mountinfo propagate_from tag

The *propagate_from:X* tag is shown in the optional fields of a */proc/*pid*/mountinfo* record in cases where a process can't see a slave's immediate master (i.e., the pathname of the master is not reachable from the filesystem root directory) and so cannot determine the chain of propagation between the mounts it can see.

In the following example, we first create a two-link master-slave chain between the mounts */mnt*, */tmp/etc*, and */mnt/tmp/etc*. Then the *chroot*(1) command is used to make the */tmp/etc* mount point unreachable from the root directory, creating a situation where the master of */mnt/tmp/etc* is not reachable from the (new) root directory of the process.

First, we bind mount the root directory onto */mnt* and then bind mount */proc* at */mnt/proc* so that after the later *chroot*(1) the *proc*(5) filesystem remains visible at the correct location in the chroot-ed environment.

```
# mkdir -p /mnt/proc
# mount --bind / /mnt
# mount --bind /proc /mnt/proc
```

Next, we ensure that the */mnt* mount is a shared mount in a new peer group (with no peers):

```
# mount --make-private /mnt  # Isolate from any previous peer group
# mount --make-shared /mnt
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
```

Next, we bind mount */mnt/etc* onto */tmp/etc*:

```
# mkdir -p /tmp/etc
# mount --bind /mnt/etc /tmp/etc
# cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
```

```
267 40 8:2 /etc /tmp/etc ... shared:102
```

Initially, these two mounts are in the same peer group, but we then make the */tmp/etc* a slave of */mnt/etc*, and then make */tmp/etc* shared as well, so that it can propagate events to the next slave in the chain:

```
# mount --make-slave /tmp/etc
# mount --make-shared /tmp/etc
# cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
267 40 8:2 /etc /tmp/etc ... shared:105 master:102
```

Then we bind mount */tmp/etc* onto */mnt/tmp/etc*. Again, the two mounts are initially in the same peer group, but we then make */mnt/tmp/etc* a slave of */tmp/etc*:

```
# mkdir -p /mnt/tmp/etc
# mount --bind /tmp/etc /mnt/tmp/etc
# mount --make-slave /mnt/tmp/etc
# cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
267 40 8:2 /etc /tmp/etc ... shared:105 master:102
273 239 8:2 /etc /mnt/tmp/etc ... master:105
```

From the above, we see that */mnt* is the master of the slave */tmp/etc*, which in turn is the master of the slave */mnt/tmp/etc*.

We then *chroot*(1) to the */mnt* directory, which renders the mount with ID 267 unreachable from the (new) root directory:

```
# chroot /mnt
```

When we examine the state of the mounts inside the chroot-ed environment, we see the following:

```
# cat /proc/self/mountinfo | sed 's/ - .*//'
239 61 8:2 / / ... shared:102
248 239 0:4 / /proc ... shared:5
273 239 8:2 /etc /tmp/etc ... master:105 propagate_from:102
```

Above, we see that the mount with ID 273 is a slave whose master is the peer group 105. The mount point for that master is unreachable, and so a *propagate_from* tag is displayed, indicating that the closest dominant peer group (i.e., the nearest reachable mount in the slave chain) is the peer group with the ID 102 (corresponding to the */mnt* mount point before the *chroot*(1) was performed).

STANDARDS

Linux.

HISTORY

Linux 2.4.19.

NOTES

The propagation type assigned to a new mount depends on the propagation type of the parent mount. If the mount has a parent (i.e., it is a non-root mount point) and the propagation type of the parent is **MS_SHARED**, then the propagation type of the new mount is also **MS_SHARED**. Otherwise, the propagation type of the new mount is **MS_PRIVATE**.

Notwithstanding the fact that the default propagation type for new mount is in many cases **MS_PRIVATE**, **MS_SHARED** is typically more useful. For this reason, *systemd*(1) automatically remounts all mounts as **MS_SHARED** on system startup. Thus, on most modern systems, the default propagation type is in practice **MS_SHARED**.

Since, when one uses *unshare*(1) to create a mount namespace, the goal is commonly to provide full isolation of the mounts in the new namespace, *unshare*(1) (since *util-linux* 2.27) in turn reverses the step performed by *systemd*(1), by making all mounts private in the new namespace. That is, *unshare*(1) performs the equivalent of the following in the new mount namespace:

```
mount --make-rprivate /
```

To prevent this, one can use the *−−propagation unchanged* option to *unshare*(1)

An application that creates a new mount namespace directly using *clone*(2) or *unshare*(2) may desire to prevent propagation of mount events to other mount namespaces (as is done by *unshare*(1)). This can be done by changing the propagation type of mounts in the new namespace to either **MS_SLAVE** or **MS_PRIVATE**, using a call such as the following:

```
mount(NULL, "/", MS_SLAVE | MS_REC, NULL);
```

For a discussion of propagation types when moving mounts (**MS_MOVE**) and creating bind mounts (**MS_BIND**), see *Documentation/filesystems/sharedsubtree.rst*.

Restrictions on mount namespaces

Note the following points with respect to mount namespaces:

[1] Each mount namespace has an owner user namespace. As explained above, when a new mount namespace is created, its mount list is initialized as a copy of the mount list of another mount namespace. If the new namespace and the namespace from which the mount list was copied are owned by different user namespaces, then the new mount namespace is considered *less privileged*.

[2] When creating a less privileged mount namespace, shared mounts are reduced to slave mounts. This ensures that mappings performed in less privileged mount namespaces will not propagate to more privileged mount namespaces.

[3] Mounts that come as a single unit from a more privileged mount namespace are locked together and may not be separated in a less privileged mount namespace. (The *unshare*(2) **CLONE_NEWNS** operation brings across all of the mounts from the original mount namespace as a single unit, and recursive mounts that propagate between mount namespaces propagate as a single unit.)

In this context, "may not be separated" means that the mounts are locked so that they may not be individually unmounted. Consider the following example:

```
$ sudo sh
# mount --bind /dev/null /etc/shadow
# cat /etc/shadow       # Produces no output
```

The above steps, performed in a more privileged mount namespace, have created a bind mount that obscures the contents of the shadow password file, */etc/shadow*. For security reasons, it should not be possible to *umount*(2) that mount in a less privileged mount namespace, since that would reveal the contents of */etc/shadow*.

Suppose we now create a new mount namespace owned by a new user namespace. The new mount namespace will inherit copies of all of the mounts from the previous mount namespace. However, those mounts will be locked because the new mount namespace is less privileged. Consequently, an attempt to *umount*(2) the mount fails as show in the following step:

```
# unshare --user --map-root-user --mount \
              strace -o /tmp/log \
              umount /mnt/dir
umount: /etc/shadow: not mounted.
# grep '^umount' /tmp/log
umount2("/etc/shadow", 0)     = -1 EINVAL (Invalid argument)
```

The error message from *mount*(8) is a little confusing, but the *strace*(1) output reveals that the underlying *umount2*(2) system call failed with the error **EINVAL**, which is the error that the kernel returns to indicate that the mount is locked.

Note, however, that it is possible to stack (and unstack) a mount on top of one of the inherited locked mounts in a less privileged mount namespace:

```
# echo 'aaaaa' > /tmp/a    # File to mount onto /etc/shadow
# unshare --user --map-root-user --mount \
    sh -c 'mount --bind /tmp/a /etc/shadow; cat /etc/shadow'
aaaaa
# umount /etc/shadow
```

The final *umount*(8) command above, which is performed in the initial mount namespace, makes the original */etc/shadow* file once more visible in that namespace.

[4] Following on from point [3], note that it is possible to *umount*(2) an entire subtree of mounts that propagated as a unit into a less privileged mount namespace, as illustrated in the following example.

First, we create new user and mount namespaces using *unshare*(1)In the new mount namespace, the propagation type of all mounts is set to private. We then create a shared bind mount at */mnt*, and a small hierarchy of mounts underneath that mount.

```
$ PS1='ns1# ' sudo unshare --user --map-root-user \
                    --mount --propagation private bash
ns1# echo $$        # We need the PID of this shell later
778501
ns1# mount --make-shared --bind /mnt /mnt
ns1# mkdir /mnt/x
ns1# mount --make-private -t tmpfs none /mnt/x
ns1# mkdir /mnt/x/y
ns1# mount --make-private -t tmpfs none /mnt/x/y
ns1# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
986 83 8:5 /mnt /mnt rw,relatime shared:344
989 986 0:56 / /mnt/x rw,relatime
990 989 0:57 / /mnt/x/y rw,relatime
```

Continuing in the same shell session, we then create a second shell in a new user namespace and a new (less privileged) mount namespace and check the state of the propagated mounts rooted at */mnt*.

```
ns1# PS1='ns2# ' unshare --user --map-root-user \
                    --mount --propagation unchanged bash
ns2# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
1239 1204 8:5 /mnt /mnt rw,relatime master:344
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
```

Of note in the above output is that the propagation type of the mount */mnt* has been reduced to slave, as explained in point [2]. This means that submount events will propagate from the master */mnt* in "ns1", but propagation will not occur in the opposite direction.

From a separate terminal window, we then use *nsenter*(1) to enter the mount and user namespaces corresponding to "ns1". In that terminal window, we then recursively bind mount */mnt/x* at the location */mnt/ppp*.

```
$ PS1='ns3# ' sudo nsenter -t 778501 --user --mount
ns3# mount --rbind --make-private /mnt/x /mnt/ppp
ns3# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
986 83 8:5 /mnt /mnt rw,relatime shared:344
989 986 0:56 / /mnt/x rw,relatime
990 989 0:57 / /mnt/x/y rw,relatime
1242 986 0:56 / /mnt/ppp rw,relatime
1243 1242 0:57 / /mnt/ppp/y rw,relatime shared:518
```

Because the propagation type of the parent mount, */mnt*, was shared, the recursive bind mount propagated a small subtree of mounts under the slave mount */mnt* into "ns2", as can be verified by executing the following command in that shell session:

```
ns2# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
1239 1204 8:5 /mnt /mnt rw,relatime master:344
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
1244 1239 0:56 / /mnt/ppp rw,relatime
1245 1244 0:57 / /mnt/ppp/y rw,relatime master:518
```

While it is not possible to *umount*(2) a part of the propagated subtree (*/mnt/ppp/y*) in "ns2", it is possible to *umount*(2) the entire subtree, as shown by the following commands:

```
ns2# umount /mnt/ppp/y
umount: /mnt/ppp/y: not mounted.
ns2# umount -l /mnt/ppp | sed 's/ - .*//'      # Succeeds...
ns2# grep /mnt /proc/self/mountinfo
1239 1204 8:5 /mnt /mnt rw,relatime master:344
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
```

[5] The *mount*(2) flags **MS_RDONLY**, **MS_NOSUID**, **MS_NOEXEC**, and the "atime" flags (**MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME**) settings become locked when propagated from a more privileged to a less privileged mount namespace, and may not be changed in the less privileged mount namespace.

This point is illustrated in the following example where, in a more privileged mount namespace, we create a bind mount that is marked as read-only. For security reasons, it should not be possible to make the mount writable in a less privileged mount namespace, and indeed the kernel prevents this:

```
$ sudo mkdir /mnt/dir
$ sudo mount --bind -o ro /some/path /mnt/dir
$ sudo unshare --user --map-root-user --mount \
               mount -o remount,rw /mnt/dir
mount: /mnt/dir: permission denied.
```

[6] A file or directory that is a mount point in one namespace that is not a mount point in another namespace, may be renamed, unlinked, or removed (**rmdir**(2)) in the mount namespace in which it is not a mount point (subject to the usual permission checks). Consequently, the mount point is removed in the mount namespace where it was a mount point.

Previously (before Linux 3.18), attempting to unlink, rename, or remove a file or directory that was a mount point in another mount namespace would result in the error **EBUSY**. That behavior had technical problems of enforcement (e.g., for NFS) and permitted denial-of-service attacks against more privileged users (i.e., preventing individual files from being updated by bind mounting on top of them).

EXAMPLES

See *pivot_root*(2).

SEE ALSO

unshare(1), *clone*(2), *mount*(2), *mount_setattr*(2), *pivot_root*(2), *setns*(2), *umount*(2), *unshare*(2), *proc*(5), *namespaces*(7), *user_namespaces*(7), *findmnt*(8), *mount*(8), *pam_namespace*(8), *pivot_root*(8), *umount*(8)

Documentation/filesystems/sharedsubtree.rst in the kernel source tree.

NAME

mq_overview − overview of POSIX message queues

DESCRIPTION

POSIX message queues allow processes to exchange data in the form of messages. This API is distinct from that provided by System V message queues (**msgget**(2), *msgsnd*(2), *msgrcv*(2), etc.), but provides similar functionality.

Message queues are created and opened using *mq_open*(3); this function returns a *message queue descriptor* (*mqd_t*), which is used to refer to the open message queue in later calls. Each message queue is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX** (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same queue by passing the same name to *mq_open*(3).

Messages are transferred to and from a queue using *mq_send*(3) and *mq_receive*(3). When a process has finished using the queue, it closes it using *mq_close*(3), and when the queue is no longer required, it can be deleted using *mq_unlink*(3). Queue attributes can be retrieved and (in some cases) modified using *mq_getattr*(3) and *mq_setattr*(3). A process can request asynchronous notification of the arrival of a message on a previously empty queue using *mq_notify*(3).

A message queue descriptor is a reference to an *open message queue description* (see *open*(2)). After a *fork*(2), a child inherits copies of its parent's message queue descriptors, and these descriptors refer to the same open message queue descriptions as the corresponding message queue descriptors in the parent. Corresponding message queue descriptors in the two processes share the flags (*mq_flags*) that are associated with the open message queue description.

Each message has an associated *priority*, and messages are always delivered to the receiving process highest priority first. Message priorities range from 0 (low) to *sysconf(_SC_MQ_PRIO_MAX) − 1* (high). On Linux, *sysconf(_SC_MQ_PRIO_MAX)* returns 32768, but POSIX.1 requires only that an implementation support at least priorities in the range 0 to 31; some implementations provide only this range.

The remainder of this section describes some specific details of the Linux implementation of POSIX message queues.

Library interfaces and system calls

In most cases the **mq_***() library interfaces listed above are implemented on top of underlying system calls of the same name. Deviations from this scheme are indicated in the following table:

Library interface	System call
mq_close(3)	close(2)
mq_getattr(3)	mq_getsetattr(2)
mq_notify(3)	mq_notify(2)
mq_open(3)	mq_open(2)
mq_receive(3)	mq_timedreceive(2)
mq_send(3)	mq_timedsend(2)
mq_setattr(3)	mq_getsetattr(2)
mq_timedreceive(3)	mq_timedreceive(2)
mq_timedsend(3)	mq_timedsend(2)
mq_unlink(3)	mq_unlink(2)

Versions

POSIX message queues have been supported since Linux 2.6.6. glibc support has been provided since glibc 2.3.4.

Kernel configuration

Support for POSIX message queues is configurable via the **CONFIG_POSIX_MQUEUE** kernel configuration option. This option is enabled by default.

Persistence

POSIX message queues have kernel persistence: if not removed by *mq_unlink*(3), a message queue will exist until the system is shut down.

Linking

Programs using the POSIX message queue API must be compiled with *cc −lrt* to link against the real-time library, *librt*.

/proc interfaces

The following interfaces can be used to limit the amount of kernel memory consumed by POSIX message queues and to set the default attributes for new message queues:

/proc/sys/fs/mqueue/msg_default (since Linux 3.5)

This file defines the value used for a new queue's *mq_maxmsg* setting when the queue is created with a call to *mq_open*(3) where *attr* is specified as NULL. The default value for this file is 10. The minimum and maximum are as for */proc/sys/fs/mqueue/msg_max*. A new queue's default *mq_maxmsg* value will be the smaller of *msg_default* and *msg_max*. Before Linux 2.6.28, the default *mq_maxmsg* was 10; from Linux 2.6.28 to Linux 3.4, the default was the value defined for the *msg_max* limit.

/proc/sys/fs/mqueue/msg_max

This file can be used to view and change the ceiling value for the maximum number of messages in a queue. This value acts as a ceiling on the *attr->mq_maxmsg* argument given to *mq_open*(3). The default value for *msg_max* is 10. The minimum value is 1 (10 before Linux 2.6.28). The upper limit is **HARD_MSGMAX**. The *msg_max* limit is ignored for privileged processes (**CAP_SYS_RESOURCE**), but the **HARD_MSGMAX** ceiling is nevertheless imposed.

The definition of **HARD_MSGMAX** has changed across kernel versions:

- Up to Linux 2.6.32: *131072 / sizeof(void *)*
- Linux 2.6.33 to Linux 3.4: *(32768 * sizeof(void *) / 4)*
- Since Linux 3.5: 65,536

/proc/sys/fs/mqueue/msgsize_default (since Linux 3.5)

This file defines the value used for a new queue's *mq_msgsize* setting when the queue is created with a call to *mq_open*(3) where *attr* is specified as NULL. The default value for this file is 8192 (bytes). The minimum and maximum are as for */proc/sys/fs/mqueue/msgsize_max*. If *msgsize_default* exceeds *msgsize_max*, a new queue's default *mq_msgsize* value is capped to the *msgsize_max* limit. Before Linux 2.6.28, the default *mq_msgsize* was 8192; from Linux 2.6.28 to Linux 3.4, the default was the value defined for the *msgsize_max* limit.

/proc/sys/fs/mqueue/msgsize_max

This file can be used to view and change the ceiling on the maximum message size. This value acts as a ceiling on the *attr->mq_msgsize* argument given to *mq_open*(3). The default value for *msgsize_max* is 8192 bytes. The minimum value is 128 (8192 before Linux 2.6.28). The upper limit for *msgsize_max* has varied across kernel versions:

- Before Linux 2.6.28, the upper limit is **INT_MAX**.
- From Linux 2.6.28 to Linux 3.4, the limit is 1,048,576.
- Since Linux 3.5, the limit is 16,777,216 (**HARD_MSGSIZEMAX**).

The *msgsize_max* limit is ignored for privileged process (**CAP_SYS_RESOURCE**), but, since Linux 3.5, the **HARD_MSGSIZEMAX** ceiling is enforced for privileged processes.

/proc/sys/fs/mqueue/queues_max

This file can be used to view and change the system-wide limit on the number of message queues that can be created. The default value for *queues_max* is 256. No ceiling is imposed on the *queues_max* limit; privileged processes (**CAP_SYS_RESOURCE**) can exceed the limit (but see BUGS).

Resource limit

The **RLIMIT_MSGQUEUE** resource limit, which places a limit on the amount of space that can be consumed by all of the message queues belonging to a process's real user ID, is described in *getrlimit*(2).

Mounting the message queue filesystem

On Linux, message queues are created in a virtual filesystem. (Other implementations may also provide such a feature, but the details are likely to differ.) This filesystem can be mounted (by the superuser) using the following commands:

```
# mkdir /dev/mqueue
```

```
# mount -t mqueue none /dev/mqueue
```

The sticky bit is automatically enabled on the mount directory.

After the filesystem has been mounted, the message queues on the system can be viewed and manipulated using the commands usually used for files (e.g., *ls*(1) and *rm*(1)).

The contents of each file in the directory consist of a single line containing information about the queue:

```
$ cat /dev/mqueue/mymq
QSIZE:129     NOTIFY:2    SIGNO:0    NOTIFY_PID:8260
```

These fields are as follows:

QSIZE Number of bytes of data in all messages in the queue (but see BUGS).

NOTIFY_PID
If this is nonzero, then the process with this PID has used *mq_notify*(3) to register for asynchronous message notification, and the remaining fields describe how notification occurs.

NOTIFY
Notification method: 0 is **SIGEV_SIGNAL**; 1 is **SIGEV_NONE**; and 2 is **SIGEV_THREAD**.

SIGNO
Signal number to be used for **SIGEV_SIGNAL**.

Linux implementation of message queue descriptors

On Linux, a message queue descriptor is actually a file descriptor. (POSIX does not require such an implementation.) This means that a message queue descriptor can be monitored using *select*(2), *poll*(2), or *epoll*(7). This is not portable.

The close-on-exec flag (see *open*(2)) is automatically set on the file descriptor returned by *mq_open*(2).

IPC namespaces

For a discussion of the interaction of POSIX message queue objects and IPC namespaces, see *ipc_namespaces*(7).

NOTES

System V message queues (**msgget**(2), *msgsnd*(2), *msgrcv*(2), etc.) are an older API for exchanging messages between processes. POSIX message queues provide a better designed interface than System V message queues; on the other hand POSIX message queues are less widely available (especially on older systems) than System V message queues.

Linux does not currently (Linux 2.6.26) support the use of access control lists (ACLs) for POSIX message queues.

BUGS

Since Linux 3.5 to Linux 3.14, the kernel imposed a ceiling of 1024 (**HARD_QUEUESMAX**) on the value to which the *queues_max* limit could be raised, and the ceiling was enforced even for privileged processes. This ceiling value was removed in Linux 3.14, and patches to stable Linux 3.5.x to Linux 3.13.x also removed the ceiling.

As originally implemented (and documented), the QSIZE field displayed the total number of (user-supplied) bytes in all messages in the message queue. Some changes in Linux 3.5 inadvertently changed the behavior, so that this field also included a count of kernel overhead bytes used to store the messages in the queue. This behavioral regression was rectified in Linux 4.2 (and earlier stable kernel series), so that the count once more included just the bytes of user data in messages in the queue.

EXAMPLES

An example of the use of various message queue functions is shown in *mq_notify*(3).

SEE ALSO

getrlimit(2), *mq_getsetattr*(2), *poll*(2), *select*(2), *mq_close*(3), *mq_getattr*(3), *mq_notify*(3), *mq_open*(3), *mq_receive*(3), *mq_send*(3), *mq_unlink*(3), *epoll*(7), *namespaces*(7)

NAME

namespaces − overview of Linux namespaces

DESCRIPTION

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers.

This page provides pointers to information on the various namespace types, describes the associated */proc* files, and summarizes the APIs for working with namespaces.

Namespace types

The following table shows the namespace types available on Linux. The second column of the table shows the flag value that is used to specify the namespace type in various APIs. The third column identifies the manual page that provides details on the namespace type. The last column is a summary of the resources that are isolated by the namespace type.

Namespace	Flag	Page	Isolates
Cgroup	**CLONE_NEWCGROUP**	*cgroup_namespaces*(7)	Cgroup root directory
IPC	**CLONE_NEWIPC**	*ipc_namespaces*(7)	System V IPC, POSIX message queues
Network	**CLONE_NEWNET**	*network_namespaces*(7)	Network devices, stacks, ports, etc.
Mount	**CLONE_NEWNS**	*mount_namespaces*(7)	Mount points
PID	**CLONE_NEWPID**	*pid_namespaces*(7)	Process IDs
Time	**CLONE_NEWTIME**	*time_namespaces*(7)	Boot and monotonic clocks
User	**CLONE_NEWUSER**	*user_namespaces*(7)	User and group IDs
UTS	**CLONE_NEWUTS**	*uts_namespaces*(7)	Hostname and NIS domain name

The namespaces API

As well as various */proc* files described below, the namespaces API includes the following system calls:

clone(2)
: The *clone*(2) system call creates a new process. If the *flags* argument of the call specifies one or more of the **CLONE_NEW*** flags listed above, then new namespaces are created for each flag, and the child process is made a member of those namespaces. (This system call also implements a number of features unrelated to namespaces.)

setns(2)
: The *setns*(2) system call allows the calling process to join an existing namespace. The namespace to join is specified via a file descriptor that refers to one of the */proc/*pid*/ns* files described below.

unshare(2)
: The *unshare*(2) system call moves the calling process to a new namespace. If the *flags* argument of the call specifies one or more of the **CLONE_NEW*** flags listed above, then new namespaces are created for each flag, and the calling process is made a member of those namespaces. (This system call also implements a number of features unrelated to namespaces.)

ioctl(2) Various *ioctl*(2) operations can be used to discover information about namespaces. These operations are described in *ioctl_ns*(2).

Creation of new namespaces using *clone*(2) and *unshare*(2) in most cases requires the **CAP_SYS_ADMIN** capability, since, in the new namespace, the creator will have the power to change global resources that are visible to other processes that are subsequently created in, or join the namespace. User namespaces are the exception: since Linux 3.8, no privilege is required to create a user namespace.

The */proc/*pid*/ns/* directory

Each process has a */proc/*pid*/ns/* subdirectory containing one entry for each namespace that supports being manipulated by *setns*(2):

```
$ ls -l /proc/$$/ns | awk '{print $1, $9, $10, $11}'
total 0
lrwxrwxrwx. cgroup -> cgroup:[4026531835]
lrwxrwxrwx. ipc -> ipc:[4026531839]
lrwxrwxrwx. mnt -> mnt:[4026531840]
lrwxrwxrwx. net -> net:[4026531969]
lrwxrwxrwx. pid -> pid:[4026531836]
lrwxrwxrwx. pid_for_children -> pid:[4026531834]
lrwxrwxrwx. time -> time:[4026531834]
lrwxrwxrwx. time_for_children -> time:[4026531834]
lrwxrwxrwx. user -> user:[4026531837]
lrwxrwxrwx. uts -> uts:[4026531838]
```

Bind mounting (see *mount*(2)) one of the files in this directory to somewhere else in the filesystem keeps the corresponding namespace of the process specified by *pid* alive even if all processes currently in the namespace terminate.

Opening one of the files in this directory (or a file that is bind mounted to one of these files) returns a file handle for the corresponding namespace of the process specified by *pid*. As long as this file descriptor remains open, the namespace will remain alive, even if all processes in the namespace terminate. The file descriptor can be passed to *setns*(2).

In Linux 3.7 and earlier, these files were visible as hard links. Since Linux 3.8, they appear as symbolic links. If two processes are in the same namespace, then the device IDs and inode numbers of their */proc/*pid*/ns/*xxx symbolic links will be the same; an application can check this using the *stat.st_dev* and *stat.st_ino* fields returned by *stat*(2). The content of this symbolic link is a string containing the namespace type and inode number as in the following example:

```
$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

The symbolic links in this subdirectory are as follows:

*/proc/*pid*/ns/cgroup* (since Linux 4.6)
: This file is a handle for the cgroup namespace of the process.

*/proc/*pid*/ns/ipc* (since Linux 3.0)
: This file is a handle for the IPC namespace of the process.

*/proc/*pid*/ns/mnt* (since Linux 3.8)
: This file is a handle for the mount namespace of the process.

*/proc/*pid*/ns/net* (since Linux 3.0)
: This file is a handle for the network namespace of the process.

*/proc/*pid*/ns/pid* (since Linux 3.8)
: This file is a handle for the PID namespace of the process. This handle is permanent for the lifetime of the process (i.e., a process's PID namespace membership never changes).

*/proc/*pid*/ns/pid_for_children* (since Linux 4.12)
: This file is a handle for the PID namespace of child processes created by this process. This can change as a consequence of calls to *unshare*(2) and *setns*(2) (see *pid_namespaces*(7)), so the file may differ from */proc/*pid*/ns/pid*. The symbolic link gains a value only after the first child process is created in the namespace. (Beforehand, *readlink*(2) of the symbolic link will return an empty buffer.)

*/proc/*pid*/ns/time* (since Linux 5.6)
: This file is a handle for the time namespace of the process.

*/proc/*pid*/ns/time_for_children* (since Linux 5.6)
: This file is a handle for the time namespace of child processes created by this process. This can change as a consequence of calls to *unshare*(2) and *setns*(2) (see *time_namespaces*(7)), so the file may differ from */proc/*pid*/ns/time*.

*/proc/*pid*/ns/user* (since Linux 3.8)
This file is a handle for the user namespace of the process.

*/proc/*pid*/ns/uts* (since Linux 3.0)
This file is a handle for the UTS namespace of the process.

Permission to dereference or read (**readlink**(2)) these symbolic links is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS** check; see *ptrace*(2).

The */proc/sys/user* directory

The files in the */proc/sys/user* directory (which is present since Linux 4.9) expose limits on the number of namespaces of various types that can be created. The files are as follows:

max_cgroup_namespaces
The value in this file defines a per-user limit on the number of cgroup namespaces that may be created in the user namespace.

max_ipc_namespaces
The value in this file defines a per-user limit on the number of ipc namespaces that may be created in the user namespace.

max_mnt_namespaces
The value in this file defines a per-user limit on the number of mount namespaces that may be created in the user namespace.

max_net_namespaces
The value in this file defines a per-user limit on the number of network namespaces that may be created in the user namespace.

max_pid_namespaces
The value in this file defines a per-user limit on the number of PID namespaces that may be created in the user namespace.

max_time_namespaces (since Linux 5.7)
The value in this file defines a per-user limit on the number of time namespaces that may be created in the user namespace.

max_user_namespaces
The value in this file defines a per-user limit on the number of user namespaces that may be created in the user namespace.

max_uts_namespaces
The value in this file defines a per-user limit on the number of uts namespaces that may be created in the user namespace.

Note the following details about these files:

- The values in these files are modifiable by privileged processes.
- The values exposed by these files are the limits for the user namespace in which the opening process resides.
- The limits are per-user. Each user in the same user namespace can create namespaces up to the defined limit.
- The limits apply to all users, including UID 0.
- These limits apply in addition to any other per-namespace limits (such as those for PID and user namespaces) that may be enforced.
- Upon encountering these limits, *clone*(2) and *unshare*(2) fail with the error **ENOSPC**.
- For the initial user namespace, the default value in each of these files is half the limit on the number of threads that may be created (*/proc/sys/kernel/threads-max*). In all descendant user namespaces, the default value in each file is **MAXINT**.
- When a namespace is created, the object is also accounted against ancestor namespaces. More precisely:

- Each user namespace has a creator UID.
- When a namespace is created, it is accounted against the creator UIDs in each of the ancestor user namespaces, and the kernel ensures that the corresponding namespace limit for the creator UID in the ancestor namespace is not exceeded.
- The aforementioned point ensures that creating a new user namespace cannot be used as a means to escape the limits in force in the current user namespace.

Namespace lifetime

Absent any other factors, a namespace is automatically torn down when the last process in the namespace terminates or leaves the namespace. However, there are a number of other factors that may pin a namespace into existence even though it has no member processes. These factors include the following:

- An open file descriptor or a bind mount exists for the corresponding */proc/*pid*/ns/** file.
- The namespace is hierarchical (i.e., a PID or user namespace), and has a child namespace.
- It is a user namespace that owns one or more nonuser namespaces.
- It is a PID namespace, and there is a process that refers to the namespace via a */proc/*pid*/ns/pid_for_children* symbolic link.
- It is a time namespace, and there is a process that refers to the namespace via a */proc/*pid*/ns/time_for_children* symbolic link.
- It is an IPC namespace, and a corresponding mount of an *mqueue* filesystem (see *mq_overview*(7)) refers to this namespace.
- It is a PID namespace, and a corresponding mount of a *proc*(5) filesystem refers to this namespace.

EXAMPLES

See *clone*(2) and *user_namespaces*(7).

SEE ALSO

nsenter(1), *readlink*(1), *unshare*(1), *clone*(2), *ioctl_ns*(2), *setns*(2), *unshare*(2), *proc*(5), *capabilities*(7), *cgroup_namespaces*(7), *cgroups*(7), *credentials*(7), *ipc_namespaces*(7), *network_namespaces*(7), *pid_namespaces*(7), *user_namespaces*(7), *uts_namespaces*(7), *lsns*(8), *switch_root*(8)

NAME

netdevice − low-level access to Linux network devices

SYNOPSIS

#include <sys/ioctl.h>
#include <net/if.h>

DESCRIPTION

This man page describes the sockets interface which is used to configure network devices.

Linux supports some standard ioctls to configure network devices. They can be used on any socket's file descriptor regardless of the family or type. Most of them pass an *ifreq* structure:

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short           ifr_flags;
        int             ifr_ifindex;
        int             ifr_metric;
        int             ifr_mtu;
        struct ifmap    ifr_map;
        char            ifr_slave[IFNAMSIZ];
        char            ifr_newname[IFNAMSIZ];
        char           *ifr_data;
    };
};
```

AF_INET6 is an exception. It passes an *in6_ifreq* structure:

```
struct in6_ifreq {
    struct in6_addr     ifr6_addr;
    u32                 ifr6_prefixlen;
    int                 ifr6_ifindex; /* Interface index */
};
```

Normally, the user specifies which device to affect by setting *ifr_name* to the name of the interface or *ifr6_ifindex* to the index of the interface. All other members of the structure may share memory.

Ioctls

If an ioctl is marked as privileged, then using it requires an effective user ID of 0 or the **CAP_NET_ADMIN** capability. If this is not the case, **EPERM** will be returned.

SIOCGIFNAME
Given the *ifr_ifindex*, return the name of the interface in *ifr_name*. This is the only ioctl which returns its result in *ifr_name*.

SIOCGIFINDEX
Retrieve the interface index of the interface into *ifr_ifindex*.

SIOCGIFFLAGS
SIOCSIFFLAGS
Get or set the active flag word of the device. *ifr_flags* contains a bit mask of the following values:

Device flags	
IFF_UP	Interface is running.
IFF_BROADCAST	Valid broadcast address set.
IFF_DEBUG	Internal debugging flag.
IFF_LOOPBACK	Interface is a loopback interface.
IFF_POINTOPOINT	Interface is a point-to-point link.

IFF_RUNNING	Resources allocated.
IFF_NOARP	No arp protocol, L2 destination address not set.
IFF_PROMISC	Interface is in promiscuous mode.
IFF_NOTRAILERS	Avoid use of trailers.
IFF_ALLMULTI	Receive all multicast packets.
IFF_MASTER	Master of a load balancing bundle.
IFF_SLAVE	Slave of a load balancing bundle.
IFF_MULTICAST	Supports multicast
IFF_PORTSEL	Is able to select media type via ifmap.
IFF_AUTOMEDIA	Auto media selection active.
IFF_DYNAMIC	The addresses are lost when the interface goes down.
IFF_LOWER_UP	Driver signals L1 up (since Linux 2.6.17)
IFF_DORMANT	Driver signals dormant (since Linux 2.6.17)
IFF_ECHO	Echo sent packets (since Linux 2.6.25)

Setting the active flag word is a privileged operation, but any process may read it.

SIOCGIFPFLAGS
SIOCSIFPFLAGS

Get or set extended (private) flags for the device. *ifr_flags* contains a bit mask of the following values:

	Private flags
IFF_802_1Q_VLAN	Interface is 802.1Q VLAN device.
IFF_EBRIDGE	Interface is Ethernet bridging device.
IFF_SLAVE_INACTIVE	Interface is inactive bonding slave.
IFF_MASTER_8023AD	Interface is 802.3ad bonding master.
IFF_MASTER_ALB	Interface is balanced-alb bonding master.
IFF_BONDING	Interface is a bonding master or slave.
IFF_SLAVE_NEEDARP	Interface needs ARPs for validation.
IFF_ISATAP	Interface is RFC4214 ISATAP interface.

Setting the extended (private) interface flags is a privileged operation.

SIOCGIFADDR
SIOCSIFADDR
SIOCDIFADDR

Get, set, or delete the address of the device using *ifr_addr*, or *ifr6_addr* with *ifr6_prefixlen*. Setting or deleting the interface address is a privileged operation. For compatibility, **SIOCGIFADDR** returns only **AF_INET** addresses, **SIOCSIFADDR** accepts **AF_INET** and **AF_INET6** addresses, and **SIOCDIFADDR** deletes only **AF_INET6** addresses. A **AF_INET** address can be deleted by setting it to zero via **SIOCSIFADDR**.

SIOCGIFDSTADDR
SIOCSIFDSTADDR

Get or set the destination address of a point-to-point device using *ifr_dstaddr*. For compatibility, only **AF_INET** addresses are accepted or returned. Setting the destination address is a privileged operation.

SIOCGIFBRDADDR
SIOCSIFBRDADDR

Get or set the broadcast address for a device using *ifr_brdaddr*. For compatibility, only **AF_INET** addresses are accepted or returned. Setting the broadcast address is a privileged operation.

SIOCGIFNETMASK
SIOCSIFNETMASK

Get or set the network mask for a device using *ifr_netmask*. For compatibility, only **AF_INET** addresses are accepted or returned. Setting the network mask is a privileged operation.

SIOCGIFMETRIC
SIOCSIFMETRIC
Get or set the metric of the device using *ifr_metric*. This is currently not implemented; it sets *ifr_metric* to 0 if you attempt to read it and returns **EOPNOTSUPP** if you attempt to set it.

SIOCGIFMTU
SIOCSIFMTU
Get or set the MTU (Maximum Transfer Unit) of a device using *ifr_mtu*. Setting the MTU is a privileged operation. Setting the MTU to too small values may cause kernel crashes.

SIOCGIFHWADDR
SIOCSIFHWADDR
Get or set the hardware address of a device using *ifr_hwaddr*. The hardware address is specified in a struct *sockaddr*. *sa_family* contains the ARPHRD_* device type, *sa_data* the L2 hardware address starting from byte 0. Setting the hardware address is a privileged operation.

SIOCSIFHWBROADCAST
Set the hardware broadcast address of a device from *ifr_hwaddr*. This is a privileged operation.

SIOCGIFMAP
SIOCSIFMAP
Get or set the interface's hardware parameters using *ifr_map*. Setting the parameters is a privileged operation.

```
struct ifmap {
    unsigned long   mem_start;
    unsigned long   mem_end;
    unsigned short  base_addr;
    unsigned char   irq;
    unsigned char   dma;
    unsigned char   port;
};
```

The interpretation of the ifmap structure depends on the device driver and the architecture.

SIOCADDMULTI
SIOCDELMULTI
Add an address to or delete an address from the device's link layer multicast filters using *ifr_hwaddr*. These are privileged operations. See also *packet*(7) for an alternative.

SIOCGIFTXQLEN
SIOCSIFTXQLEN
Get or set the transmit queue length of a device using *ifr_qlen*. Setting the transmit queue length is a privileged operation.

SIOCSIFNAME
Changes the name of the interface specified in *ifr_name* to *ifr_newname*. This is a privileged operation. It is allowed only when the interface is not up.

SIOCGIFCONF
Return a list of interface (network layer) addresses. This currently means only addresses of the **AF_INET** (IPv4) family for compatibility. Unlike the others, this ioctl passes an *ifconf* structure:

```
struct ifconf {
    int                ifc_len; /* size of buffer */
    union {
        char          *ifc_buf; /* buffer address */
        struct ifreq *ifc_req; /* array of structures */
    };
};
```

If *ifc_req* is NULL, **SIOCGIFCONF** returns the necessary buffer size in bytes for receiving all available addresses in *ifc_len*. Otherwise, *ifc_req* contains a pointer to an array of *ifreq* structures to be filled with all currently active L3 interface addresses. *ifc_len* contains the size

of the array in bytes. Within each *ifreq* structure, *ifr_name* will receive the interface name, and *ifr_addr* the address. The actual number of bytes transferred is returned in *ifc_len*.

If the size specified by *ifc_len* is insufficient to store all the addresses, the kernel will skip the exceeding ones and return success. There is no reliable way of detecting this condition once it has occurred. It is therefore recommended to either determine the necessary buffer size beforehand by calling **SIOCGIFCONF** with *ifc_req* set to NULL, or to retry the call with a bigger buffer whenever *ifc_len* upon return differs by less than *sizeof(struct ifreq)* from its original value.

If an error occurs accessing the *ifconf* or *ifreq* structures, **EFAULT** will be returned.

Most protocols support their own ioctls to configure protocol-specific interface options. See the protocol man pages for a description. For configuring IP addresses, see *ip*(7).

In addition, some devices support private ioctls. These are not described here.

NOTES

SIOCGIFCONF and the other ioctls that accept or return only **AF_INET** socket addresses are IP-specific and perhaps should rather be documented in *ip*(7).

The names of interfaces with no addresses or that don't have the **IFF_RUNNING** flag set can be found via */proc/net/dev*.

AF_INET6 IPv6 addresses can be read from */proc/net/if_inet6* or via *rtnetlink*(7). Adding a new IPv6 address and deleting an existing IPv6 address can be done via **SIOCSIFADDR** and **SIOCDIFADDR** or via *rtnetlink*(7). Retrieving or changing destination IPv6 addresses of a point-to-point interface is possible only via *rtnetlink*(7).

BUGS

glibc 2.1 is missing the *ifr_newname* macro in <*net/if.h*>. Add the following to your program as a workaround:

```
#ifndef ifr_newname
#define ifr_newname     ifr_ifru.ifru_slave
#endif
```

SEE ALSO

proc(5), *capabilities*(7), *ip*(7), *rtnetlink*(7)

NAME

netlink − communication between kernel and user space (AF_NETLINK)

SYNOPSIS

#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

netlink_socket = socket(AF_NETLINK, *socket_type***,** *netlink_family***);**

DESCRIPTION

Netlink is used to transfer information between the kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules. The internal kernel interface is not documented in this manual page. There is also an obsolete netlink interface via netlink character devices; this interface is not documented here and is provided only for backward compatibility.

Netlink is a datagram-oriented service. Both **SOCK_RAW** and **SOCK_DGRAM** are valid values for *socket_type*. However, the netlink protocol does not distinguish between datagram and raw sockets.

netlink_family selects the kernel module or netlink group to communicate with. The currently assigned netlink families are:

NETLINK_ROUTE
Receives routing and link updates and may be used to modify the routing tables (both IPv4 and IPv6), IP addresses, link parameters, neighbor setups, queueing disciplines, traffic classes, and packet classifiers (see *rtnetlink*(7)).

NETLINK_W1 (Linux 2.6.13 to Linux 2.16.17)
Messages from 1-wire subsystem.

NETLINK_USERSOCK
Reserved for user-mode socket protocols.

NETLINK_FIREWALL (up to and including Linux 3.4)
Transport IPv4 packets from netfilter to user space. Used by *ip_queue* kernel module. After a long period of being declared obsolete (in favor of the more advanced *nfnetlink_queue* feature), **NETLINK_FIREWALL** was removed in Linux 3.5.

NETLINK_SOCK_DIAG (since Linux 3.3)
Query information about sockets of various protocol families from the kernel (see *sock_diag*(7)).

NETLINK_INET_DIAG (since Linux 2.6.14)
An obsolete synonym for **NETLINK_SOCK_DIAG**.

NETLINK_NFLOG (up to and including Linux 3.16)
Netfilter/iptables ULOG.

NETLINK_XFRM
IPsec.

NETLINK_SELINUX (since Linux 2.6.4)
SELinux event notifications.

NETLINK_ISCSI (since Linux 2.6.15)
Open-iSCSI.

NETLINK_AUDIT (since Linux 2.6.6)
Auditing.

NETLINK_FIB_LOOKUP (since Linux 2.6.13)
Access to FIB lookup from user space.

NETLINK_CONNECTOR (since Linux 2.6.14)
Kernel connector. See *Documentation/driver−api/connector.rst* (or */Documentation/connector/connector.** in Linux 5.2 and earlier) in the Linux kernel source tree for further information.

NETLINK_NETFILTER (since Linux 2.6.14)
Netfilter subsystem.

NETLINK_SCSITRANSPORT (since Linux 2.6.19)
SCSI Transports.

NETLINK_RDMA (since Linux 3.0)
Infiniband RDMA.

NETLINK_IP6_FW (up to and including Linux 3.4)
Transport IPv6 packets from netfilter to user space. Used by *ip6_queue* kernel module.

NETLINK_DNRTMSG
DECnet routing messages.

NETLINK_KOBJECT_UEVENT (since Linux 2.6.10)
Kernel messages to user space.

NETLINK_GENERIC (since Linux 2.6.15)
Generic netlink family for simplified netlink usage.

NETLINK_CRYPTO (since Linux 3.2)
Netlink interface to request information about ciphers registered with the kernel crypto API as well as allow configuration of the kernel crypto API.

Netlink messages consist of a byte stream with one or multiple *nlmsghdr* headers and associated payload. The byte stream should be accessed only with the standard **NLMSG_*** macros. See *netlink*(3) for further information.

In multipart messages (multiple *nlmsghdr* headers with associated payload in one byte stream) the first and all following headers have the **NLM_F_MULTI** flag set, except for the last header which has the type **NLMSG_DONE**.

After each *nlmsghdr* the payload follows.

```
struct nlmsghdr {
    __u32 nlmsg_len;    /* Length of message including header */
    __u16 nlmsg_type;   /* Type of message content */
    __u16 nlmsg_flags;  /* Additional flags */
    __u32 nlmsg_seq;    /* Sequence number */
    __u32 nlmsg_pid;    /* Sender port ID */
};
```

nlmsg_type can be one of the standard message types: **NLMSG_NOOP** message is to be ignored, **NLMSG_ERROR** message signals an error and the payload contains an *nlmsgerr* structure, **NLMSG_DONE** message terminates a multipart message. Error messages get the original request appended, unless the user requests to cap the error message, and get extra error data if requested.

```
struct nlmsgerr {
    int error;        /* Negative errno or 0 for acknowledgements */
    struct nlmsghdr msg;  /* Message header that caused the error */
    /*
     * followed by the message contents
     * unless NETLINK_CAP_ACK was set
     * or the ACK indicates success (error == 0).
     * For example Generic Netlink message with attributes.
     * message length is aligned with NLMSG_ALIGN()
     */
    /*
     * followed by TLVs defined in enum nlmsgerr_attrs
     * if NETLINK_EXT_ACK was set
     */
};
```

A netlink family usually specifies more message types, see the appropriate manual pages for that, for example, *rtnetlink*(7) for **NETLINK_ROUTE**.

Standard flag bits in *nlmsg_flags*	
NLM_F_REQUEST	Must be set on all request messages.
NLM_F_MULTI	The message is part of a multipart message terminated by **NLMSG_DONE**.
NLM_F_ACK	Request for an acknowledgement on success.
NLM_F_ECHO	Echo this request.

Additional flag bits for GET requests	
NLM_F_ROOT	Return the complete table instead of a single entry.
NLM_F_MATCH	Return all entries matching criteria passed in message content. Not implemented yet.
NLM_F_ATOMIC	Return an atomic snapshot of the table.
NLM_F_DUMP	Convenience macro; equivalent to (NLM_F_ROOT|NLM_F_MATCH).

Note that **NLM_F_ATOMIC** requires the **CAP_NET_ADMIN** capability or an effective UID of 0.

Additional flag bits for NEW requests	
NLM_F_REPLACE	Replace existing matching object.
NLM_F_EXCL	Don't replace if the object already exists.
NLM_F_CREATE	Create object if it doesn't already exist.
NLM_F_APPEND	Add to the end of the object list.

nlmsg_seq and *nlmsg_pid* are used to track messages. *nlmsg_pid* shows the origin of the message. Note that there isn't a 1:1 relationship between *nlmsg_pid* and the PID of the process if the message originated from a netlink socket. See the **ADDRESS FORMATS** section for further information.

Both *nlmsg_seq* and *nlmsg_pid* are opaque to netlink core.

Netlink is not a reliable protocol. It tries its best to deliver a message to its destination(s), but may drop messages when an out-of-memory condition or other error occurs. For reliable transfer the sender can request an acknowledgement from the receiver by setting the **NLM_F_ACK** flag. An acknowledgement is an **NLMSG_ERROR** packet with the error field set to 0. The application must generate acknowledgements for received messages itself. The kernel tries to send an **NLMSG_ERROR** message for every failed packet. A user process should follow this convention too.

However, reliable transmissions from kernel to user are impossible in any case. The kernel can't send a netlink message if the socket buffer is full: the message will be dropped and the kernel and the user-space process will no longer have the same view of kernel state. It is up to the application to detect when this happens (via the **ENOBUFS** error returned by *recvmsg*(2)) and resynchronize.

Address formats

The *sockaddr_nl* structure describes a netlink client in user space or in the kernel. A *sockaddr_nl* can be either unicast (only sent to one peer) or sent to netlink multicast groups (*nl_groups* not equal 0).

```
struct sockaddr_nl {
    sa_family_t     nl_family;  /* AF_NETLINK */
    unsigned short  nl_pad;     /* Zero */
    pid_t           nl_pid;     /* Port ID */
    __u32           nl_groups;  /* Multicast groups mask */
};
```

nl_pid is the unicast address of netlink socket. It's always 0 if the destination is in the kernel. For a user-space process, *nl_pid* is usually the PID of the process owning the destination socket. However, *nl_pid* identifies a netlink socket, not a process. If a process owns several netlink sockets, then *nl_pid* can be equal to the process ID only for at most one socket. There are two ways to assign *nl_pid* to a netlink socket. If the application sets *nl_pid* before calling *bind*(2), then it is up to the application to make sure that *nl_pid* is unique. If the application sets it to 0, the kernel takes care of assigning it. The kernel assigns the process ID to the first netlink socket the process opens and assigns a unique *nl_pid* to every netlink socket that the process subsequently creates.

nl_groups is a bit mask with every bit representing a netlink group number. Each netlink family has a set of 32 multicast groups. When *bind*(2) is called on the socket, the *nl_groups* field in the *sockaddr_nl* should be set to a bit mask of the groups which it wishes to listen to. The default value for this field is zero which means that no multicasts will be received. A socket may multicast messages to any of the multicast groups by setting *nl_groups* to a bit mask of the groups it wishes to send to when it calls *sendmsg*(2) or does a *connect*(2). Only processes with an effective UID of 0 or the

CAP_NET_ADMIN capability may send or listen to a netlink multicast group. Since Linux 2.6.13, messages can't be broadcast to multiple groups. Any replies to a message received for a multicast group should be sent back to the sending PID and the multicast group. Some Linux kernel subsystems may additionally allow other users to send and/or receive messages. As at Linux 3.0, the **NETLINK_KOBJECT_UEVENT**, **NETLINK_GENERIC**, **NETLINK_ROUTE**, and **NETLINK_SELINUX** groups allow other users to receive messages. No groups allow other users to send messages.

Socket options

To set or get a netlink socket option, call *getsockopt*(2) to read or *setsockopt*(2) to write the option with the option level argument set to **SOL_NETLINK**. Unless otherwise noted, *optval* is a pointer to an *int*.

NETLINK_PKTINFO (since Linux 2.6.14)
: Enable **nl_pktinfo** control messages for received packets to get the extended destination group number.

NETLINK_ADD_MEMBERSHIP
NETLINK_DROP_MEMBERSHIP (since Linux 2.6.14)
: Join/leave a group specified by *optval*.

NETLINK_LIST_MEMBERSHIPS (since Linux 4.2)
: Retrieve all groups a socket is a member of. *optval* is a pointer to **__u32** and *optlen* is the size of the array. The array is filled with the full membership set of the socket, and the required array size is returned in *optlen*.

NETLINK_BROADCAST_ERROR (since Linux 2.6.30)
: When not set, **netlink_broadcast()** only reports **ESRCH** errors and silently ignore **ENOBUFS** errors.

NETLINK_NO_ENOBUFS (since Linux 2.6.30)
: This flag can be used by unicast and broadcast listeners to avoid receiving **ENOBUFS** errors.

NETLINK_LISTEN_ALL_NSID (since Linux 4.2)
: When set, this socket will receive netlink notifications from all network namespaces that have an *nsid* assigned into the network namespace where the socket has been opened. The *nsid* is sent to user space via an ancillary data.

NETLINK_CAP_ACK (since Linux 4.3)
: The kernel may fail to allocate the necessary room for the acknowledgement message back to user space. This option trims off the payload of the original netlink message. The netlink message header is still included, so the user can guess from the sequence number which message triggered the acknowledgement.

VERSIONS

The socket interface to netlink first appeared Linux 2.2.

Linux 2.0 supported a more primitive device-based netlink interface (which is still available as a compatibility option). This obsolete interface is not described here.

NOTES

It is often better to use netlink via *libnetlink* or *libnl* than via the low-level kernel interface.

BUGS

This manual page is not complete.

EXAMPLES

The following example creates a **NETLINK_ROUTE** netlink socket which will listen to the **RTMGRP_LINK** (network interface create/delete/up/down events) and **RTMGRP_IPV4_IFADDR** (IPv4 addresses add/delete events) multicast groups.

```
struct sockaddr_nl sa;

memset(&sa, 0, sizeof(sa));
sa.nl_family = AF_NETLINK;
sa.nl_groups = RTMGRP_LINK | RTMGRP_IPV4_IFADDR;
```

```
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
bind(fd, (struct sockaddr *) &sa, sizeof(sa));
```

The next example demonstrates how to send a netlink message to the kernel (pid 0). Note that the application must take care of message sequence numbers in order to reliably track acknowledgements.

```
struct nlmsghdr *nh;    /* The nlmsghdr with payload to send */
struct sockaddr_nl sa;
struct iovec iov = { nh, nh->nlmsg_len };
struct msghdr msg;

msg = { &sa, sizeof(sa), &iov, 1, NULL, 0, 0 };
memset(&sa, 0, sizeof(sa));
sa.nl_family = AF_NETLINK;
nh->nlmsg_pid = 0;
nh->nlmsg_seq = ++sequence_number;
/* Request an ack from kernel by setting NLM_F_ACK */
nh->nlmsg_flags |= NLM_F_ACK;

sendmsg(fd, &msg, 0);
```

And the last example is about reading netlink message.

```
int len;
/* 8192 to avoid message truncation on platforms with
   page size > 4096 */
struct nlmsghdr buf[8192/sizeof(struct nlmsghdr)];
struct iovec iov = { buf, sizeof(buf) };
struct sockaddr_nl sa;
struct msghdr msg;
struct nlmsghdr *nh;

msg = { &sa, sizeof(sa), &iov, 1, NULL, 0, 0 };
len = recvmsg(fd, &msg, 0);

for (nh = (struct nlmsghdr *) buf; NLMSG_OK (nh, len);
     nh = NLMSG_NEXT (nh, len)) {
    /* The end of multipart message */
    if (nh->nlmsg_type == NLMSG_DONE)
        return;

    if (nh->nlmsg_type == NLMSG_ERROR)
        /* Do some error handling */
    ...

    /* Continue with parsing payload */
    ...
}
```

SEE ALSO

cmsg(3), *netlink*(3), *capabilities*(7), *rtnetlink*(7), *sock_diag*(7)

information about libnetlink

information about libnl

RFC 3549 "Linux Netlink as an IP Services Protocol"

NAME

network_namespaces − overview of Linux network namespaces

DESCRIPTION

Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the */proc/net* directory (which is a symbolic link to */proc/*pid*/net*), the */sys/class/net* directory, various files under */proc/sys/net*, port numbers (sockets), and so on. In addition, network namespaces isolate the UNIX domain abstract socket namespace (see *unix*(7)).

A physical network device can live in exactly one network namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the namespace of the parent of the process).

A virtual network (**veth**(4)) device pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a namespace is freed, the *veth*(4) devices that it contains are destroyed.

Use of network namespaces requires a kernel that is configured with the **CONFIG_NET_NS** option.

SEE ALSO

nsenter(1), *unshare*(1), *clone*(2), *veth*(4), *proc*(5), *sysfs*(5), *namespaces*(7), *user_namespaces*(7), *brctl*(8), *ip*(8), *ip-address*(8), *ip-link*(8), *ip-netns*(8), *iptables*(8), *ovs-vsctl*(8)

NAME

nptl − Native POSIX Threads Library

DESCRIPTION

NPTL (Native POSIX Threads Library) is the GNU C library POSIX threads implementation that is used on modern Linux systems.

NPTL and signals

NPTL makes internal use of the first two real-time signals (signal numbers 32 and 33). One of these signals is used to support thread cancelation and POSIX timers (see *timer_create*(2)); the other is used as part of a mechanism that ensures all threads in a process always have the same UIDs and GIDs, as required by POSIX. These signals cannot be used in applications.

To prevent accidental use of these signals in applications, which might interfere with the operation of the NPTL implementation, various glibc library functions and system call wrapper functions attempt to hide these signals from applications, as follows:

- **SIGRTMIN** is defined with the value 34 (rather than 32).
- The *sigwaitinfo*(2), *sigtimedwait*(2), and *sigwait*(3) interfaces silently ignore requests to wait for these two signals if they are specified in the signal set argument of these calls.
- The *sigprocmask*(2) and *pthread_sigmask*(3) interfaces silently ignore attempts to block these two signals.
- The *sigaction*(2), *pthread_kill*(3), and *pthread_sigqueue*(3) interfaces fail with the error **EINVAL** (indicating an invalid signal number) if these signals are specified.
- *sigfillset*(3) does not include these two signals when it creates a full signal set.

NPTL and process credential changes

At the Linux kernel level, credentials (user and group IDs) are a per-thread attribute. However, POSIX requires that all of the POSIX threads in a process have the same credentials. To accommodate this requirement, the NPTL implementation wraps all of the system calls that change process credentials with functions that, in addition to invoking the underlying system call, arrange for all other threads in the process to also change their credentials.

The implementation of each of these system calls involves the use of a real-time signal that is sent (using *tgkill*(2)) to each of the other threads that must change its credentials. Before sending these signals, the thread that is changing credentials saves the new credential(s) and records the system call being employed in a global buffer. A signal handler in the receiving thread(s) fetches this information and then uses the same system call to change its credentials.

Wrapper functions employing this technique are provided for *setgid*(2), *setuid*(2), *setegid*(2), *seteuid*(2), *setregid*(2), *setreuid*(2), *setresgid*(2), *setresuid*(2), and *setgroups*(2).

STANDARDS

For details of the conformance of NPTL to the POSIX standard, see *pthreads*(7).

NOTES

POSIX says that any thread in any process with access to the memory containing a process-shared (**PTHREAD_PROCESS_SHARED**) mutex can operate on that mutex. However, on 64-bit x86 systems, the mutex definition for x86-64 is incompatible with the mutex definition for i386, meaning that 32-bit and 64-bit binaries can’t share mutexes on x86-64 systems.

SEE ALSO

credentials(7), *pthreads*(7), *signal*(7), *standards*(7)

NAME

numa − overview of Non-Uniform Memory Architecture

DESCRIPTION

Non-Uniform Memory Access (NUMA) refers to multiprocessor systems whose memory is divided into multiple memory nodes. The access time of a memory node depends on the relative locations of the accessing CPU and the accessed node. (This contrasts with a symmetric multiprocessor system, where the access time for all of the memory is the same for all CPUs.) Normally, each CPU on a NUMA system has a local memory node whose contents can be accessed faster than the memory in the node local to another CPU or the memory on a bus shared by all CPUs.

NUMA system calls

The Linux kernel implements the following NUMA-related system calls: *get_mempolicy*(2), *mbind*(2), *migrate_pages*(2), *move_pages*(2), and *set_mempolicy*(2). However, applications should normally use the interface provided by *libnuma*; see "Library Support" below.

*/proc/*pid*/numa_maps* (since Linux 2.6.14)

This file displays information about a process's NUMA memory policy and allocation.

Each line contains information about a memory range used by the process, displaying—among other information—the effective memory policy for that memory range and on which nodes the pages have been allocated.

numa_maps is a read-only file. When */proc/*pid*/numa_maps* is read, the kernel will scan the virtual address space of the process and report how memory is used. One line is displayed for each unique memory range of the process.

The first field of each line shows the starting address of the memory range. This field allows a correlation with the contents of the */proc/*pid*/maps* file, which contains the end address of the range and other information, such as the access permissions and sharing.

The second field shows the memory policy currently in effect for the memory range. Note that the effective policy is not necessarily the policy installed by the process for that memory range. Specifically, if the process installed a "default" policy for that range, the effective policy for that range will be the process policy, which may or may not be "default".

The rest of the line contains information about the pages allocated in the memory range, as follows:

N<*node*>=<*nr_pages*>
: The number of pages allocated on <*node*>. <*nr_pages*> includes only pages currently mapped by the process. Page migration and memory reclaim may have temporarily unmapped pages associated with this memory range. These pages may show up again only after the process has attempted to reference them. If the memory range represents a shared memory area or file mapping, other processes may currently have additional pages mapped in a corresponding memory range.

file=<*filename*>
: The file backing the memory range. If the file is mapped as private, write accesses may have generated COW (Copy-On-Write) pages in this memory range. These pages are displayed as anonymous pages.

heap
: Memory range is used for the heap.

stack
: Memory range is used for the stack.

huge
: Huge memory range. The page counts shown are huge pages and not regular sized pages.

anon=<*pages*>
: The number of anonymous page in the range.

dirty=<*pages*>
: Number of dirty pages.

mapped=<*pages*>
: Total number of mapped pages, if different from *dirty* and *anon* pages.

mapmax=<*count*>
: Maximum mapcount (number of processes mapping a single page) encountered during the scan. This may be used as an indicator of the degree of sharing occurring in a given memory

range.

swapcache=<count>
Number of pages that have an associated entry on a swap device.

active=<pages>
The number of pages on the active list. This field is shown only if different from the number of pages in this range. This means that some inactive pages exist in the memory range that may be removed from memory by the swapper soon.

writeback=<pages>
Number of pages that are currently being written out to disk.

STANDARDS

None.

NOTES

The Linux NUMA system calls and */proc* interface are available only if the kernel was configured and built with the **CONFIG_NUMA** option.

Library support

Link with *−lnuma* to get the system call definitions. *libnuma* and the required *<numaif.h>* header are available in the *numactl* package.

However, applications should not use these system calls directly. Instead, the higher level interface provided by the *numa*(3) functions in the *numactl* package is recommended. The *numactl* package is available at . The package is also included in some Linux distributions. Some distributions include the development library and header in the separate *numactl−devel* package.

SEE ALSO

get_mempolicy(2), *mbind*(2), *move_pages*(2), *set_mempolicy*(2), *numa*(3), *cpuset*(7), *numactl*(8)

NAME

operator − C operator precedence and order of evaluation

DESCRIPTION

This manual page lists C operators and their precedence in evaluation.

Operator	**Associativity**	**Notes**
[] () . −> ++ −−	left to right	[1]
++ −− & * + − ~ ! sizeof	right to left	[2]
(type)	right to left	
* / %	left to right	
+ −	left to right	
<< >>	left to right	
< > <= >=	left to right	
== !=	left to right	
&	left to right	
^	left to right	
|	left to right	
&&	left to right	
||	left to right	
?:	right to left	
= *= /= %= += −= <<= >>= &= ^= |=	right to left	
,	left to right	

The following notes provide further information to the above table:

[1] The ++ and −− operators at this precedence level are the postfix flavors of the operators.

[2] The ++ and −− operators at this precedence level are the prefix flavors of the operators.

NAME

packet − packet interface on device level

SYNOPSIS

#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */

packet_socket = socket(AF_PACKET, int *socket_type***, int** *protocol***);**

DESCRIPTION

Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.

The *socket_type* is either **SOCK_RAW** for raw packets including the link-level header or **SOCK_DGRAM** for cooked packets with the link-level header removed. The link-level header information is available in a common format in a *sockaddr_ll* structure. *protocol* is the IEEE 802.3 protocol number in network byte order. See the <*linux/if_ether.h*> include file for a list of allowed protocols. When protocol is set to **htons(ETH_P_ALL)**, then all protocols are received. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel. If *protocol* is set to zero, no packets are received. *bind*(2) can optionally be called with a nonzero *sll_protocol* to start receiving packets for the protocols specified.

In order to create a packet socket, a process must have the **CAP_NET_RAW** capability in the user namespace that governs its network namespace.

SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. When receiving a packet, the address is still parsed and passed in a standard *sockaddr_ll* address structure. When transmitting a packet, the user-supplied buffer should contain the physical-layer header. That packet is then queued unmodified to the network driver of the interface defined by the destination address. Some device drivers always add other headers. **SOCK_RAW** is similar to but not compatible with the obsolete **AF_INET/SOCK_PACKET** of Linux 2.0.

SOCK_DGRAM operates on a slightly higher level. The physical header is removed before the packet is passed to the user. Packets sent through a **SOCK_DGRAM** packet socket get a suitable physical-layer header based on the information in the *sockaddr_ll* destination address before they are queued.

By default, all packets of the specified protocol type are passed to a packet socket. To get packets only from a specific interface use *bind*(2) specifying an address in a *struct sockaddr_ll* to bind the packet socket to an interface. Fields used for binding are *sll_family* (should be **AF_PACKET**), *sll_protocol*, and *sll_ifindex*.

The *connect*(2) operation is not supported on packet sockets.

When the **MSG_TRUNC** flag is passed to *recvmsg*(2), *recv*(2), or *recvfrom*(2), the real length of the packet on the wire is always returned, even when it is longer than the buffer.

Address types

The *sockaddr_ll* structure is a device-independent physical-layer address.

```
struct sockaddr_ll {
    unsigned short sll_family;   /* Always AF_PACKET */
    unsigned short sll_protocol; /* Physical-layer protocol */
    int            sll_ifindex;  /* Interface number */
    unsigned short sll_hatype;   /* ARP hardware type */
    unsigned char  sll_pkttype;  /* Packet type */
    unsigned char  sll_halen;    /* Length of address */
    unsigned char  sll_addr[8];  /* Physical-layer address */
};
```

The fields of this structure are as follows:

sll_protocol
: is the standard ethernet protocol type in network byte order as defined in the <*linux/if_ether.h*> include file. It defaults to the socket's protocol.

sll_ifindex
: is the interface index of the interface (see *netdevice*(7)); 0 matches any interface (only permitted for binding). *sll_hatype* is an ARP type as defined in the <*linux/if_arp.h*> include file.

sll_pkttype
: contains the packet type. Valid types are **PACKET_HOST** for a packet addressed to the local host, **PACKET_BROADCAST** for a physical-layer broadcast packet, **PACKET_MULTICAST** for a packet sent to a physical-layer multicast address, **PACKET_OTHERHOST** for a packet to some other host that has been caught by a device driver in promiscuous mode, and **PACKET_OUTGOING** for a packet originating from the local host that is looped back to a packet socket. These types make sense only for receiving.

sll_addr
sll_halen
: contain the physical-layer (e.g., IEEE 802.3) address and its length. The exact interpretation depends on the device.

When you send packets, it is enough to specify *sll_family*, *sll_addr*, *sll_halen*, *sll_ifindex*, and *sll_protocol*. The other fields should be 0. *sll_hatype* and *sll_pkttype* are set on received packets for your information.

Socket options

Packet socket options are configured by calling *setsockopt*(2) with level **SOL_PACKET**.

PACKET_ADD_MEMBERSHIP
PACKET_DROP_MEMBERSHIP
: Packet sockets can be used to configure physical-layer multicasting and promiscuous mode. **PACKET_ADD_MEMBERSHIP** adds a binding and **PACKET_DROP_MEMBERSHIP** drops it. They both expect a *packet_mreq* structure as argument:

```
struct packet_mreq {
    int            mr_ifindex;    /* interface index */
    unsigned short mr_type;       /* action */
    unsigned short mr_alen;       /* address length */
    unsigned char  mr_address[8]; /* physical-layer address */
};
```

mr_ifindex contains the interface index for the interface whose status should be changed. The *mr_type* field specifies which action to perform. **PACKET_MR_PROMISC** enables receiving all packets on a shared medium (often known as "promiscuous mode"), **PACKET_MR_MULTICAST** binds the socket to the physical-layer multicast group specified in *mr_address* and *mr_alen*, and **PACKET_MR_ALLMULTI** sets the socket up to receive all multicast packets arriving at the interface.

In addition, the traditional ioctls **SIOCSIFFLAGS**, **SIOCADDMULTI**, **SIOCDELMULTI** can be used for the same purpose.

PACKET_AUXDATA (since Linux 2.6.21)
: If this binary option is enabled, the packet socket passes a metadata structure along with each packet in the *recvmsg*(2) control field. The structure can be read with *cmsg*(3). It is defined as

```
struct tpacket_auxdata {
    __u32 tp_status;
    __u32 tp_len;      /* packet length */
    __u32 tp_snaplen;  /* captured length */
    __u16 tp_mac;
    __u16 tp_net;
    __u16 tp_vlan_tci;
    __u16 tp_vlan_tpid; /* Since Linux 3.14; earlier, these
                           were unused padding bytes */
};
```

PACKET_FANOUT (since Linux 3.1)
: To scale processing across threads, packet sockets can form a fanout group. In this mode, each matching packet is enqueued onto only one socket in the group. A socket joins a fanout

group by calling *setsockopt*(2) with level **SOL_PACKET** and option **PACKET_FANOUT**. Each network namespace can have up to 65536 independent groups. A socket selects a group by encoding the ID in the first 16 bits of the integer option value. The first packet socket to join a group implicitly creates it. To successfully join an existing group, subsequent packet sockets must have the same protocol, device settings, fanout mode, and flags (see below). Packet sockets can leave a fanout group only by closing the socket. The group is deleted when the last socket is closed.

Fanout supports multiple algorithms to spread traffic between sockets, as follows:

- The default mode, **PACKET_FANOUT_HASH**, sends packets from the same flow to the same socket to maintain per-flow ordering. For each packet, it chooses a socket by taking the packet flow hash modulo the number of sockets in the group, where a flow hash is a hash over network-layer address and optional transport-layer port fields.
- The load-balance mode **PACKET_FANOUT_LB** implements a round-robin algorithm.
- **PACKET_FANOUT_CPU** selects the socket based on the CPU that the packet arrived on.
- **PACKET_FANOUT_ROLLOVER** processes all data on a single socket, moving to the next when one becomes backlogged.
- **PACKET_FANOUT_RND** selects the socket using a pseudo-random number generator.
- **PACKET_FANOUT_QM** (available since Linux 3.14) selects the socket using the recorded queue_mapping of the received skb.

Fanout modes can take additional options. IP fragmentation causes packets from the same flow to have different flow hashes. The flag **PACKET_FANOUT_FLAG_DEFRAG**, if set, causes packets to be defragmented before fanout is applied, to preserve order even in this case. Fanout mode and options are communicated in the second 16 bits of the integer option value. The flag **PACKET_FANOUT_FLAG_ROLLOVER** enables the roll over mechanism as a backup strategy: if the original fanout algorithm selects a backlogged socket, the packet rolls over to the next available one.

PACKET_LOSS (with **PACKET_TX_RING**)
When a malformed packet is encountered on a transmit ring, the default is to reset its *tp_status* to **TP_STATUS_WRONG_FORMAT** and abort the transmission immediately. The malformed packet blocks itself and subsequently enqueued packets from being sent. The format error must be fixed, the associated *tp_status* reset to **TP_STATUS_SEND_REQUEST**, and the transmission process restarted via *send*(2). However, if **PACKET_LOSS** is set, any malformed packet will be skipped, its *tp_status* reset to **TP_STATUS_AVAILABLE**, and the transmission process continued.

PACKET_RESERVE (with **PACKET_RX_RING**)
By default, a packet receive ring writes packets immediately following the metadata structure and alignment padding. This integer option reserves additional headroom.

PACKET_RX_RING
Create a memory-mapped ring buffer for asynchronous packet reception. The packet socket reserves a contiguous region of application address space, lays it out into an array of packet slots and copies packets (up to *tp_snaplen*) into subsequent slots. Each packet is preceded by a metadata structure similar to *tpacket_auxdata*. The protocol fields encode the offset to the data from the start of the metadata header. *tp_net* stores the offset to the network layer. If the packet socket is of type **SOCK_DGRAM**, then *tp_mac* is the same. If it is of type **SOCK_RAW**, then that field stores the offset to the link-layer frame. Packet socket and application communicate the head and tail of the ring through the *tp_status* field. The packet socket owns all slots with *tp_status* equal to **TP_STATUS_KERNEL**. After filling a slot, it changes the status of the slot to transfer ownership to the application. During normal operation, the new *tp_status* value has at least the **TP_STATUS_USER** bit set to signal that a received packet has been stored. When the application has finished processing a packet, it transfers ownership of the slot back to the socket by setting *tp_status* equal to **TP_STATUS_KERNEL**.

Packet sockets implement multiple variants of the packet ring. The implementation details are described in *Documentation/networking/packet_mmap.rst* in the Linux kernel source tree.

PACKET_STATISTICS
Retrieve packet socket statistics in the form of a structure

```
struct tpacket_stats {
    unsigned int tp_packets;  /* Total packet count */
    unsigned int tp_drops;    /* Dropped packet count */
};
```

Receiving statistics resets the internal counters. The statistics structure differs when using a ring of variant **TPACKET_V3**.

PACKET_TIMESTAMP (with **PACKET_RX_RING**; since Linux 2.6.36)
The packet receive ring always stores a timestamp in the metadata header. By default, this is a software generated timestamp generated when the packet is copied into the ring. This integer option selects the type of timestamp. Besides the default, it support the two hardware formats described in *Documentation/networking/timestamping.rst* in the Linux kernel source tree.

PACKET_TX_RING (since Linux 2.6.31)
Create a memory-mapped ring buffer for packet transmission. This option is similar to **PACKET_RX_RING** and takes the same arguments. The application writes packets into slots with *tp_status* equal to **TP_STATUS_AVAILABLE** and schedules them for transmission by changing *tp_status* to **TP_STATUS_SEND_REQUEST**. When packets are ready to be transmitted, the application calls *send*(2) or a variant thereof. The *buf* and *len* fields of this call are ignored. If an address is passed using *sendto*(2) or *sendmsg*(2), then that overrides the socket default. On successful transmission, the socket resets *tp_status* to **TP_STATUS_AVAILABLE**. It immediately aborts the transmission on error unless **PACKET_LOSS** is set.

PACKET_VERSION (with **PACKET_RX_RING**; since Linux 2.6.27)
By default, **PACKET_RX_RING** creates a packet receive ring of variant **TPACKET_V1**. To create another variant, configure the desired variant by setting this integer option before creating the ring.

PACKET_QDISC_BYPASS (since Linux 3.14)
By default, packets sent through packet sockets pass through the kernel's qdisc (traffic control) layer, which is fine for the vast majority of use cases. For traffic generator appliances using packet sockets that intend to brute-force flood the network—for example, to test devices under load in a similar fashion to pktgen—this layer can be bypassed by setting this integer option to 1. A side effect is that packet buffering in the qdisc layer is avoided, which will lead to increased drops when network device transmit queues are busy; therefore, use at your own risk.

Ioctls

SIOCGSTAMP can be used to receive the timestamp of the last received packet. Argument is a *struct timeval* variable.

In addition, all standard ioctls defined in *netdevice*(7) and *socket*(7) are valid on packet sockets.

Error handling

Packet sockets do no error handling other than errors occurred while passing the packet to the device driver. They don't have the concept of a pending error.

ERRORS

EADDRNOTAVAIL
Unknown multicast group address passed.

EFAULT
User passed invalid memory address.

EINVAL
Invalid argument.

EMSGSIZE
Packet is bigger than interface MTU.

ENETDOWN
Interface is not up.

ENOBUFS
Not enough memory to allocate the packet.

ENODEV
Unknown device name or interface index specified in interface address.

ENOENT
No packet received.

ENOTCONN
No interface address passed.

ENXIO
Interface address contained an invalid interface index.

EPERM
User has insufficient privileges to carry out this operation.

In addition, other errors may be generated by the low-level driver.

VERSIONS

AF_PACKET is a new feature in Linux 2.2. Earlier Linux versions supported only **SOCK_PACKET**.

NOTES

For portable programs it is suggested to use **AF_PACKET** via *pcap*(3); although this covers only a subset of the **AF_PACKET** features.

The **SOCK_DGRAM** packet sockets make no attempt to create or parse the IEEE 802.2 LLC header for a IEEE 802.3 frame. When **ETH_P_802_3** is specified as protocol for sending the kernel creates the 802.3 frame and fills out the length field; the user has to supply the LLC header to get a fully conforming packet. Incoming 802.3 packets are not multiplexed on the DSAP/SSAP protocol fields; instead they are supplied to the user as protocol **ETH_P_802_2** with the LLC header prefixed. It is thus not possible to bind to **ETH_P_802_3**; bind to **ETH_P_802_2** instead and do the protocol multiplex yourself. The default for sending is the standard Ethernet DIX encapsulation with the protocol filled in.

Packet sockets are not subject to the input or output firewall chains.

Compatibility

In Linux 2.0, the only way to get a packet socket was with the call:

```
socket(AF_INET, SOCK_PACKET, protocol)
```

This is still supported, but deprecated and strongly discouraged. The main difference between the two methods is that **SOCK_PACKET** uses the old *struct sockaddr_pkt* to specify an interface, which doesn't provide physical-layer independence.

```
struct sockaddr_pkt {
    unsigned short spkt_family;
    unsigned char  spkt_device[14];
    unsigned short spkt_protocol;
};
```

spkt_family contains the device type, *spkt_protocol* is the IEEE 802.3 protocol type as defined in <*sys/if_ether.h*> and *spkt_device* is the device name as a null-terminated string, for example, eth0.

This structure is obsolete and should not be used in new code.

BUGS

LLC header handling

The IEEE 802.2/803.3 LLC handling could be considered as a bug.

MSG_TRUNC issues

The **MSG_TRUNC** *recvmsg*(2) extension is an ugly hack and should be replaced by a control message. There is currently no way to get the original destination address of packets via **SOCK_DGRAM**.

spkt_device device name truncation

The *spkt_device* field of *sockaddr_pkt* has a size of 14 bytes, which is less than the constant **IFNAMSIZ** defined in <*net/if.h*> which is 16 bytes and describes the system limit for a network interface

name. This means the names of network devices longer than 14 bytes will be truncated to fit into *spkt_device*. All these lengths include the terminating null byte ('\0').

Issues from this with old code typically show up with very long interface names used by the **Predictable Network Interface Names** feature enabled by default in many modern Linux distributions.

The preferred solution is to rewrite code to avoid **SOCK_PACKET**. Possible user solutions are to disable **Predictable Network Interface Names** or to rename the interface to a name of at most 13 bytes, for example using the *ip*(8) tool.

Documentation issues

Socket filters are not documented.

SEE ALSO

socket(2), *pcap*(3), *capabilities*(7), *ip*(7), *raw*(7), *socket*(7), *ip*(8),

RFC 894 for the standard IP Ethernet encapsulation. RFC 1700 for the IEEE 802.3 IP encapsulation.

The <*linux/if_ether.h*> include file for physical-layer protocols.

The Linux kernel source tree. *Documentation/networking/filter.rst* describes how to apply Berkeley Packet Filters to packet sockets. *tools/testing/selftests/net/psock_tpacket.c* contains example source code for all available versions of **PACKET_RX_RING** and **PACKET_TX_RING**.

NAME

path_resolution − how a pathname is resolved to a file

DESCRIPTION

Some UNIX/Linux system calls have as parameter one or more filenames. A filename (or pathname) is resolved as follows.

Step 1: start of the resolution process

If the pathname starts with the '/' character, the starting lookup directory is the root directory of the calling process. A process inherits its root directory from its parent. Usually this will be the root directory of the file hierarchy. A process may get a different root directory by use of the *chroot*(2) system call, or may temporarily use a different root directory by using *openat2*(2) with the **RESOLVE_IN_ROOT** flag set.

A process may get an entirely private mount namespace in case it—or one of its ancestors—was started by an invocation of the *clone*(2) system call that had the **CLONE_NEWNS** flag set. This handles the '/' part of the pathname.

If the pathname does not start with the '/' character, the starting lookup directory of the resolution process is the current working directory of the process — or in the case of *openat*(2)-style system calls, the *dfd* argument (or the current working directory if **AT_FDCWD** is passed as the *dfd* argument). The current working directory is inherited from the parent, and can be changed by use of the *chdir*(2) system call.

Pathnames starting with a '/' character are called absolute pathnames. Pathnames not starting with a '/' are called relative pathnames.

Step 2: walk along the path

Set the current lookup directory to the starting lookup directory. Now, for each nonfinal component of the pathname, where a component is a substring delimited by '/' characters, this component is looked up in the current lookup directory.

If the process does not have search permission on the current lookup directory, an **EACCES** error is returned ("Permission denied").

If the component is not found, an **ENOENT** error is returned ("No such file or directory").

If the component is found, but is neither a directory nor a symbolic link, an **ENOTDIR** error is returned ("Not a directory").

If the component is found and is a directory, we set the current lookup directory to that directory, and go to the next component.

If the component is found and is a symbolic link, we first resolve this symbolic link (with the current lookup directory as starting lookup directory). Upon error, that error is returned. If the result is not a directory, an **ENOTDIR** error is returned. If the resolution of the symbolic link is successful and returns a directory, we set the current lookup directory to that directory, and go to the next component. Note that the resolution process here can involve recursion if the prefix (’dirname’) component of a pathname contains a filename that is a symbolic link that resolves to a directory (where the prefix component of that directory may contain a symbolic link, and so on). In order to protect the kernel against stack overflow, and also to protect against denial of service, there are limits on the maximum recursion depth, and on the maximum number of symbolic links followed. An **ELOOP** error is returned when the maximum is exceeded ("Too many levels of symbolic links").

As currently implemented on Linux, the maximum number of symbolic links that will be followed while resolving a pathname is 40. Before Linux 2.6.18, the limit on the recursion depth was 5. Starting with Linux 2.6.18, this limit was raised to 8. In Linux 4.2, the kernel’s pathname-resolution code was reworked to eliminate the use of recursion, so that the only limit that remains is the maximum of 40 resolutions for the entire pathname.

The resolution of symbolic links during this stage can be blocked by using *openat2*(2), with the **RESOLVE_NO_SYMLINKS** flag set.

Step 3: find the final entry

The lookup of the final component of the pathname goes just like that of all other components, as described in the previous step, with two differences: (i) the final component need not be a directory (at least as far as the path resolution process is concerned—it may have to be a directory, or a nondirectory,

because of the requirements of the specific system call), and (ii) it is not necessarily an error if the component is not found—maybe we are just creating it. The details on the treatment of the final entry are described in the manual pages of the specific system calls.

. and ..

By convention, every directory has the entries "." and "..", which refer to the directory itself and to its parent directory, respectively.

The path resolution process will assume that these entries have their conventional meanings, regardless of whether they are actually present in the physical filesystem.

One cannot walk up past the root: "/.." is the same as "/".

Mount points

After a *mount dev path* command, the pathname "path" refers to the root of the filesystem hierarchy on the device "dev", and no longer to whatever it referred to earlier.

One can walk out of a mounted filesystem: "path/.." refers to the parent directory of "path", outside of the filesystem hierarchy on "dev".

Traversal of mount points can be blocked by using *openat2*(2), with the **RESOLVE_NO_XDEV** flag set (though note that this also restricts bind mount traversal).

Trailing slashes

If a pathname ends in a '/', that forces resolution of the preceding component as in Step 2: the component preceding the slash either exists and resolves to a directory or it names a directory that is to be created immediately after the pathname is resolved. Otherwise, a trailing '/' is ignored.

Final symbolic link

If the last component of a pathname is a symbolic link, then it depends on the system call whether the file referred to will be the symbolic link or the result of path resolution on its contents. For example, the system call *lstat*(2) will operate on the symbolic link, while *stat*(2) operates on the file pointed to by the symbolic link.

Length limit

There is a maximum length for pathnames. If the pathname (or some intermediate pathname obtained while resolving symbolic links) is too long, an **ENAMETOOLONG** error is returned ("Filename too long").

Empty pathname

In the original UNIX, the empty pathname referred to the current directory. Nowadays POSIX decrees that an empty pathname must not be resolved successfully. Linux returns **ENOENT** in this case.

Permissions

The permission bits of a file consist of three groups of three bits; see *chmod*(1) and *stat*(2). The first group of three is used when the effective user ID of the calling process equals the owner ID of the file. The second group of three is used when the group ID of the file either equals the effective group ID of the calling process, or is one of the supplementary group IDs of the calling process (as set by *setgroups*(2)). When neither holds, the third group is used.

Of the three bits used, the first bit determines read permission, the second write permission, and the last execute permission in case of ordinary files, or search permission in case of directories.

Linux uses the fsuid instead of the effective user ID in permission checks. Ordinarily the fsuid will equal the effective user ID, but the fsuid can be changed by the system call *setfsuid*(2).

(Here "fsuid" stands for something like "filesystem user ID". The concept was required for the implementation of a user space NFS server at a time when processes could send a signal to a process with the same effective user ID. It is obsolete now. Nobody should use *setfsuid*(2).)

Similarly, Linux uses the fsgid ("filesystem group ID") instead of the effective group ID. See *setfsgid*(2).

Bypassing permission checks: superuser and capabilities

On a traditional UNIX system, the superuser (*root*, user ID 0) is all-powerful, and bypasses all permissions restrictions when accessing files.

On Linux, superuser privileges are divided into capabilities (see *capabilities*(7)). Two capabilities are relevant for file permissions checks: **CAP_DAC_OVERRIDE** and **CAP_DAC_READ_SEARCH**. (A

process has these capabilities if its fsuid is 0.)

The **CAP_DAC_OVERRIDE** capability overrides all permission checking, but grants execute permission only when at least one of the file's three execute permission bits is set.

The **CAP_DAC_READ_SEARCH** capability grants read and search permission on directories, and read permission on ordinary files.

SEE ALSO

readlink(2), *capabilities*(7), *credentials*(7), *symlink*(7)

NAME

persistent-keyring − per-user persistent keyring

DESCRIPTION

The persistent keyring is a keyring used to anchor keys on behalf of a user. Each UID the kernel deals with has its own persistent keyring that is shared between all threads owned by that UID. The persistent keyring has a name (description) of the form *_persistent.<UID>* where *<UID>* is the user ID of the corresponding user.

The persistent keyring may not be accessed directly, even by processes with the appropriate UID. Instead, it must first be linked to one of a process's keyrings, before that keyring can access the persistent keyring by virtue of its possessor permits. This linking is done with the *keyctl_get_persistent*(3) function.

If a persistent keyring does not exist when it is accessed by the *keyctl_get_persistent*(3) operation, it will be automatically created.

Each time the *keyctl_get_persistent*(3) operation is performed, the persistent keyring's expiration timer is reset to the value in:

```
/proc/sys/kernel/keys/persistent_keyring_expiry
```

Should the timeout be reached, the persistent keyring will be removed and everything it pins can then be garbage collected. The keyring will then be re-created on a subsequent call to *keyctl_get_persistent*(3)

The persistent keyring is not directly searched by *request_key*(2); it is searched only if it is linked into one of the keyrings that is searched by *request_key*(2).

The persistent keyring is independent of *clone*(2), *fork*(2), *vfork*(2), *execve*(2), and *_exit*(2). It persists until its expiration timer triggers, at which point it is garbage collected. This allows the persistent keyring to carry keys beyond the life of the kernel's record of the corresponding UID (the destruction of which results in the destruction of the *user-keyring*(7) and the *user-session-keyring*(7)). The persistent keyring can thus be used to hold authentication tokens for processes that run without user interaction, such as programs started by *cron*(8)

The persistent keyring is used to store UID-specific objects that themselves have limited lifetimes (e.g., kerberos tokens). If those tokens cease to be used (i.e., the persistent keyring is not accessed), then the timeout of the persistent keyring ensures that the corresponding objects are automatically discarded.

Special operations

The *keyutils* library provides the *keyctl_get_persistent*(3) function for manipulating persistent keyrings. (This function is an interface to the *keyctl*(2) **KEYCTL_GET_PERSISTENT** operation.) This operation allows the calling thread to get the persistent keyring corresponding to its own UID or, if the thread has the **CAP_SETUID** capability, the persistent keyring corresponding to some other UID in the same user namespace.

NOTES

Each user namespace owns a keyring called *.persistent_register* that contains links to all of the persistent keys in that namespace. (The *.persistent_register* keyring can be seen when reading the contents of the */proc/keys* file for the UID 0 in the namespace.) The *keyctl_get_persistent*(3) operation looks for a key with a name of the form *_persistent.*UID in that keyring, creates the key if it does not exist, and links it into the keyring.

SEE ALSO

keyctl(1), *keyctl*(3), *keyctl_get_persistent*(3), *keyrings*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7)

NAME

pid_namespaces − overview of Linux PID namespaces

DESCRIPTION

For an overview of namespaces, see *namespaces*(7).

PID namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.

PIDs in a new PID namespace start at 1, somewhat like a standalone system, and calls to *fork*(2), *vfork*(2), or *clone*(2) will produce processes with PIDs that are unique within the namespace.

Use of PID namespaces requires a kernel that is configured with the **CONFIG_PID_NS** option.

The namespace init process

The first process created in a new namespace (i.e., the process created using *clone*(2) with the **CLONE_NEWPID** flag, or the first child created by a process after a call to *unshare*(2) using the **CLONE_NEWPID** flag) has the PID 1, and is the "init" process for the namespace (see *init*(1)). This process becomes the parent of any child processes that are orphaned because a process that resides in this PID namespace terminated (see below for further details).

If the "init" process of a PID namespace terminates, the kernel terminates all of the processes in the namespace via a **SIGKILL** signal. This behavior reflects the fact that the "init" process is essential for the correct operation of a PID namespace. In this case, a subsequent *fork*(2) into this PID namespace fail with the error **ENOMEM**; it is not possible to create a new process in a PID namespace whose "init" process has terminated. Such scenarios can occur when, for example, a process uses an open file descriptor for a */proc/*pid*/ns/pid* file corresponding to a process that was in a namespace to *setns*(2) into that namespace after the "init" process has terminated. Another possible scenario can occur after a call to *unshare*(2): if the first child subsequently created by a *fork*(2) terminates, then subsequent calls to *fork*(2) fail with **ENOMEM**.

Only signals for which the "init" process has established a signal handler can be sent to the "init" process by other members of the PID namespace. This restriction applies even to privileged processes, and prevents other members of the PID namespace from accidentally killing the "init" process.

Likewise, a process in an ancestor namespace can—subject to the usual permission checks described in *kill*(2)—send signals to the "init" process of a child PID namespace only if the "init" process has established a handler for that signal. (Within the handler, the *siginfo_t si_pid* field described in *sigaction*(2) will be zero.) **SIGKILL** or **SIGSTOP** are treated exceptionally: these signals are forcibly delivered when sent from an ancestor PID namespace. Neither of these signals can be caught by the "init" process, and so will result in the usual actions associated with those signals (respectively, terminating and stopping the process).

Starting with Linux 3.4, the *reboot*(2) system call causes a signal to be sent to the namespace "init" process. See *reboot*(2) for more details.

Nesting PID namespaces

PID namespaces can be nested: each PID namespace has a parent, except for the initial ("root") PID namespace. The parent of a PID namespace is the PID namespace of the process that created the namespace using *clone*(2) or *unshare*(2). PID namespaces thus form a tree, with all namespaces ultimately tracing their ancestry to the root namespace. Since Linux 3.7, the kernel limits the maximum nesting depth for PID namespaces to 32.

A process is visible to other processes in its PID namespace, and to the processes in each direct ancestor PID namespace going back to the root PID namespace. In this context, "visible" means that one process can be the target of operations by another process using system calls that specify a process ID. Conversely, the processes in a child PID namespace can’t see processes in the parent and further removed ancestor namespaces. More succinctly: a process can see (e.g., send signals with *kill*(2), set nice values with *setpriority*(2), etc.) only processes contained in its own PID namespace and in descendants of that namespace.

A process has one process ID in each of the layers of the PID namespace hierarchy in which is visible, and walking back though each direct ancestor namespace through to the root PID namespace. System calls that operate on process IDs always operate using the process ID that is visible in the PID

namespace of the caller. A call to *getpid*(2) always returns the PID associated with the namespace in which the process was created.

Some processes in a PID namespace may have parents that are outside of the namespace. For example, the parent of the initial process in the namespace (i.e., the *init*(1) process with PID 1) is necessarily in another namespace. Likewise, the direct children of a process that uses *setns*(2) to cause its children to join a PID namespace are in a different PID namespace from the caller of *setns*(2). Calls to *getppid*(2) for such processes return 0.

While processes may freely descend into child PID namespaces (e.g., using *setns*(2) with a PID namespace file descriptor), they may not move in the other direction. That is to say, processes may not enter any ancestor namespaces (parent, grandparent, etc.). Changing PID namespaces is a one-way operation.

The **NS_GET_PARENT** *ioctl*(2) operation can be used to discover the parental relationship between PID namespaces; see *ioctl_ns*(2).

setns(2) and unshare(2) semantics

Calls to *setns*(2) that specify a PID namespace file descriptor and calls to *unshare*(2) with the **CLONE_NEWPID** flag cause children subsequently created by the caller to be placed in a different PID namespace from the caller. (Since Linux 4.12, that PID namespace is shown via the */proc/*pid*/ns/pid_for_children* file, as described in *namespaces*(7).) These calls do not, however, change the PID namespace of the calling process, because doing so would change the caller's idea of its own PID (as reported by *getpid*()), which would break many applications and libraries.

To put things another way: a process's PID namespace membership is determined when the process is created and cannot be changed thereafter. Among other things, this means that the parental relationship between processes mirrors the parental relationship between PID namespaces: the parent of a process is either in the same namespace or resides in the immediate parent PID namespace.

A process may call *unshare*(2) with the **CLONE_NEWPID** flag only once. After it has performed this operation, its */proc/*pid*/ns/pid_for_children* symbolic link will be empty until the first child is created in the namespace.

Adoption of orphaned children

When a child process becomes orphaned, it is reparented to the "init" process in the PID namespace of its parent (unless one of the nearer ancestors of the parent employed the *prctl*(2) **PR_SET_CHILD_SUBREAPER** command to mark itself as the reaper of orphaned descendant processes). Note that because of the *setns*(2) and *unshare*(2) semantics described above, this may be the "init" process in the PID namespace that is the *parent* of the child's PID namespace, rather than the "init" process in the child's own PID namespace.

Compatibility of CLONE_NEWPID with other CLONE_* flags

In current versions of Linux, **CLONE_NEWPID** can't be combined with **CLONE_THREAD**. Threads are required to be in the same PID namespace such that the threads in a process can send signals to each other. Similarly, it must be possible to see all of the threads of a process in the *proc*(5) filesystem. Additionally, if two threads were in different PID namespaces, the process ID of the process sending a signal could not be meaningfully encoded when a signal is sent (see the description of the *siginfo_t* type in *sigaction*(2)). Since this is computed when a signal is enqueued, a signal queue shared by processes in multiple PID namespaces would defeat that.

In earlier versions of Linux, **CLONE_NEWPID** was additionally disallowed (failing with the error **EINVAL**) in combination with **CLONE_SIGHAND** (before Linux 4.3) as well as **CLONE_VM** (before Linux 3.12). The changes that lifted these restrictions have also been ported to earlier stable kernels.

/proc and PID namespaces

A */proc* filesystem shows (in the */proc/*pid directories) only processes visible in the PID namespace of the process that performed the mount, even if the */proc* filesystem is viewed from processes in other namespaces.

After creating a new PID namespace, it is useful for the child to change its root directory and mount a new procfs instance at */proc* so that tools such as *ps*(1) work correctly. If a new mount namespace is simultaneously created by including **CLONE_NEWNS** in the *flags* argument of *clone*(2) or *unshare*(2), then it isn't necessary to change the root directory: a new procfs instance can be mounted

directly over */proc*.

From a shell, the command to mount */proc* is:

```
$ mount -t proc proc /proc
```

Calling *readlink*(2) on the path */proc/self* yields the process ID of the caller in the PID namespace of the procfs mount (i.e., the PID namespace of the process that mounted the procfs). This can be useful for introspection purposes, when a process wants to discover its PID in other namespaces.

/proc files

/proc/sys/kernel/ns_last_pid (since Linux 3.3)

This file (which is virtualized per PID namespace) displays the last PID that was allocated in this PID namespace. When the next PID is allocated, the kernel will search for the lowest unallocated PID that is greater than this value, and when this file is subsequently read it will show that PID.

This file is writable by a process that has the **CAP_SYS_ADMIN** or (since Linux 5.9) **CAP_CHECKPOINT_RESTORE** capability inside the user namespace that owns the PID namespace. This makes it possible to determine the PID that is allocated to the next process that is created inside this PID namespace.

Miscellaneous

When a process ID is passed over a UNIX domain socket to a process in a different PID namespace (see the description of **SCM_CREDENTIALS** in *unix*(7)), it is translated into the corresponding PID value in the receiving process's PID namespace.

STANDARDS

Linux.

EXAMPLES

See *user_namespaces*(7).

SEE ALSO

clone(2), *reboot*(2), *setns*(2), *unshare*(2), *proc*(5), *capabilities*(7), *credentials*(7), *mount_namespaces*(7), *namespaces*(7), *user_namespaces*(7), *switch_root*(8)

NAME

pipe − overview of pipes and FIFOs

DESCRIPTION

Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel. A pipe has a *read end* and a *write end*. Data written to the write end of a pipe can be read from the read end of the pipe.

A pipe is created using *pipe*(2), which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes; see *pipe*(2) for an example.

A FIFO (short for First In First Out) has a name within the filesystem (created using *mkfifo*(3)), and is opened using *open*(2). Any process may open a FIFO, assuming the file permissions allow it. The read end is opened using the **O_RDONLY** flag; the write end is opened using the **O_WRONLY** flag. See *fifo*(7) for further details. *Note*: although FIFOs have a pathname in the filesystem, I/O on FIFOs does not involve operations on the underlying device (if there is one).

I/O on pipes and FIFOs

The only difference between pipes and FIFOs is the manner in which they are created and opened. Once these tasks have been accomplished, I/O on pipes and FIFOs has exactly the same semantics.

If a process attempts to read from an empty pipe, then *read*(2) will block until data is available. If a process attempts to write to a full pipe (see below), then *write*(2) blocks until sufficient data has been read from the pipe to allow the write to complete.

Nonblocking I/O is possible by using the *fcntl*(2) **F_SETFL** operation to enable the **O_NONBLOCK** open file status flag or by opening a *fifo*(7) with **O_NONBLOCK**. If any process has the pipe open for writing, reads fail with **EAGAIN**; otherwise—with no potential writers—reads succeed and return empty.

The communication channel provided by a pipe is a *byte stream*: there is no concept of message boundaries.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to *read*(2) from the pipe will see end-of-file (**read**(2) will return 0). If all file descriptors referring to the read end of a pipe have been closed, then a *write*(2) will cause a **SIGPIPE** signal to be generated for the calling process. If the calling process is ignoring this signal, then *write*(2) fails with the error **EPIPE**. An application that uses *pipe*(2) and *fork*(2) should use suitable *close*(2) calls to close unnecessary duplicate file descriptors; this ensures that end-of-file and **SIGPIPE**/**EPIPE** are delivered when appropriate.

It is not possible to apply *lseek*(2) to a pipe.

Pipe capacity

A pipe has a limited capacity. If the pipe is full, then a *write*(2) will block or fail, depending on whether the **O_NONBLOCK** flag is set (see below). Different implementations have different limits for the pipe capacity. Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.

Before Linux 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on i386). Since Linux 2.6.11, the pipe capacity is 16 pages (i.e., 65,536 bytes in a system with a page size of 4096 bytes). Since Linux 2.6.35, the default pipe capacity is 16 pages, but the capacity can be queried and set using the *fcntl*(2) **F_GETPIPE_SZ** and **F_SETPIPE_SZ** operations. See *fcntl*(2) for more information.

The following *ioctl*(2) operation, which can be applied to a file descriptor that refers to either end of a pipe, places a count of the number of unread bytes in the pipe in the *int* buffer pointed to by the final argument of the call:

```
ioctl(fd, FIONREAD, &nbytes);
```

The **FIONREAD** operation is not specified in any standard, but is provided on many implementations.

/proc files

On Linux, the following files control how much memory can be used for pipes:

/proc/sys/fs/pipe-max-pages (only in Linux 2.6.34)
An upper limit, in pages, on the capacity that an unprivileged user (one without the **CAP_SYS_RESOURCE** capability) can set for a pipe.

The default value for this limit is 16 times the default pipe capacity (see above); the lower limit is two pages.

This interface was removed in Linux 2.6.35, in favor of */proc/sys/fs/pipe-max-size*.

/proc/sys/fs/pipe-max-size (since Linux 2.6.35)
The maximum size (in bytes) of individual pipes that can be set by users without the **CAP_SYS_RESOURCE** capability. The value assigned to this file may be rounded upward, to reflect the value actually employed for a convenient implementation. To determine the rounded-up value, display the contents of this file after assigning a value to it.

The default value for this file is 1048576 (1 MiB). The minimum value that can be assigned to this file is the system page size. Attempts to set a limit less than the page size cause *write*(2) to fail with the error **EINVAL**.

Since Linux 4.9, the value on this file also acts as a ceiling on the default capacity of a new pipe or newly opened FIFO.

/proc/sys/fs/pipe-user-pages-hard (since Linux 4.5)
The hard limit on the total size (in pages) of all pipes created or set by a single unprivileged user (i.e., one with neither the **CAP_SYS_RESOURCE** nor the **CAP_SYS_ADMIN** capability). So long as the total number of pages allocated to pipe buffers for this user is at this limit, attempts to create new pipes will be denied, and attempts to increase a pipe's capacity will be denied.

When the value of this limit is zero (which is the default), no hard limit is applied.

/proc/sys/fs/pipe-user-pages-soft (since Linux 4.5)
The soft limit on the total size (in pages) of all pipes created or set by a single unprivileged user (i.e., one with neither the **CAP_SYS_RESOURCE** nor the **CAP_SYS_ADMIN** capability). So long as the total number of pages allocated to pipe buffers for this user is at this limit, individual pipes created by a user will be limited to one page, and attempts to increase a pipe's capacity will be denied.

When the value of this limit is zero, no soft limit is applied. The default value for this file is 16384, which permits creating up to 1024 pipes with the default capacity.

Before Linux 4.9, some bugs affected the handling of the *pipe-user-pages-soft* and *pipe-user-pages-hard* limits; see BUGS.

PIPE_BUF
POSIX.1 says that writes of less than **PIPE_BUF** bytes must be atomic: the output data is written to the pipe as a contiguous sequence. Writes of more than **PIPE_BUF** bytes may be nonatomic: the kernel may interleave the data with data written by other processes. POSIX.1 requires **PIPE_BUF** to be at least 512 bytes. (On Linux, **PIPE_BUF** is 4096 bytes.) The precise semantics depend on whether the file descriptor is nonblocking (**O_NONBLOCK**), whether there are multiple writers to the pipe, and on *n*, the number of bytes to be written:

O_NONBLOCK disabled, *n* <= **PIPE_BUF**
All *n* bytes are written atomically; *write*(2) may block if there is not room for *n* bytes to be written immediately

O_NONBLOCK enabled, *n* <= **PIPE_BUF**
If there is room to write *n* bytes to the pipe, then *write*(2) succeeds immediately, writing all *n* bytes; otherwise *write*(2) fails, with *errno* set to **EAGAIN**.

O_NONBLOCK disabled, *n* > **PIPE_BUF**
The write is nonatomic: the data given to *write*(2) may be interleaved with *write*(2)s by other process; the *write*(2) blocks until *n* bytes have been written.

O_NONBLOCK enabled, *n* > **PIPE_BUF**
If the pipe is full, then *write*(2) fails, with *errno* set to **EAGAIN**. Otherwise, from 1 to *n* bytes may be written (i.e., a "partial write" may occur; the caller should check the return value from *write*(2) to see how many bytes were actually written), and these bytes may be

interleaved with writes by other processes.

Open file status flags

The only open file status flags that can be meaningfully applied to a pipe or FIFO are **O_NONBLOCK** and **O_ASYNC**.

Setting the **O_ASYNC** flag for the read end of a pipe causes a signal (**SIGIO** by default) to be generated when new input becomes available on the pipe. The target for delivery of signals must be set using the *fcntl*(2) **F_SETOWN** command. On Linux, **O_ASYNC** is supported for pipes and FIFOs only since Linux 2.6.

Portability notes

On some systems (but not Linux), pipes are bidirectional: data can be transmitted in both directions between the pipe ends. POSIX.1 requires only unidirectional pipes. Portable applications should avoid reliance on bidirectional pipe semantics.

BUGS

Before Linux 4.9, some bugs affected the handling of the *pipe-user-pages-soft* and *pipe-user-pages-hard* limits when using the *fcntl*(2) **F_SETPIPE_SZ** operation to change a pipe's capacity:

(a) When increasing the pipe capacity, the checks against the soft and hard limits were made against existing consumption, and excluded the memory required for the increased pipe capacity. The new increase in pipe capacity could then push the total memory used by the user for pipes (possibly far) over a limit. (This could also trigger the problem described next.)

Starting with Linux 4.9, the limit checking includes the memory required for the new pipe capacity.

(b) The limit checks were performed even when the new pipe capacity was less than the existing pipe capacity. This could lead to problems if a user set a large pipe capacity, and then the limits were lowered, with the result that the user could no longer decrease the pipe capacity.

Starting with Linux 4.9, checks against the limits are performed only when increasing a pipe's capacity; an unprivileged user can always decrease a pipe's capacity.

(c) The accounting and checking against the limits were done as follows:

(1) Test whether the user has exceeded the limit.
(2) Make the new pipe buffer allocation.
(3) Account new allocation against the limits.

This was racey. Multiple processes could pass point (1) simultaneously, and then allocate pipe buffers that were accounted for only in step (3), with the result that the user's pipe buffer allocation could be pushed over the limit.

Starting with Linux 4.9, the accounting step is performed before doing the allocation, and the operation fails if the limit would be exceeded.

Before Linux 4.9, bugs similar to points (a) and (c) could also occur when the kernel allocated memory for a new pipe buffer; that is, when calling *pipe*(2) and when opening a previously unopened FIFO.

SEE ALSO

mkfifo(1), *dup*(2), *fcntl*(2), *open*(2), *pipe*(2), *poll*(2), *select*(2), *socketpair*(2), *splice*(2), *stat*(2), *tee*(2), *vmsplice*(2), *mkfifo*(3), *epoll*(7), *fifo*(7)

NAME

pkeys − overview of Memory Protection Keys

DESCRIPTION

Memory Protection Keys (pkeys) are an extension to existing page-based memory permissions. Normal page permissions using page tables require expensive system calls and TLB invalidations when changing permissions. Memory Protection Keys provide a mechanism for changing protections without requiring modification of the page tables on every permission change.

To use pkeys, software must first "tag" a page in the page tables with a pkey. After this tag is in place, an application only has to change the contents of a register in order to remove write access, or all access to a tagged page.

Protection keys work in conjunction with the existing **PROT_READ**, **PROT_WRITE**, and **PROT_EXEC** permissions passed to system calls such as *mprotect*(2) and *mmap*(2), but always act to further restrict these traditional permission mechanisms.

If a process performs an access that violates pkey restrictions, it receives a **SIGSEGV** signal. See *sigaction*(2) for details of the information available with that signal.

To use the pkeys feature, the processor must support it, and the kernel must contain support for the feature on a given processor. As of early 2016 only future Intel x86 processors are supported, and this hardware supports 16 protection keys in each process. However, pkey 0 is used as the default key, so a maximum of 15 are available for actual application use. The default key is assigned to any memory region for which a pkey has not been explicitly assigned via *pkey_mprotect*(2).

Protection keys have the potential to add a layer of security and reliability to applications. But they have not been primarily designed as a security feature. For instance, WRPKRU is a completely unprivileged instruction, so pkeys are useless in any case that an attacker controls the PKRU register or can execute arbitrary instructions.

Applications should be very careful to ensure that they do not "leak" protection keys. For instance, before calling *pkey_free*(2), the application should be sure that no memory has that pkey assigned. If the application left the freed pkey assigned, a future user of that pkey might inadvertently change the permissions of an unrelated data structure, which could impact security or stability. The kernel currently allows in-use pkeys to have *pkey_free*(2) called on them because it would have processor or memory performance implications to perform the additional checks needed to disallow it. Implementation of the necessary checks is left up to applications. Applications may implement these checks by searching the */proc/*pid*/smaps* file for memory regions with the pkey assigned. Further details can be found in *proc*(5).

Any application wanting to use protection keys needs to be able to function without them. They might be unavailable because the hardware that the application runs on does not support them, the kernel code does not contain support, the kernel support has been disabled, or because the keys have all been allocated, perhaps by a library the application is using. It is recommended that applications wanting to use protection keys should simply call *pkey_alloc*(2) and test whether the call succeeds, instead of attempting to detect support for the feature in any other way.

Although unnecessary, hardware support for protection keys may be enumerated with the *cpuid* instruction. Details of how to do this can be found in the Intel Software Developers Manual. The kernel performs this enumeration and exposes the information in */proc/cpuinfo* under the "flags" field. The string "pku" in this field indicates hardware support for protection keys and the string "ospke" indicates that the kernel contains and has enabled protection keys support.

Applications using threads and protection keys should be especially careful. Threads inherit the protection key rights of the parent at the time of the *clone*(2), system call. Applications should either ensure that their own permissions are appropriate for child threads at the time when *clone*(2) is called, or ensure that each child thread can perform its own initialization of protection key rights.

Signal Handler Behavior

Each time a signal handler is invoked (including nested signals), the thread is temporarily given a new, default set of protection key rights that override the rights from the interrupted context. This means that applications must re-establish their desired protection key rights upon entering a signal handler if the desired rights differ from the defaults. The rights of any interrupted context are restored when the signal handler returns.

This signal behavior is unusual and is due to the fact that the x86 PKRU register (which stores protection key access rights) is managed with the same hardware mechanism (XSAVE) that manages floating-point registers. The signal behavior is the same as that of floating-point registers.

Protection Keys system calls

The Linux kernel implements the following pkey-related system calls: *pkey_mprotect*(2), *pkey_alloc*(2), and *pkey_free*(2).

The Linux pkey system calls are available only if the kernel was configured and built with the **CONFIG_X86_INTEL_MEMORY_PROTECTION_KEYS** option.

EXAMPLES

The program below allocates a page of memory with read and write permissions. It then writes some data to the memory and successfully reads it back. After that, it attempts to allocate a protection key and disallows access to the page by using the WRPKRU instruction. It then tries to access the page, which we now expect to cause a fatal signal to the application.

```
$ ./a.out
buffer contains: 73
about to read buffer again...
Segmentation fault (core dumped)
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int
main(void)
{
    int status;
    int pkey;
    int *buffer;

    /*
     * Allocate one page of memory.
     */
    buffer = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (buffer == MAP_FAILED)
        err(EXIT_FAILURE, "mmap");

    /*
     * Put some random data into the page (still OK to touch).
     */
    *buffer = __LINE__;
    printf("buffer contains: %d\n", *buffer);

    /*
     * Allocate a protection key:
     */
    pkey = pkey_alloc(0, 0);
    if (pkey == -1)
        err(EXIT_FAILURE, "pkey_alloc");

    /*
     * Disable access to any memory with "pkey" set,
     * even though there is none right now.
```

```
         */
        status = pkey_set(pkey, PKEY_DISABLE_ACCESS);
        if (status)
            err(EXIT_FAILURE, "pkey_set");

        /*
         * Set the protection key on "buffer".
         * Note that it is still read/write as far as mprotect() is
         * concerned and the previous pkey_set() overrides it.
         */
        status = pkey_mprotect(buffer, getpagesize(),
                               PROT_READ | PROT_WRITE, pkey);
        if (status == -1)
            err(EXIT_FAILURE, "pkey_mprotect");

        printf("about to read buffer again...\n");

        /*
         * This will crash, because we have disallowed access.
         */
        printf("buffer contains: %d\n", *buffer);

        status = pkey_free(pkey);
        if (status == -1)
            err(EXIT_FAILURE, "pkey_free");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

pkey_alloc(2), *pkey_free*(2), *pkey_mprotect*(2), *sigaction*(2)

NAME

posixoptions − optional parts of the POSIX standard

DESCRIPTION

The POSIX standard (the information below is from POSIX.1-2001) describes a set of behaviors and interfaces for a compliant system. However, many interfaces are optional and there are feature test macros to test the availability of interfaces at compile time, and functions *sysconf*(3), *fpathconf*(3), *pathconf*(3), *confstr*(3) to do this at run time. From shell scripts one can use *getconf* (1)For more detail, see *sysconf*(3).

We give the name of the POSIX abbreviation, the option, the name of the *sysconf*(3) parameter used to inquire about the option, and possibly a very short description. Much more precise detail can be found in the POSIX standard itself, versions of which can nowadays be accessed freely on the web.

ADV - _POSIX_ADVISORY_INFO - _SC_ADVISORY_INFO

The following advisory functions are present:

posix_fadvise()
posix_fallocate()
posix_memalign()
posix_madvise()

AIO - _POSIX_ASYNCHRONOUS_IO - _SC_ASYNCHRONOUS_IO

The header <*aio.h*> is present. The following functions are present:

aio_cancel()
aio_error()
aio_fsync()
aio_read()
aio_return()
aio_suspend()
aio_write()
lio_listio()

BAR - _POSIX_BARRIERS - _SC_BARRIERS

This option implies the **_POSIX_THREADS** and **_POSIX_THREAD_SAFE_FUNCTIONS** options. The following functions are present:

pthread_barrier_destroy()
pthread_barrier_init()
pthread_barrier_wait()
pthread_barrierattr_destroy()
pthread_barrierattr_init()

--- - POSIX_CHOWN_RESTRICTED

If this option is in effect (as it always is under POSIX.1-2001), then only root may change the owner of a file, and nonroot can set the group of a file only to one of the groups it belongs to. This affects the following functions:

chown()
fchown()

CS - _POSIX_CLOCK_SELECTION - _SC_CLOCK_SELECTION

This option implies the **_POSIX_TIMERS** option. The following functions are present:

pthread_condattr_getclock()
pthread_condattr_setclock()
clock_nanosleep()

If **CLOCK_REALTIME** is changed by the function *clock_settime*(), then this affects all timers set for an absolute time.

CPT - _POSIX_CPUTIME - _SC_CPUTIME

The **CLOCK_PROCESS_CPUTIME_ID** clock ID is supported. The initial value of this clock is 0 for each process. This option implies the **_POSIX_TIMERS** option. The function *clock_getcpuclockid*() is present.

--- - _POSIX_FILE_LOCKING - _SC_FILE_LOCKING
This option has been deleted. Not in final XPG6.

FSC - _POSIX_FSYNC - _SC_FSYNC
The function *fsync*() is present.

IP6 - _POSIX_IPV6 - _SC_IPV6
Internet Protocol Version 6 is supported.

--- - _POSIX_JOB_CONTROL - _SC_JOB_CONTROL
If this option is in effect (as it always is under POSIX.1-2001), then the system implements POSIX-style job control, and the following functions are present:

setpgid()
tcdrain()
tcflush()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()

MF - _POSIX_MAPPED_FILES - _SC_MAPPED_FILES
Shared memory is supported. The include file <*sys/mman.h*> is present. The following functions are present:

mmap()
msync()
munmap()

ML - _POSIX_MEMLOCK - _SC_MEMLOCK
Shared memory can be locked into core. The following functions are present:

mlockall()
munlockall()

MR/MLR - _POSIX_MEMLOCK_RANGE - _SC_MEMLOCK_RANGE
More precisely, ranges can be locked into core. The following functions are present:

mlock()
munlock()

MPR - _POSIX_MEMORY_PROTECTION - _SC_MEMORY_PROTECTION
The function *mprotect*() is present.

MSG - _POSIX_MESSAGE_PASSING - _SC_MESSAGE_PASSING
The include file <*mqueue.h*> is present. The following functions are present:

mq_close()
mq_getattr()
mq_notify()
mq_open()
mq_receive()
mq_send()
mq_setattr()
mq_unlink()

MON - _POSIX_MONOTONIC_CLOCK - _SC_MONOTONIC_CLOCK
CLOCK_MONOTONIC is supported. This option implies the **_POSIX_TIMERS** option. The following functions are affected:

aio_suspend()
clock_getres()
clock_gettime()
clock_settime()
timer_create()

--- - _POSIX_MULTI_PROCESS - _SC_MULTI_PROCESS
This option has been deleted. Not in final XPG6.

--- - _POSIX_NO_TRUNC
If this option is in effect (as it always is under POSIX.1-2001), then pathname components longer than **NAME_MAX** are not truncated, but give an error. This property may be dependent on the path prefix of the component.

PIO - _POSIX_PRIORITIZED_IO - _SC_PRIORITIZED_IO
This option says that one can specify priorities for asynchronous I/O. This affects the functions

aio_read()
aio_write()

PS - _POSIX_PRIORITY_SCHEDULING - _SC_PRIORITY_SCHEDULING
The include file <*sched.h*> is present. The following functions are present:

sched_get_priority_max()
sched_get_priority_min()
sched_getparam()
sched_getscheduler()
sched_rr_get_interval()
sched_setparam()
sched_setscheduler()
sched_yield()

If also **_POSIX_SPAWN** is in effect, then the following functions are present:

posix_spawnattr_getschedparam()
posix_spawnattr_getschedpolicy()
posix_spawnattr_setschedparam()
posix_spawnattr_setschedpolicy()

RS - _POSIX_RAW_SOCKETS
Raw sockets are supported. The following functions are affected:

getsockopt()
setsockopt()

--- - _POSIX_READER_WRITER_LOCKS - _SC_READER_WRITER_LOCKS
This option implies the **_POSIX_THREADS** option. Conversely, under POSIX.1-2001 the **_POSIX_THREADS** option implies this option.

The following functions are present:

pthread_rwlock_destroy()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
pthread_rwlockattr_destroy()
pthread_rwlockattr_init()

RTS - _POSIX_REALTIME_SIGNALS - _SC_REALTIME_SIGNALS
Realtime signals are supported. The following functions are present:

sigqueue()
sigtimedwait()
sigwaitinfo()

--- - _POSIX_REGEXP - _SC_REGEXP
If this option is in effect (as it always is under POSIX.1-2001), then POSIX regular expressions are supported and the following functions are present:

regcomp()
regerror()

regexec()
regfree()

--- - _POSIX_SAVED_IDS - _SC_SAVED_IDS
If this option is in effect (as it always is under POSIX.1-2001), then a process has a saved set-user-ID and a saved set-group-ID. The following functions are affected:

exec()
kill()
seteuid()
setegid()
setgid()
setuid()

SEM - _POSIX_SEMAPHORES - _SC_SEMAPHORES
The include file <*semaphore.h*> is present. The following functions are present:

sem_close()
sem_destroy()
sem_getvalue()
sem_init()
sem_open()
sem_post()
sem_trywait()
sem_unlink()
sem_wait()

SHM - _POSIX_SHARED_MEMORY_OBJECTS - _SC_SHARED_MEMORY_OBJECTS
The following functions are present:

mmap()
munmap()
shm_open()
shm_unlink()

--- - _POSIX_SHELL - _SC_SHELL
If this option is in effect (as it always is under POSIX.1-2001), the function *system*() is present.

SPN - _POSIX_SPAWN - _SC_SPAWN
This option describes support for process creation in a context where it is difficult or impossible to use *fork*(), for example, because no MMU is present.

If **_POSIX_SPAWN** is in effect, then the include file <*spawn.h*> and the following functions are present:

posix_spawn()
posix_spawn_file_actions_addclose()
posix_spawn_file_actions_adddup2()
posix_spawn_file_actions_addopen()
posix_spawn_file_actions_destroy()
posix_spawn_file_actions_init()
posix_spawnattr_destroy()
posix_spawnattr_getsigdefault()
posix_spawnattr_getflags()
posix_spawnattr_getpgroup()
posix_spawnattr_getsigmask()
posix_spawnattr_init()
posix_spawnattr_setsigdefault()
posix_spawnattr_setflags()
posix_spawnattr_setpgroup()
posix_spawnattr_setsigmask()
posix_spawnp()

If also **_POSIX_PRIORITY_SCHEDULING** is in effect, then the following functions are present:

posix_spawnattr_getschedparam()

posix_spawnattr_getschedpolicy()
posix_spawnattr_setschedparam()
posix_spawnattr_setschedpolicy()

SPI - _POSIX_SPIN_LOCKS - _SC_SPIN_LOCKS
This option implies the **_POSIX_THREADS** and **_POSIX_THREAD_SAFE_FUNCTIONS** options. The following functions are present:

pthread_spin_destroy()
pthread_spin_init()
pthread_spin_lock()
pthread_spin_trylock()
pthread_spin_unlock()

SS - _POSIX_SPORADIC_SERVER - _SC_SPORADIC_SERVER
The scheduling policy **SCHED_SPORADIC** is supported. This option implies the **_POSIX_PRIORITY_SCHEDULING** option. The following functions are affected:

sched_setparam()
sched_setscheduler()

SIO - _POSIX_SYNCHRONIZED_IO - _SC_SYNCHRONIZED_IO
The following functions are affected:

open()
msync()
fsync()
fdatasync()

TSA - _POSIX_THREAD_ATTR_STACKADDR - _SC_THREAD_ATTR_STACKADDR
The following functions are affected:

pthread_attr_getstack()
pthread_attr_getstackaddr()
pthread_attr_setstack()
pthread_attr_setstackaddr()

TSS - _POSIX_THREAD_ATTR_STACKSIZE - _SC_THREAD_ATTR_STACKSIZE
The following functions are affected:

pthread_attr_getstack()
pthread_attr_getstacksize()
pthread_attr_setstack()
pthread_attr_setstacksize()

TCT - _POSIX_THREAD_CPUTIME - _SC_THREAD_CPUTIME
The clockID CLOCK_THREAD_CPUTIME_ID is supported. This option implies the **_POSIX_TIMERS** option. The following functions are affected:

pthread_getcpuclockid()
clock_getres()
clock_gettime()
clock_settime()
timer_create()

TPI - _POSIX_THREAD_PRIO_INHERIT - _SC_THREAD_PRIO_INHERIT
The following functions are affected:

pthread_mutexattr_getprotocol()
pthread_mutexattr_setprotocol()

TPP - _POSIX_THREAD_PRIO_PROTECT - _SC_THREAD_PRIO_PROTECT
The following functions are affected:

pthread_mutex_getprioceiling()
pthread_mutex_setprioceiling()
pthread_mutexattr_getprioceiling()
pthread_mutexattr_getprotocol()

pthread_mutexattr_setprioceiling()
pthread_mutexattr_setprotocol()

TPS - _POSIX_THREAD_PRIORITY_SCHEDULING - _SC_THREAD_PRIORITY_SCHEDULING
If this option is in effect, the different threads inside a process can run with different priorities and/or different schedulers. The following functions are affected:

pthread_attr_getinheritsched()
pthread_attr_getschedpolicy()
pthread_attr_getscope()
pthread_attr_setinheritsched()
pthread_attr_setschedpolicy()
pthread_attr_setscope()
pthread_getschedparam()
pthread_setschedparam()
pthread_setschedprio()

TSH - _POSIX_THREAD_PROCESS_SHARED - _SC_THREAD_PROCESS_SHARED
The following functions are affected:

pthread_barrierattr_getpshared()
pthread_barrierattr_setpshared()
pthread_condattr_getpshared()
pthread_condattr_setpshared()
pthread_mutexattr_getpshared()
pthread_mutexattr_setpshared()
pthread_rwlockattr_getpshared()
pthread_rwlockattr_setpshared()

TSF - _POSIX_THREAD_SAFE_FUNCTIONS - _SC_THREAD_SAFE_FUNCTIONS
The following functions are affected:

readdir_r()
getgrgid_r()
getgrnam_r()
getpwnam_r()
getpwuid_r()
flockfile()
ftrylockfile()
funlockfile()
getc_unlocked()
getchar_unlocked()
putc_unlocked()
putchar_unlocked()
rand_r()
strerror_r()
strtok_r()
asctime_r()
ctime_r()
gmtime_r()
localtime_r()

TSP - _POSIX_THREAD_SPORADIC_SERVER - _SC_THREAD_SPORADIC_SERVER
This option implies the **_POSIX_THREAD_PRIORITY_SCHEDULING** option. The following functions are affected:

sched_getparam()
sched_setparam()
sched_setscheduler()

THR - _POSIX_THREADS - _SC_THREADS
Basic support for POSIX threads is available. The following functions are present:

pthread_atfork()

pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getschedparam()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setschedparam()
pthread_cancel()
pthread_cleanup_push()
pthread_cleanup_pop()
pthread_cond_broadcast()
pthread_cond_destroy()
pthread_cond_init()
pthread_cond_signal()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_condattr_destroy()
pthread_condattr_init()
pthread_create()
pthread_detach()
pthread_equal()
pthread_exit()
pthread_getspecific()
pthread_join()
pthread_key_create()
pthread_key_delete()
pthread_mutex_destroy()
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_mutexattr_destroy()
pthread_mutexattr_init()
pthread_once()
pthread_rwlock_destroy()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
pthread_rwlockattr_destroy()
pthread_rwlockattr_init()
pthread_self ()
pthread_setcancelstate()
pthread_setcanceltype()
pthread_setspecific()
pthread_testcancel()

TMO - _POSIX_TIMEOUTS - _SC_TIMEOUTS

The following functions are present:

mq_timedreceive()
mq_timedsend()
pthread_mutex_timedlock()
pthread_rwlock_timedrdlock()
pthread_rwlock_timedwrlock()
sem_timedwait()
posix_trace_timedgetnext_event()

TMR - _POSIX_TIMERS - _SC_TIMERS
The following functions are present:

clock_getres()
clock_gettime()
clock_settime()
nanosleep()
timer_create()
timer_delete()
timer_gettime()
timer_getoverrun()
timer_settime()

TRC - _POSIX_TRACE - _SC_TRACE
POSIX tracing is available. The following functions are present:

posix_trace_attr_destroy()
posix_trace_attr_getclockres()
posix_trace_attr_getcreatetime()
posix_trace_attr_getgenversion()
posix_trace_attr_getmaxdatasize()
posix_trace_attr_getmaxsystemeventsize()
posix_trace_attr_getmaxusereventsize()
posix_trace_attr_getname()
posix_trace_attr_getstreamfullpolicy()
posix_trace_attr_getstreamsize()
posix_trace_attr_init()
posix_trace_attr_setmaxdatasize()
posix_trace_attr_setname()
posix_trace_attr_setstreamsize()
posix_trace_attr_setstreamfullpolicy()
posix_trace_clear()
posix_trace_create()
posix_trace_event()
posix_trace_eventid_equal()
posix_trace_eventid_get_name()
posix_trace_eventid_open()
posix_trace_eventtypelist_getnext_id()
posix_trace_eventtypelist_rewind()
posix_trace_flush()
posix_trace_get_attr()
posix_trace_get_status()
posix_trace_getnext_event()
posix_trace_shutdown()
posix_trace_start()
posix_trace_stop()
posix_trace_trygetnext_event()

TEF - _POSIX_TRACE_EVENT_FILTER - _SC_TRACE_EVENT_FILTER
This option implies the **_POSIX_TRACE** option. The following functions are present:

posix_trace_eventset_add()
posix_trace_eventset_del()
posix_trace_eventset_empty()
posix_trace_eventset_fill()
posix_trace_eventset_ismember()
posix_trace_get_filter()
posix_trace_set_filter()
posix_trace_trid_eventid_open()

TRI - _POSIX_TRACE_INHERIT - _SC_TRACE_INHERIT
Tracing children of the traced process is supported. This option implies the **_POSIX_TRACE** option. The following functions are present:

posix_trace_attr_getinherited()
posix_trace_attr_setinherited()

TRL - _POSIX_TRACE_LOG - _SC_TRACE_LOG
This option implies the **_POSIX_TRACE** option. The following functions are present:

posix_trace_attr_getlogfullpolicy()
posix_trace_attr_getlogsize()
posix_trace_attr_setlogfullpolicy()
posix_trace_attr_setlogsize()
posix_trace_close()
posix_trace_create_withlog()
posix_trace_open()
posix_trace_rewind()

TYM - _POSIX_TYPED_MEMORY_OBJECTS - _SC_TYPED_MEMORY_OBJECT
The following functions are present:

posix_mem_offset()
posix_typed_mem_get_info()
posix_typed_mem_open()

--- - _POSIX_VDISABLE
Always present (probably 0). Value to set a changeable special control character to indicate that it is disabled.

X/OPEN SYSTEM INTERFACE EXTENSIONS

XSI - _XOPEN_CRYPT - _SC_XOPEN_CRYPT
The following functions are present:

crypt()
encrypt()
setkey()

XSI - _XOPEN_REALTIME - _SC_XOPEN_REALTIME
This option implies the following options:

_POSIX_ASYNCHRONOUS_IO==200112L
_POSIX_FSYNC
_POSIX_MAPPED_FILES
_POSIX_MEMLOCK==200112L
_POSIX_MEMLOCK_RANGE==200112L
_POSIX_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING==200112L
_POSIX_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING==200112L
_POSIX_REALTIME_SIGNALS==200112L
_POSIX_SEMAPHORES==200112L
_POSIX_SHARED_MEMORY_OBJECTS==200112L
_POSIX_SYNCHRONIZED_IO==200112L
_POSIX_TIMERS==200112L

ADV - --- - ---
The Advanced Realtime option group implies that the following options are all defined to 200112L:

_POSIX_ADVISORY_INFO
_POSIX_CLOCK_SELECTION
(implies **_POSIX_TIMERS**)
_POSIX_CPUTIME
(implies **_POSIX_TIMERS**)

_POSIX_MONOTONIC_CLOCK
(implies **_POSIX_TIMERS**)
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
(implies **_POSIX_PRIORITY_SCHEDULING**)
_POSIX_TIMEOUTS
_POSIX_TYPED_MEMORY_OBJECTS

XSI - _XOPEN_REALTIME_THREADS - _SC_XOPEN_REALTIME_THREADS
This option implies that the following options are all defined to 200112L:

_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING

ADVANCED REALTIME THREADS - --- - ---
This option implies that the following options are all defined to 200112L:

_POSIX_BARRIERS
(implies **_POSIX_THREADS**, **_POSIX_THREAD_SAFE_FUNCTIONS**)
_POSIX_SPIN_LOCKS
(implies **_POSIX_THREADS**, **_POSIX_THREAD_SAFE_FUNCTIONS**)
_POSIX_THREAD_CPUTIME
(implies **_POSIX_TIMERS**)
_POSIX_THREAD_SPORADIC_SERVER
(implies **_POSIX_THREAD_PRIORITY_SCHEDULING**)

TRACING - --- - ---
This option implies that the following options are all defined to 200112L:

_POSIX_TRACE
_POSIX_TRACE_EVENT_FILTER
_POSIX_TRACE_LOG
_POSIX_TRACE_INHERIT

STREAMS - _XOPEN_STREAMS - _SC_XOPEN_STREAMS
The following functions are present:

fattach()
fdetach()
getmsg()
getpmsg()
ioctl()
isastream()
putmsg()
putpmsg()

XSI - _XOPEN_LEGACY - _SC_XOPEN_LEGACY
Functions included in the legacy option group were previously mandatory, but are now optional in this version. The following functions are present:

bcmp()
bcopy()
bzero()
ecvt()
fcvt()
ftime()
gcvt()
getwd()
index()
mktemp()
rindex()
utimes()
wcswcs()

XSI - _XOPEN_UNIX - _SC_XOPEN_UNIX
The following functions are present:

mmap()
munmap()
msync()

This option implies the following options:

_POSIX_FSYNC
_POSIX_MAPPED_FILES
_POSIX_MEMORY_PROTECTION
_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PROCESS_SHARED
_POSIX_THREAD_SAFE_FUNCTIONS
_POSIX_THREADS

This option may imply the following options from the XSI option groups:

Encryption (**_XOPEN_CRYPT**)
Realtime (**_XOPEN_REALTIME**)
Advanced Realtime (**ADB**)
Realtime Threads (**_XOPEN_REALTIME_THREADS**)
Advanced Realtime Threads (**ADVANCED REALTIME THREADS**)
Tracing (**TRACING**)
XSI Streams (**STREAMS**)
Legacy (**_XOPEN_LEGACY**)

SEE ALSO

sysconf(3), *standards*(7)

NAME

process-keyring − per-process shared keyring

DESCRIPTION

The process keyring is a keyring used to anchor keys on behalf of a process. It is created only when a process requests it. The process keyring has the name (description) *_pid*.

A special serial number value, **KEY_SPEC_PROCESS_KEYRING**, is defined that can be used in lieu of the actual serial number of the calling process's process keyring.

From the *keyctl*(1) utility, '**@p**' can be used instead of a numeric key ID in much the same way, but since *keyctl*(1) is a program run after forking, this is of no utility.

A thread created using the *clone*(2) **CLONE_THREAD** flag has the same process keyring as the caller of *clone*(2). When a new process is created using **fork**() it initially has no process keyring. A process's process keyring is cleared on *execve*(2). The process keyring is destroyed when the last thread that refers to it terminates.

If a process doesn't have a process keyring when it is accessed, then the process keyring will be created if the keyring is to be modified; otherwise, the error **ENOKEY** results.

SEE ALSO

keyctl(1), *keyctl*(3), *keyrings*(7), *persistent-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7)

NAME

pthreads − POSIX threads

DESCRIPTION

POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

POSIX.1 also requires that threads share a range of other attributes (i.e., these attributes are process-wide rather than per-thread):

- process ID
- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs
- open file descriptors
- record locks (see *fcntl*(2))
- signal dispositions
- file mode creation mask (**umask**(2))
- current directory (**chdir**(2)) and root directory (**chroot**(2))
- interval timers (**setitimer**(2)) and POSIX timers (**timer_create**(2))
- nice value (**setpriority**(2))
- resource limits (**setrlimit**(2))
- measurements of the consumption of CPU time (**times**(2)) and resources (**getrusage**(2))

As well as the stack, POSIX.1 specifies that various other attributes are distinct for each thread, including:

- thread ID (the *pthread_t* data type)
- signal mask (**pthread_sigmask**(3))
- the *errno* variable
- alternate signal stack (**sigaltstack**(2))
- real-time scheduling policy and priority (**sched**(7))

The following Linux-specific features are also per-thread:

- capabilities (see *capabilities*(7))
- CPU affinity (**sched_setaffinity**(2))

Pthreads function return values

Most pthreads functions return 0 on success, and an error number on failure. The error numbers that can be returned have the same meaning as the error numbers returned in *errno* by conventional system calls and C library functions. Note that the pthreads functions do not set *errno*. For each of the pthreads functions that can return an error, POSIX.1-2001 specifies that the function can never fail with the error **EINTR**.

Thread IDs

Each of the threads in a process has a unique thread identifier (stored in the type *pthread_t*). This identifier is returned to the caller of *pthread_create*(3), and a thread can obtain its own thread identifier using *pthread_self*(3).

Thread IDs are guaranteed to be unique only within a process. (In all pthreads functions that accept a thread ID as an argument, that ID by definition refers to a thread in the same process as the caller.)

The system may reuse a thread ID after a terminated thread has been joined, or a detached thread has terminated. POSIX says: "If an application attempts to use a thread ID whose lifetime has ended, the

behavior is undefined."

Thread-safe functions

A thread-safe function is one that can be safely (i.e., it will deliver the same results regardless of whether it is) called from multiple threads at the same time.

POSIX.1-2001 and POSIX.1-2008 require that all functions specified in the standard shall be thread-safe, except for the following functions:

```
asctime()
basename()
catgets()
crypt()
ctermid() if passed a non-NULL argument
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()
ecvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
encrypt()
endgrent()
endpwent()
endutxent()
fcvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
ftw()
gcvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostbyname() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
```

```
getservent()
getutxent()
getutxid()
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()
putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strsignal() [Added in POSIX.1-2008]
strtok()
system() [Added in POSIX.1-2008]
tmpnam() if passed a non-NULL argument
ttyname()
unsetenv()
wcrtomb() if its final argument is NULL
wcsrtombs() if its final argument is NULL
wcstombs()
wctomb()
```

Async-cancel-safe functions

An async-cancel-safe function is one that can be safely called in an application where asynchronous cancelability is enabled (see *pthread_setcancelstate*(3)).

Only the following functions are required to be async-cancel-safe by POSIX.1-2001 and POSIX.1-2008:

```
pthread_cancel()
pthread_setcancelstate()
pthread_setcanceltype()
```

Cancelation points

POSIX.1 specifies that certain functions must, and certain other functions may, be cancelation points. If a thread is cancelable, its cancelability type is deferred, and a cancelation request is pending for the thread, then the thread is canceled when it calls a function that is a cancelation point.

The following functions are required to be cancelation points by POSIX.1-2001 and/or POSIX.1-2008:

```
accept()
```

```
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl() F_SETLKW
fdatasync()
fsync()
getmsg()
getpmsg()
lockf() F_LOCK
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat() [Added in POSIX.1-2008]
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()
putmsg()
putpmsg()
pwrite()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_timedwait()
sem_wait()
send()
sendmsg()
sendto()
sigpause() [POSIX.1-2001 only (moves to "may" list in POSIX.1-2008)]
sigsuspend()
sigtimedwait()
sigwait()
sigwaitinfo()
sleep()
system()
tcdrain()
usleep() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
wait()
waitid()
waitpid()
write()
writev()
```

The following functions may be cancelation points according to POSIX.1-2001 and/or POSIX.1-2008:

```
access()
asctime()
asctime_r()
catclose()
catgets()
catopen()
chmod() [Added in POSIX.1-2008]
chown() [Added in POSIX.1-2008]
closedir()
closelog()
ctermid()
ctime()
ctime_r()
dbm_close()
dbm_delete()
dbm_fetch()
dbm_nextkey()
dbm_open()
dbm_store()
dlclose()
dlopen()
dprintf() [Added in POSIX.1-2008]
endgrent()
endhostent()
endnetent()
endprotoent()
endpwent()
endservent()
endutxent()
faccessat() [Added in POSIX.1-2008]
fchmod() [Added in POSIX.1-2008]
fchmodat() [Added in POSIX.1-2008]
fchown() [Added in POSIX.1-2008]
fchownat() [Added in POSIX.1-2008]
fclose()
fcntl() (for any value of cmd argument)
fflush()
fgetc()
fgetpos()
fgets()
fgetwc()
fgetws()
fmtmsg()
fopen()
fpathconf()
fprintf()
fputc()
fputs()
fputwc()
fputws()
fread()
freopen()
fscanf()
fseek()
fseeko()
fsetpos()
fstat()
fstatat() [Added in POSIX.1-2008]
ftell()
```

```
ftello()
ftw()
futimens() [Added in POSIX.1-2008]
fwprintf()
fwrite()
fwscanf()
getaddrinfo()
getc()
getc_unlocked()
getchar()
getchar_unlocked()
getcwd()
getdate()
getdelim() [Added in POSIX.1-2008]
getgrent()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
gethostbyaddr() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostbyname() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostent()
gethostid()
gethostname()
getline() [Added in POSIX.1-2008]
getlogin()
getlogin_r()
getnameinfo()
getnetbyaddr()
getnetbyname()
getnetent()
getopt() (if opterr is nonzero)
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwnam_r()
getpwuid()
getpwuid_r()
gets()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
getwc()
getwchar()
getwd() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
glob()
iconv_close()
iconv_open()
ioctl()
link()
linkat() [Added in POSIX.1-2008]
lio_listio() [Added in POSIX.1-2008]
```

```
localtime()
localtime_r()
lockf() [Added in POSIX.1-2008]
lseek()
lstat()
mkdir() [Added in POSIX.1-2008]
mkdirat() [Added in POSIX.1-2008]
mkdtemp() [Added in POSIX.1-2008]
mkfifo() [Added in POSIX.1-2008]
mkfifoat() [Added in POSIX.1-2008]
mknod() [Added in POSIX.1-2008]
mknodat() [Added in POSIX.1-2008]
mkstemp()
mktime()
nftw()
opendir()
openlog()
pathconf()
pclose()
perror()
popen()
posix_fadvise()
posix_fallocate()
posix_madvise()
posix_openpt()
posix_spawn()
posix_spawnp()
posix_trace_clear()
posix_trace_close()
posix_trace_create()
posix_trace_create_withlog()
posix_trace_eventtypelist_getnext_id()
posix_trace_eventtypelist_rewind()
posix_trace_flush()
posix_trace_get_attr()
posix_trace_get_filter()
posix_trace_get_status()
posix_trace_getnext_event()
posix_trace_open()
posix_trace_rewind()
posix_trace_set_filter()
posix_trace_shutdown()
posix_trace_timedgetnext_event()
posix_typed_mem_open()
printf()
psiginfo() [Added in POSIX.1-2008]
psignal() [Added in POSIX.1-2008]
pthread_rwlock_rdlock()
pthread_rwlock_timedrdlock()
pthread_rwlock_timedwrlock()
pthread_rwlock_wrlock()
putc()
putc_unlocked()
putchar()
putchar_unlocked()
puts()
pututxline()
putwc()
putwchar()
```

```
readdir()
readdir_r()
readlink() [Added in POSIX.1-2008]
readlinkat() [Added in POSIX.1-2008]
remove()
rename()
renameat() [Added in POSIX.1-2008]
rewind()
rewinddir()
scandir() [Added in POSIX.1-2008]
scanf()
seekdir()
semop()
setgrent()
sethostent()
setnetent()
setprotoent()
setpwent()
setservent()
setutxent()
sigpause() [Added in POSIX.1-2008]
stat()
strerror()
strerror_r()
strftime()
symlink()
symlinkat() [Added in POSIX.1-2008]
sync()
syslog()
tmpfile()
tmpnam()
ttyname()
ttyname_r()
tzset()
ungetc()
ungetwc()
unlink()
unlinkat() [Added in POSIX.1-2008]
utime() [Added in POSIX.1-2008]
utimensat() [Added in POSIX.1-2008]
utimes() [Added in POSIX.1-2008]
vdprintf() [Added in POSIX.1-2008]
vfprintf()
vfwprintf()
vprintf()
vwprintf()
wcsftime()
wordexp()
wprintf()
wscanf()
```

An implementation may also mark other functions not specified in the standard as cancelation points. In particular, an implementation is likely to mark any nonstandard function that may block as a cancelation point. (This includes most functions that can touch files.)

It should be noted that even if an application is not using asynchronous cancelation, that calling a function from the above list from an asynchronous signal handler may cause the equivalent of asynchronous cancelation. The underlying user code may not expect asynchronous cancelation and the state of the user data may become inconsistent. Therefore signals should be used with caution when entering a region of deferred cancelation.

Compiling on Linux

On Linux, programs that use the Pthreads API should be compiled using *cc −pthread*.

Linux implementations of POSIX threads

Over time, two threading implementations have been provided by the GNU C library on Linux:

LinuxThreads

This is the original Pthreads implementation. Since glibc 2.4, this implementation is no longer supported.

NPTL (Native POSIX Threads Library)

This is the modern Pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called 1:1 implementations, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux *clone*(2) system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux *futex*(2) system call.

LinuxThreads

The notable features of this implementation are the following:

- In addition to the main (initial) thread, and the threads that the program creates using *pthread_create*(3), the implementation creates a "manager" thread. This thread handles thread creation and termination. (Problems can result if this thread is inadvertently killed.)
- Signals are used internally by the implementation. On Linux 2.2 and later, the first three real-time signals are used (see also *signal*(7)). On older Linux kernels, **SIGUSR1** and **SIGUSR2** are used. Applications must avoid the use of whichever set of signals is employed by the implementation.
- Threads do not share process IDs. (In effect, LinuxThreads threads are implemented as processes which share more information than usual, but which do not share a common process ID.) LinuxThreads threads (including the manager thread) are visible as separate processes using *ps*(1)

The LinuxThreads implementation deviates from the POSIX.1 specification in a number of ways, including the following:

- Calls to *getpid*(2) return a different value in each thread.
- Calls to *getppid*(2) in threads other than the main thread return the process ID of the manager thread; instead *getppid*(2) in these threads should return the same value as *getppid*(2) in the main thread.
- When one thread creates a new child process using *fork*(2), any thread should be able to *wait*(2) on the child. However, the implementation allows only the thread that created the child to *wait*(2) on it.
- When a thread calls *execve*(2), all other threads are terminated (as required by POSIX.1). However, the resulting process has the same PID as the thread that called *execve*(2): it should have the same PID as the main thread.
- Threads do not share user and group IDs. This can cause complications with set-user-ID programs and can cause failures in Pthreads functions if an application changes its credentials using *seteuid*(2) or similar.
- Threads do not share a common session ID and process group ID.
- Threads do not share record locks created using *fcntl*(2).
- The information returned by *times*(2) and *getrusage*(2) is per-thread rather than process-wide.
- Threads do not share semaphore undo values (see *semop*(2)).
- Threads do not share interval timers.
- Threads do not share a common nice value.

- POSIX.1 distinguishes the notions of signals that are directed to the process as a whole and signals that are directed to individual threads. According to POSIX.1, a process-directed signal (sent using *kill*(2), for example) should be handled by a single, arbitrarily selected thread within the process. LinuxThreads does not support the notion of process-directed signals: signals may be sent only to specific threads.
- Threads have distinct alternate signal stack settings. However, a new thread's alternate signal stack settings are copied from the thread that created it, so that the threads initially share an alternate signal stack. (A new thread should start with no alternate signal stack defined. If two threads handle signals on their shared alternate signal stack at the same time, unpredictable program failures are likely to occur.)

NPTL

With NPTL, all of the threads in a process are placed in the same thread group; all members of a thread group share the same PID. NPTL does not employ a manager thread.

NPTL makes internal use of the first two real-time signals; these signals cannot be used in applications. See *nptl*(7) for further details.

NPTL still has at least one nonconformance with POSIX.1:

- Threads do not share a common nice value.

Some NPTL nonconformances occur only with older kernels:

- The information returned by *times*(2) and *getrusage*(2) is per-thread rather than process-wide (fixed in Linux 2.6.9).
- Threads do not share resource limits (fixed in Linux 2.6.10).
- Threads do not share interval timers (fixed in Linux 2.6.12).
- Only the main thread is permitted to start a new session using *setsid*(2) (fixed in Linux 2.6.16).
- Only the main thread is permitted to make the process into a process group leader using *setpgid*(2) (fixed in Linux 2.6.16).
- Threads have distinct alternate signal stack settings. However, a new thread's alternate signal stack settings are copied from the thread that created it, so that the threads initially share an alternate signal stack (fixed in Linux 2.6.16).

Note the following further points about the NPTL implementation:

- If the stack size soft resource limit (see the description of **RLIMIT_STACK** in *setrlimit*(2)) is set to a value other than *unlimited*, then this value defines the default stack size for new threads. To be effective, this limit must be set before the program is executed, perhaps using the *ulimit −s* shell built-in command (*limit stacksize* in the C shell).

Determining the threading implementation

Since glibc 2.3.2, the *getconf* (1) command can be used to determine the system's threading implementation, for example:

```
bash$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.3.4
```

With older glibc versions, a command such as the following should be sufficient to determine the default threading implementation:

```
bash$ $( ldd /bin/ls | grep libc.so | awk '{print $3}' ) | \
                egrep -i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

Selecting the threading implementation: LD_ASSUME_KERNEL

On systems with a glibc that supports both LinuxThreads and NPTL (i.e., glibc 2.3.*x*), the **LD_ASSUME_KERNEL** environment variable can be used to override the dynamic linker's default choice of threading implementation. This variable tells the dynamic linker to assume that it is running on top of a particular kernel version. By specifying a kernel version that does not provide the support required by NPTL, we can force the use of LinuxThreads. (The most likely reason for doing this is to run a (broken) application that depends on some nonconformant behavior in LinuxThreads.) For example:

```
bash$ $( LD_ASSUME_KERNEL=2.2.5 ldd /bin/ls | grep libc.so | \
```

```
        awk '{print $3}' ) | egrep -i 'threads|nptl'
linuxthreads-0.10 by Xavier Leroy
```

SEE ALSO

clone(2), *fork*(2), *futex*(2), *gettid*(2), *proc*(5), *attributes*(7), *futex*(7), *nptl*(7), *sigevent*(3type), *signal*(7)

Various Pthreads manual pages, for example: *pthread_atfork*(3), *pthread_attr_init*(3), *pthread_cancel*(3), *pthread_cleanup_push*(3), *pthread_cond_signal*(3), *pthread_cond_wait*(3), *pthread_create*(3), *pthread_detach*(3), *pthread_equal*(3), *pthread_exit*(3), *pthread_key_create*(3), *pthread_kill*(3), *pthread_mutex_lock*(3), *pthread_mutex_unlock*(3), *pthread_mutexattr_destroy*(3), *pthread_mutexattr_init*(3), *pthread_once*(3), *pthread_spin_init*(3), *pthread_spin_lock*(3), *pthread_rwlockattr_setkind_np*(3), *pthread_setcancelstate*(3), *pthread_setcanceltype*(3), *pthread_setspecific*(3), *pthread_sigmask*(3), *pthread_sigqueue*(3), and *pthread_testcancel*(3)

NAME

pty − pseudoterminal interfaces

DESCRIPTION

A pseudoterminal (sometimes abbreviated "pty") is a pair of virtual character devices that provide a bidirectional communication channel. One end of the channel is called the *master*; the other end is called the *slave*.

The slave end of the pseudoterminal provides an interface that behaves exactly like a classical terminal. A process that expects to be connected to a terminal, can open the slave end of a pseudoterminal and then be driven by a program that has opened the master end. Anything that is written on the master end is provided to the process on the slave end as though it was input typed on a terminal. For example, writing the interrupt character (usually control-C) to the master device would cause an interrupt signal (**SIGINT**) to be generated for the foreground process group that is connected to the slave. Conversely, anything that is written to the slave end of the pseudoterminal can be read by the process that is connected to the master end.

Data flow between master and slave is handled asynchronously, much like data flow with a physical terminal. Data written to the slave will be available at the master promptly, but may not be available immediately. Similarly, there may be a small processing delay between a write to the master, and the effect being visible at the slave.

Historically, two pseudoterminal APIs have evolved: BSD and System V. SUSv1 standardized a pseudoterminal API based on the System V API, and this API should be employed in all new programs that use pseudoterminals.

Linux provides both BSD-style and (standardized) System V-style pseudoterminals. System V-style terminals are commonly called UNIX 98 pseudoterminals on Linux systems.

Since Linux 2.6.4, BSD-style pseudoterminals are considered deprecated: support can be disabled when building the kernel by disabling the **CONFIG_LEGACY_PTYS** option. (Starting with Linux 2.6.30, that option is disabled by default in the mainline kernel.) UNIX 98 pseudoterminals should be used in new applications.

UNIX 98 pseudoterminals

An unused UNIX 98 pseudoterminal master is opened by calling *posix_openpt*(3). (This function opens the master clone device, */dev/ptmx*; see *pts*(4).) After performing any program-specific initializations, changing the ownership and permissions of the slave device using *grantpt*(3), and unlocking the slave using *unlockpt*(3)), the corresponding slave device can be opened by passing the name returned by *ptsname*(3) in a call to *open*(2).

The Linux kernel imposes a limit on the number of available UNIX 98 pseudoterminals. Up to and including Linux 2.6.3, this limit is configured at kernel compilation time (**CONFIG_UNIX98_PTYS**), and the permitted number of pseudoterminals can be up to 2048, with a default setting of 256. Since Linux 2.6.4, the limit is dynamically adjustable via */proc/sys/kernel/pty/max*, and a corresponding file, */proc/sys/kernel/pty/nr*, indicates how many pseudoterminals are currently in use. For further details on these two files, see *proc*(5).

BSD pseudoterminals

BSD-style pseudoterminals are provided as precreated pairs, with names of the form */dev/ptyXY* (master) and */dev/ttyXY* (slave), where X is a letter from the 16-character set [p−za−e], and Y is a letter from the 16-character set [0−9a−f]. (The precise range of letters in these two sets varies across UNIX implementations.) For example, */dev/ptyp1* and */dev/ttyp1* constitute a BSD pseudoterminal pair. A process finds an unused pseudoterminal pair by trying to *open*(2) each pseudoterminal master until an open succeeds. The corresponding pseudoterminal slave (substitute "tty" for "pty" in the name of the master) can then be opened.

FILES

/dev/ptmx
: UNIX 98 master clone device

*/dev/pts/**
: UNIX 98 slave devices

/dev/pty[p−za−e][0−9a−f]
: BSD master devices

/dev/tty[p−za−e][0−9a−f]
: BSD slave devices

NOTES

Pseudoterminals are used by applications such as network login services (*ssh*(1), *rlogin*(1), *telnet*(1)), terminal emulators such as *xterm*(1), *script*(1), *screen*(1), *tmux*(1), *unbuffer*(1), and *expect*(1)

A description of the **TIOCPKT** *ioctl*(2), which controls packet mode operation, can be found in *ioctl_tty*(2).

The BSD *ioctl*(2) operations **TIOCSTOP**, **TIOCSTART**, **TIOCUCNTL**, and **TIOCREMOTE** have not been implemented under Linux.

SEE ALSO

ioctl_tty(2), *select*(2), *setsid*(2), *forkpty*(3), *openpty*(3), *termios*(3), *pts*(4), *tty*(4)

NAME

queue − implementations of linked lists and queues

DESCRIPTION

The <*sys/queue.h*> header file provides a set of macros that define and operate on the following data structures:

SLIST singly linked lists

LIST doubly linked lists

STAILQ
: singly linked tail queues

TAILQ doubly linked tail queues

CIRCLEQ
: doubly linked circular queues

All structures support the following functionality:

- Insertion of a new entry at the head of the list.
- Insertion of a new entry after any element in the list.
- O(1) removal of an entry from the head of the list.
- Forward traversal through the list.

Code size and execution time depend on the complexity of the data structure being used, so programmers should take care to choose the appropriate one.

Singly linked lists (SLIST)

Singly linked lists are the simplest and support only the above functionality. Singly linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue. Singly linked lists add the following functionality:

- O(n) removal of any entry in the list.

Singly linked tail queues (STAILQ)

Singly linked tail queues add the following functionality:

- Entries can be added at the end of a list.
- O(n) removal of any entry in the list.
- They may be concatenated.

However:

- All list insertions must specify the head of the list.
- Each head entry requires two pointers rather than one.

Singly linked tail queues are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

Doubly linked data structures

All doubly linked types of data structures (lists and tail queues) additionally allow:

- Insertion of a new entry before any element in the list.
- O(1) removal of any entry in the list.

However:

- Each element requires two pointers rather than one.

Doubly linked lists (LIST)

Linked lists are the simplest of the doubly linked data structures. They add the following functionality over the above:

- They may be traversed backwards.

However:

- To traverse backwards, an entry to begin the traversal and the list in which it is contained must be specified.

Doubly linked tail queues (TAILQ)

Tail queues add the following functionality:

- Entries can be added at the end of a list.
- They may be traversed backwards, from tail to head.
- They may be concatenated.

However:

- All list insertions and removals must specify the head of the list.
- Each head entry requires two pointers rather than one.

Doubly linked circular queues (CIRCLEQ)

Circular queues add the following functionality over the above:

- The first and last entries are connected.

However:

- The termination condition for traversal is more complex.

STANDARDS

BSD.

HISTORY

<*sys/queue.h*> macros first appeared in 4.4BSD.

NOTES

Some BSDs provide SIMPLEQ instead of STAILQ. They are identical, but for historical reasons they were named differently on different BSDs. STAILQ originated on FreeBSD, and SIMPLEQ originated on NetBSD. For compatibility reasons, some systems provide both sets of macros. glibc provides both STAILQ and SIMPLEQ, which are identical except for a missing SIMPLEQ equivalent to **STAILQ_CONCAT**().

SEE ALSO

circleq(3), *insque*(3), *list*(3), *slist*(3), *stailq*(3), *tailq*(3)

NAME

random − overview of interfaces for obtaining randomness

DESCRIPTION

The kernel random-number generator relies on entropy gathered from device drivers and other sources of environmental noise to seed a cryptographically secure pseudorandom number generator (CSPRNG). It is designed for security, rather than speed.

The following interfaces provide access to output from the kernel CSPRNG:

- The */dev/urandom* and */dev/random* devices, both described in *random*(4). These devices have been present on Linux since early times, and are also available on many other systems.
- The Linux-specific *getrandom*(2) system call, available since Linux 3.17. This system call provides access either to the same source as */dev/urandom* (called the *urandom* source in this page) or to the same source as */dev/random* (called the *random* source in this page). The default is the *urandom* source; the *random* source is selected by specifying the **GRND_RANDOM** flag to the system call. (The *getentropy*(3) function provides a slightly more portable interface on top of *getrandom*(2).)

Initialization of the entropy pool

The kernel collects bits of entropy from the environment. When a sufficient number of random bits has been collected, the entropy pool is considered to be initialized.

Choice of random source

Unless you are doing long-term key generation (and most likely not even then), you probably shouldn't be reading from the */dev/random* device or employing *getrandom*(2) with the **GRND_RANDOM** flag. Instead, either read from the */dev/urandom* device or employ *getrandom*(2) without the **GRND_RANDOM** flag. The cryptographic algorithms used for the *urandom* source are quite conservative, and so should be sufficient for all purposes.

The disadvantage of **GRND_RANDOM** and reads from */dev/random* is that the operation can block for an indefinite period of time. Furthermore, dealing with the partially fulfilled requests that can occur when using **GRND_RANDOM** or when reading from */dev/random* increases code complexity.

Monte Carlo and other probabilistic sampling applications

Using these interfaces to provide large quantities of data for Monte Carlo simulations or other programs/algorithms which are doing probabilistic sampling will be slow. Furthermore, it is unnecessary, because such applications do not need cryptographically secure random numbers. Instead, use the interfaces described in this page to obtain a small amount of data to seed a user-space pseudorandom number generator for use by such applications.

Comparison between getrandom, /dev/urandom, and /dev/random

The following table summarizes the behavior of the various interfaces that can be used to obtain randomness. **GRND_NONBLOCK** is a flag that can be used to control the blocking behavior of *getrandom*(2). The final column of the table considers the case that can occur in early boot time when the entropy pool is not yet initialized.

Interface	**Pool**	**Blocking behavior**	**Behavior when pool is not yet ready**
/dev/random	Blocking pool	If entropy too low, blocks until there is enough entropy again	Blocks until enough entropy gathered
/dev/urandom	CSPRNG output	Never blocks	Returns output from uninitialized CSPRNG (may be low entropy and unsuitable for cryptography)
getrandom()	Same as */dev/urandom*	Does not block once is pool ready	Blocks until pool ready
getrandom() **GRND_RANDOM**	Same as */dev/random*	If entropy too low, blocks until there is enough entropy again	Blocks until pool ready
getrandom() **GRND_NONBLOCK**	Same as */dev/urandom*	Does not block once is pool ready	**EAGAIN**
getrandom() **GRND_RANDOM** + **GRND_NONBLOCK**	Same as */dev/random*	**EAGAIN** if not enough entropy available	**EAGAIN**

Generating cryptographic keys

The amount of seed material required to generate a cryptographic key equals the effective key size of the key. For example, a 3072-bit RSA or Diffie-Hellman private key has an effective key size of 128 bits (it requires about 2^128 operations to break) so a key generator needs only 128 bits (16 bytes) of seed material from */dev/random*.

While some safety margin above that minimum is reasonable, as a guard against flaws in the CSPRNG algorithm, no cryptographic primitive available today can hope to promise more than 256 bits of security, so if any program reads more than 256 bits (32 bytes) from the kernel random pool per invocation, or per reasonable reseed interval (not less than one minute), that should be taken as a sign that its cryptography is *not* skillfully implemented.

SEE ALSO

getrandom(2), *getauxval*(3), *getentropy*(3), *random*(4), *urandom*(4), *signal*(7)

NAME

raw − Linux IPv4 raw sockets

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
raw_socket = socket(AF_INET, SOCK_RAW, int *protocol***);**

DESCRIPTION

Raw sockets allow new IPv4 protocols to be implemented in user space. A raw socket receives or sends the raw datagram not including link level headers.

The IPv4 layer generates an IP header when sending a packet unless the **IP_HDRINCL** socket option is enabled on the socket. When it is enabled, the packet must contain an IP header. For receiving, the IP header is always included in the packet.

In order to create a raw socket, a process must have the **CAP_NET_RAW** capability in the user namespace that governs its network namespace.

All packets or errors matching the *protocol* number specified for the raw socket are passed to this socket. For a list of the allowed protocols, see the IANA list of assigned protocol numbers at and *getprotobyname*(3).

A protocol of **IPPROTO_RAW** implies enabled **IP_HDRINCL** and is able to send any IP protocol that is specified in the passed header. Receiving of all IP protocols via **IPPROTO_RAW** is not possible using raw sockets.

IP Header fields modified on sending by **IP_HDRINCL**	
IP Checksum	Always filled in
Source Address	Filled in when zero
Packet ID	Filled in when zero
Total Length	Always filled in

If **IP_HDRINCL** is specified and the IP header has a nonzero destination address, then the destination address of the socket is used to route the packet. When **MSG_DONTROUTE** is specified, the destination address should refer to a local interface, otherwise a routing table lookup is done anyway but gatewayed routes are ignored.

If **IP_HDRINCL** isn't set, then IP header options can be set on raw sockets with *setsockopt*(2); see *ip*(7) for more information.

Starting with Linux 2.2, all IP header fields and options can be set using IP socket options. This means raw sockets are usually needed only for new protocols or protocols with no user interface (like ICMP).

When a packet is received, it is passed to any raw sockets which have been bound to its protocol before it is passed to other protocol handlers (e.g., kernel protocol modules).

Address format

For sending and receiving datagrams (**sendto**(2), *recvfrom*(2), and similar), raw sockets use the standard *sockaddr_in* address structure defined in *ip*(7). The *sin_port* field could be used to specify the IP protocol number, but it is ignored for sending in Linux 2.2 and later, and should be always set to 0 (see BUGS). For incoming packets, *sin_port* is set to zero.

Socket options

Raw socket options can be set with *setsockopt*(2) and read with *getsockopt*(2) by passing the **IPPROTO_RAW** family flag.

ICMP_FILTER

Enable a special filter for raw sockets bound to the **IPPROTO_ICMP** protocol. The value has a bit set for each ICMP message type which should be filtered out. The default is to filter no ICMP messages.

In addition, all *ip*(7) **IPPROTO_IP** socket options valid for datagram sockets are supported.

Error handling

Errors originating from the network are passed to the user only when the socket is connected or the **IP_RECVERR** flag is enabled. For connected sockets, only **EMSGSIZE** and **EPROTO** are passed for compatibility. With **IP_RECVERR**, all network errors are saved in the error queue.

ERRORS

EACCES
User tried to send to a broadcast address without having the broadcast flag set on the socket.

EFAULT
An invalid memory address was supplied.

EINVAL
Invalid argument.

EMSGSIZE
Packet too big. Either Path MTU Discovery is enabled (the **IP_MTU_DISCOVER** socket flag) or the packet size exceeds the maximum allowed IPv4 packet size of 64 kB.

EOPNOTSUPP
Invalid flag has been passed to a socket call (like **MSG_OOB**).

EPERM
The user doesn't have permission to open raw sockets. Only processes with an effective user ID of 0 or the **CAP_NET_RAW** attribute may do that.

EPROTO
An ICMP error has arrived reporting a parameter problem.

VERSIONS

IP_RECVERR and **ICMP_FILTER** are new in Linux 2.2. They are Linux extensions and should not be used in portable programs.

Linux 2.0 enabled some bug-to-bug compatibility with BSD in the raw socket code when the **SO_BSDCOMPAT** socket option was set; since Linux 2.2, this option no longer has that effect.

NOTES

By default, raw sockets do path MTU (Maximum Transmission Unit) discovery. This means the kernel will keep track of the MTU to a specific target IP address and return **EMSGSIZE** when a raw packet write exceeds it. When this happens, the application should decrease the packet size. Path MTU discovery can be also turned off using the **IP_MTU_DISCOVER** socket option or the */proc/sys/net/ipv4/ip_no_pmtu_disc* file, see *ip*(7) for details. When turned off, raw sockets will fragment outgoing packets that exceed the interface MTU. However, disabling it is not recommended for performance and reliability reasons.

A raw socket can be bound to a specific local address using the *bind*(2) call. If it isn't bound, all packets with the specified IP protocol are received. In addition, a raw socket can be bound to a specific network device using **SO_BINDTODEVICE**; see *socket*(7).

An **IPPROTO_RAW** socket is send only. If you really want to receive all IP packets, use a *packet*(7) socket with the **ETH_P_IP** protocol. Note that packet sockets don't reassemble IP fragments, unlike raw sockets.

If you want to receive all ICMP packets for a datagram socket, it is often better to use **IP_RECVERR** on that particular socket; see *ip*(7).

Raw sockets may tap all IP protocols in Linux, even protocols like ICMP or TCP which have a protocol module in the kernel. In this case, the packets are passed to both the kernel module and the raw socket(s). This should not be relied upon in portable programs, many other BSD socket implementation have limitations here.

Linux never changes headers passed from the user (except for filling in some zeroed fields as described for **IP_HDRINCL**). This differs from many other implementations of raw sockets.

Raw sockets are generally rather unportable and should be avoided in programs intended to be portable.

Sending on raw sockets should take the IP protocol from *sin_port*; this ability was lost in Linux 2.2. The workaround is to use **IP_HDRINCL**.

BUGS

Transparent proxy extensions are not described.

When the **IP_HDRINCL** option is set, datagrams will not be fragmented and are limited to the interface MTU.

Setting the IP protocol for sending in *sin_port* got lost in Linux 2.2. The protocol that the socket was bound to or that was specified in the initial *socket*(2) call is always used.

SEE ALSO

recvmsg(2), *sendmsg*(2), *capabilities*(7), *ip*(7), *socket*(7)

RFC 1191 for path MTU discovery. **RFC 791** and the <*linux/ip.h*> header file for the IP protocol.

NAME

regex − POSIX.2 regular expressions

DESCRIPTION

Regular expressions ("RE"s), as defined in POSIX.2, come in two forms: modern REs (roughly those of *egrep*(1); POSIX.2 calls these "extended" REs) and obsolete REs (roughly those of *ed*(1); POSIX.2 "basic" REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. POSIX.2 leaves some aspects of RE syntax and semantics open; "†" marks decisions on these aspects that may not be fully portable to other POSIX.2 implementations.

A (modern) RE is one† or more nonempty† *branches*, separated by '|'. It matches anything that matches one of the branches.

A branch is one† or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, and so on.

A piece is an *atom* possibly followed by a single† '*', '+', '?', or *bound*. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a sequence of 0 or 1 matches of the atom.

A *bound* is '{' followed by an unsigned decimal integer, possibly followed by ',' possibly followed by another unsigned decimal integer, always followed by '}'. The integers must lie between 0 and **RE_DUP_MAX** (255†) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *i* and no comma matches a sequence of exactly *i* matches of the atom. An atom followed by a bound containing one integer *i* and a comma matches a sequence of *i* or more matches of the atom. An atom followed by a bound containing two integers *i* and *j* matches a sequence of *i* through *j* (inclusive) matches of the atom.

An atom is a regular expression enclosed in "*()*" (matching a match for the regular expression), an empty set of "*()*" (matching the null string)†, a *bracket expression* (see below), '.' (matching any single character), '^' (matching the null string at the beginning of a line), '$' (matching the null string at the end of a line), a '\' followed by one of the characters "^.*[$()|*+?{\" (matching that character taken as an ordinary character), a '\' followed by any other character† (matching that character taken as an ordinary character, as if the '\' had not been present†), or a single character with no other significance (matching that character). A '{' followed by a character other than a digit is an ordinary character, not the beginning of a bound†. It is illegal to end an RE with '\'.

A *bracket expression* is a list of characters enclosed in "*[]*". It normally matches any single character from the list (but see below). If the list begins with '^', it matches any single character (but see below) *not* from the rest of the list. If two characters in the list are separated by '−', this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence, for example, "*[0−9]*" in ASCII matches any decimal digit. It is illegal† for two ranges to share an endpoint, for example, "*a−c−e*". Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal ']' in the list, make it the first character (following a possible '^'). To include a literal '−', make it the first or last character, or the second endpoint of a range. To use a literal '−' as the first endpoint of a range, enclose it in "*[.*" and "*.]*" to make it a collating element (see below). With the exception of these and some combinations using '[' (see next paragraphs), all other special characters, including '\', lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multicharacter sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in "*[.*" and "*.]*" stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multicharacter collating element can thus match more than one character, for example, if the collating sequence includes a "ch" collating element, then the RE "*[[.ch.]]*c*" matches the first five characters of "chchcc".

Within a bracket expression, a collating element enclosed in "*[=*" and "*=]*" is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were "*[.*" and "*.]*".) For example, if o and ô are the members of an equivalence class, then "*[[=o=]]*", "*[[=ô=]]*", and "*[oô]*" are all synonymous. An equivalence class may not† be an endpoint of a range.

Within a bracket expression, the name of a *character class* enclosed in "*[:*" and "*:]*" stands for the list of all characters belonging to that class. Standard character class names are:

alnum	digit	punct
alpha	graph	space
blank	lower	upper
cntrl	print	xdigit

These stand for the character classes defined in *wctype*(3). A locale may provide others. A character class may not be used as an endpoint of a range.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, "*bb**" matches the three middle characters of "abbbc", "*(wee|week)(knights|nights)*" matches all ten characters of "weeknights", when "*(.*).**" is matched against "abc" the parenthesized subexpression matches all three characters, and when "*(a*)**" is matched against "bc" both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, for example, 'x' becomes "*[xX]*". When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that, for example, "*[x]*" becomes "*[xX]*" and "*[^x]*" becomes "*[^xX]*".

No particular limit is imposed on the length of REs†. Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete ("basic") regular expressions differ in several respects. '|', '+', and '?' are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are "*\{*" and "*\}*", with '{' and '}' by themselves ordinary characters. The parentheses for nested subexpressions are "*\(*" and "*\)*", with '(' and ')' by themselves ordinary characters. '^' is an ordinary character except at the beginning of the RE or† the beginning of a parenthesized subexpression, '$' is an ordinary character except at the end of the RE or† the end of a parenthesized subexpression, and '*' is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading '^').

Finally, there is one new type of atom, a *back reference*: '\' followed by a nonzero decimal digit *d* matches the same sequence of characters matched by the *d*th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that, for example, "*\([bc]\)\1*" matches "bb" or "cc" but not "bc".

BUGS

Having two kinds of REs is a botch.

The current POSIX.2 spec says that ')' is an ordinary character in the absence of an unmatched '('; this was an unintentional result of a wording error, and change is likely. Avoid relying on it.

Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does "*a\(\(b\)*\2\)*d*" match "abbbd"?). Avoid using them.

POSIX.2's specification of case-independent matching is vague. The "one case implies all cases" definition given above is current consensus among implementors as to the right interpretation.

AUTHOR

This page was taken from Henry Spencer's regex package.

SEE ALSO

grep(1), *regex*(3)

POSIX.2, section 2.8 (Regular Expression Notation).

NAME

rtld−audit − auditing API for the dynamic linker

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <link.h>
```

DESCRIPTION

The GNU dynamic linker (run-time linker) provides an auditing API that allows an application to be notified when various dynamic linking events occur. This API is very similar to the auditing interface provided by the Solaris run-time linker. The necessary constants and prototypes are defined by including *<link.h>*.

To use this interface, the programmer creates a shared library that implements a standard set of function names. Not all of the functions need to be implemented: in most cases, if the programmer is not interested in a particular class of auditing event, then no implementation needs to be provided for the corresponding auditing function.

To employ the auditing interface, the environment variable **LD_AUDIT** must be defined to contain a colon-separated list of shared libraries, each of which can implement (parts of) the auditing API. When an auditable event occurs, the corresponding function is invoked in each library, in the order that the libraries are listed.

la_version()

unsigned int la_version(unsigned int *version***);**

This is the only function that *must* be defined by an auditing library: it performs the initial handshake between the dynamic linker and the auditing library. When invoking this function, the dynamic linker passes, in *version*, the highest version of the auditing interface that the linker supports.

A typical implementation of this function simply returns the constant **LAV_CURRENT**, which indicates the version of *<link.h>* that was used to build the audit module. If the dynamic linker does not support this version of the audit interface, it will refuse to activate this audit module. If the function returns zero, the dynamic linker also does not activate this audit module.

In order to enable backwards compatibility with older dynamic linkers, an audit module can examine the *version* argument and return an earlier version than **LAV_CURRENT**, assuming the module can adjust its implementation to match the requirements of the previous version of the audit interface. The **la_version** function should not return the value of *version* without further checks because it could correspond to an interface that does not match the *<link.h>* definitions used to build the audit module.

la_objsearch()

char *la_objsearch(const char ***name*, uintptr_t ***cookie***,**
unsigned int *flag***);**

The dynamic linker invokes this function to inform the auditing library that it is about to search for a shared object. The *name* argument is the filename or pathname that is to be searched for. *cookie* identifies the shared object that initiated the search. *flag* is set to one of the following values:

LA_SER_ORIG This is the original name that is being searched for. Typically, this name comes from an ELF **DT_NEEDED** entry, or is the *filename* argument given to *dlopen*(3).

LA_SER_LIBPATH
name was created using a directory specified in **LD_LIBRARY_PATH**.

LA_SER_RUNPATH
name was created using a directory specified in an ELF **DT_RPATH** or **DT_RUNPATH** list.

LA_SER_CONFIG
name was found via the *ldconfig*(8) cache (*/etc/ld.so.cache*).

LA_SER_DEFAULT
name was found via a search of one of the default directories.

LA_SER_SECURE
name is specific to a secure object (unused on Linux).

As its function result, **la_objsearch**() returns the pathname that the dynamic linker should use for further processing. If NULL is returned, then this pathname is ignored for further processing. If this audit library simply intends to monitor search paths, then *name* should be returned.

la_activity()

void la_activity(uintptr_t **cookie*, unsigned int *flag*);

The dynamic linker calls this function to inform the auditing library that link-map activity is occurring. *cookie* identifies the object at the head of the link map. When the dynamic linker invokes this function, *flag* is set to one of the following values:

LA_ACT_ADD New objects are being added to the link map.

LA_ACT_DELETE Objects are being removed from the link map.

LA_ACT_CONSISTENT
Link-map activity has been completed: the map is once again consistent.

la_objopen()

unsigned int la_objopen(struct link_map **map*, Lmid_t *lmid*,
uintptr_t **cookie*);

The dynamic linker calls this function when a new shared object is loaded. The *map* argument is a pointer to a link-map structure that describes the object. The *lmid* field has one of the following values

LM_ID_BASE Link map is part of the initial namespace.

LM_ID_NEWLM Link map is part of a new namespace requested via *dlmopen*(3).

cookie is a pointer to an identifier for this object. The identifier is provided to later calls to functions in the auditing library in order to identify this object. This identifier is initialized to point to object's link map, but the audit library can change the identifier to some other value that it may prefer to use to identify the object.

As its return value, **la_objopen**() returns a bit mask created by ORing zero or more of the following constants, which allow the auditing library to select the objects to be monitored by **la_symbind***():

LA_FLG_BINDTO
Audit symbol bindings to this object.

LA_FLG_BINDFROM
Audit symbol bindings from this object.

A return value of 0 from **la_objopen**() indicates that no symbol bindings should be audited for this object.

la_objclose()

unsigned int la_objclose(uintptr_t **cookie*);

The dynamic linker invokes this function after any finalization code for the object has been executed, before the object is unloaded. The *cookie* argument is the identifier obtained from a previous invocation of **la_objopen**().

In the current implementation, the value returned by **la_objclose**() is ignored.

la_preinit()

void la_preinit(uintptr_t **cookie*);

The dynamic linker invokes this function after all shared objects have been loaded, before control is passed to the application (i.e., before calling *main*()). Note that *main*() may still later dynamically load objects using *dlopen*(3).

la_symbind*()

uintptr_t la_symbind32(Elf32_Sym *_sym_, unsigned int _ndx_,
uintptr_t *_refcook_, uintptr_t *_defcook_,
unsigned int * _flags_, const char *_symname_);
uintptr_t la_symbind64(Elf64_Sym *_sym_, unsigned int _ndx_,
uintptr_t *_refcook_, uintptr_t *_defcook_,
unsigned int * _flags_, const char *_symname_);

The dynamic linker invokes one of these functions when a symbol binding occurs between two shared objects that have been marked for auditing notification by **la_objopen**(). The **la_symbind32**() function is employed on 32-bit platforms; the **la_symbind64**() function is employed on 64-bit platforms.

The *sym* argument is a pointer to a structure that provides information about the symbol being bound. The structure definition is shown in <*elf.h*>. Among the fields of this structure, *st_value* indicates the address to which the symbol is bound.

The *ndx* argument gives the index of the symbol in the symbol table of the bound shared object.

The *refcook* argument identifies the shared object that is making the symbol reference; this is the same identifier that is provided to the **la_objopen**() function that returned **LA_FLG_BINDFROM**. The *defcook* argument identifies the shared object that defines the referenced symbol; this is the same identifier that is provided to the **la_objopen**() function that returned **LA_FLG_BINDTO**.

The *symname* argument points a string containing the name of the symbol.

The *flags* argument is a bit mask that both provides information about the symbol and can be used to modify further auditing of this PLT (Procedure Linkage Table) entry. The dynamic linker may supply the following bit values in this argument:

LA_SYMB_DLSYM The binding resulted from a call to *dlsym*(3).

LA_SYMB_ALTVALUE
A previous **la_symbind***() call returned an alternate value for this symbol.

By default, if the auditing library implements **la_pltenter**() and **la_pltexit**() functions (see below), then these functions are invoked, after **la_symbind**(), for PLT entries, each time the symbol is referenced. The following flags can be ORed into **flags* to change this default behavior:

LA_SYMB_NOPLTENTER
Don't call **la_pltenter**() for this symbol.

LA_SYMB_NOPLTEXIT
Don't call **la_pltexit**() for this symbol.

The return value of **la_symbind32**() and **la_symbind64**() is the address to which control should be passed after the function returns. If the auditing library is simply monitoring symbol bindings, then it should return *sym->st_value*. A different value may be returned if the library wishes to direct control to an alternate location.

la_pltenter()
The precise name and argument types for this function depend on the hardware platform. (The appropriate definition is supplied by <*link.h*>.) Here is the definition for x86-32:

Elf32_Addr la_i86_gnu_pltenter(Elf32_Sym *_sym_, unsigned int _ndx_,
uintptr_t *_refcook_, uintptr_t *_defcook_,
La_i86_regs *_regs_, unsigned int * _flags_,
const char *_symname_, long * _framesizep_);

This function is invoked just before a PLT entry is called, between two shared objects that have been marked for binding notification.

The *sym*, *ndx*, *refcook*, *defcook*, and *symname* are as for **la_symbind***().

The *regs* argument points to a structure (defined in <*link.h*>) containing the values of registers to be used for the call to this PLT entry.

The *flags* argument points to a bit mask that conveys information about, and can be used to modify subsequent auditing of, this PLT entry, as for **la_symbind***().

The *framesizep* argument points to a *long int* buffer that can be used to explicitly set the frame size used for the call to this PLT entry. If different **la_pltenter**() invocations for this symbol return different values, then the maximum returned value is used. The **la_pltexit**() function is called only if this buffer is explicitly set to a suitable value.

The return value of **la_pltenter**() is as for **la_symbind***().

la_pltexit()

The precise name and argument types for this function depend on the hardware platform. (The appropriate definition is supplied by <*link.h*>.) Here is the definition for x86-32:

unsigned int la_i86_gnu_pltexit(Elf32_Sym ***sym*, unsigned int** *ndx***,**
uintptr_t ***refcook*, uintptr_t ***defcook***,**
const La_i86_regs ***inregs*, La_i86_retval ***outregs***,**
const char ***symname*);**

This function is called when a PLT entry, made between two shared objects that have been marked for binding notification, returns. The function is called just before control returns to the caller of the PLT entry.

The *sym*, *ndx*, *refcook*, *defcook*, and *symname* are as for **la_symbind***().

The *inregs* argument points to a structure (defined in <*link.h*>) containing the values of registers used for the call to this PLT entry. The *outregs* argument points to a structure (defined in <*link.h*>) containing return values for the call to this PLT entry. These values can be modified by the caller, and the changes will be visible to the caller of the PLT entry.

In the current GNU implementation, the return value of **la_pltexit**() is ignored.

VERSIONS

This API is very similar to the Solaris API described in the Solaris *Linker and Libraries Guide*, in the chapter *Runtime Linker Auditing Interface*.

STANDARDS

None.

NOTES

Note the following differences from the Solaris dynamic linker auditing API:

- The Solaris **la_objfilter**() interface is not supported by the GNU implementation.
- The Solaris **la_symbind32**() and **la_pltexit**() functions do not provide a *symname* argument.
- The Solaris **la_pltexit**() function does not provide *inregs* and *outregs* arguments (but does provide a *retval* argument with the function return value).

BUGS

In glibc versions up to and include 2.9, specifying more than one audit library in **LD_AUDIT** results in a run-time crash. This is reportedly fixed in glibc 2.10.

EXAMPLES

```
#include <link.h>
#include <stdio.h>

unsigned int
la_version(unsigned int version)
{
    printf("la_version(): version = %u; LAV_CURRENT = %u\n",
            version, LAV_CURRENT);

    return LAV_CURRENT;
}

char *
la_objsearch(const char *name, uintptr_t *cookie, unsigned int flag)
{
    printf("la_objsearch(): name = %s; cookie = %p", name, cookie);
    printf("; flag = %s\n",
```

```
            (flag == LA_SER_ORIG) ?    "LA_SER_ORIG" :
            (flag == LA_SER_LIBPATH) ? "LA_SER_LIBPATH" :
            (flag == LA_SER_RUNPATH) ? "LA_SER_RUNPATH" :
            (flag == LA_SER_DEFAULT) ? "LA_SER_DEFAULT" :
            (flag == LA_SER_CONFIG) ?  "LA_SER_CONFIG" :
            (flag == LA_SER_SECURE) ?  "LA_SER_SECURE" :
            "???");

    return name;
}

void
la_activity (uintptr_t *cookie, unsigned int flag)
{
    printf("la_activity(): cookie = %p; flag = %s\n", cookie,
            (flag == LA_ACT_CONSISTENT) ? "LA_ACT_CONSISTENT" :
            (flag == LA_ACT_ADD) ?        "LA_ACT_ADD" :
            (flag == LA_ACT_DELETE) ?     "LA_ACT_DELETE" :
            "???");
}

unsigned int
la_objopen(struct link_map *map, Lmid_t lmid, uintptr_t *cookie)
{
    printf("la_objopen(): loading \"%s\"; lmid = %s; cookie=%p\n",
            map->l_name,
            (lmid == LM_ID_BASE) ?  "LM_ID_BASE" :
            (lmid == LM_ID_NEWLM) ? "LM_ID_NEWLM" :
            "???",
            cookie);

    return LA_FLG_BINDTO | LA_FLG_BINDFROM;
}

unsigned int
la_objclose (uintptr_t *cookie)
{
    printf("la_objclose(): %p\n", cookie);

    return 0;
}

void
la_preinit(uintptr_t *cookie)
{
    printf("la_preinit(): %p\n", cookie);
}

uintptr_t
la_symbind32(Elf32_Sym *sym, unsigned int ndx, uintptr_t *refcook,
        uintptr_t *defcook, unsigned int *flags, const char *symname)
{
    printf("la_symbind32(): symname = %s; sym->st_value = %p\n",
            symname, sym->st_value);
    printf("        ndx = %u; flags = %#x", ndx, *flags);
    printf("; refcook = %p; defcook = %p\n", refcook, defcook);

    return sym->st_value;
}
```

```
uintptr_t
la_symbind64(Elf64_Sym *sym, unsigned int ndx, uintptr_t *refcook,
        uintptr_t *defcook, unsigned int *flags, const char *symname)
{
    printf("la_symbind64(): symname = %s; sym->st_value = %p\n",
            symname, sym->st_value);
    printf("        ndx = %u; flags = %#x", ndx, *flags);
    printf("; refcook = %p; defcook = %p\n", refcook, defcook);

    return sym->st_value;
}

Elf32_Addr
la_i86_gnu_pltenter(Elf32_Sym *sym, unsigned int ndx,
        uintptr_t *refcook, uintptr_t *defcook, La_i86_regs *regs,
        unsigned int *flags, const char *symname, long *framesizep)
{
    printf("la_i86_gnu_pltenter(): %s (%p)\n", symname, sym->st_value);

    return sym->st_value;
}
```

SEE ALSO

ldd(1), *dlopen*(3), *ld.so*(8), *ldconfig*(8)

NAME

rtnetlink − Linux routing socket

SYNOPSIS

#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>

rtnetlink_socket = socket(AF_NETLINK, int *socket_type***, NETLINK_ROUTE);**

DESCRIPTION

Rtnetlink allows the kernel's routing tables to be read and altered. It is used within the kernel to communicate between various subsystems, though this usage is not documented here, and for communication with user-space programs. Network routes, IP addresses, link parameters, neighbor setups, queueing disciplines, traffic classes and packet classifiers may all be controlled through **NETLINK_ROUTE** sockets. It is based on netlink messages; see *netlink*(7) for more information.

Routing attributes

Some rtnetlink messages have optional attributes after the initial header:

```
struct rtattr {
    unsigned short rta_len;    /* Length of option */
    unsigned short rta_type;   /* Type of option */
    /* Data follows */
};
```

These attributes should be manipulated using only the RTA_* macros or libnetlink, see *rtnetlink*(3).

Messages

Rtnetlink consists of these message types (in addition to standard netlink messages):

RTM_NEWLINK
RTM_DELLINK
RTM_GETLINK

Create, remove, or get information about a specific network interface. These messages contain an *ifinfomsg* structure followed by a series of *rtattr* structures.

```
struct ifinfomsg {
    unsigned char  ifi_family; /* AF_UNSPEC */
    unsigned short ifi_type;   /* Device type */
    int            ifi_index;  /* Interface index */
    unsigned int   ifi_flags;  /* Device flags  */
    unsigned int   ifi_change; /* change mask */
};
```

ifi_flags contains the device flags, see *netdevice*(7); *ifi_index* is the unique interface index (since Linux 3.7, it is possible to feed a nonzero value with the **RTM_NEWLINK** message, thus creating a link with the given *ifindex*); *ifi_change* is reserved for future use and should be always set to 0xFFFFFFFF.

Routing attributes

rta_type	Value type	Description
IFLA_UNSPEC	-	unspecified
IFLA_ADDRESS	hardware address	interface L2 address
IFLA_BROADCAST	hardware address	L2 broadcast address
IFLA_IFNAME	asciiz string	Device name
IFLA_MTU	unsigned int	MTU of the device
IFLA_LINK	int	Link type
IFLA_QDISC	asciiz string	Queueing discipline
IFLA_STATS	see below	Interface Statistics
IFLA_PERM_ADDRESS	hardware address	hardware address provided by device (since Linux 5.5)

The value type for **IFLA_STATS** is *struct rtnl_link_stats* (*struct net_device_stats* in Linux 2.4 and earlier).

RTM_NEWADDR
RTM_DELADDR
RTM_GETADDR

Add, remove, or receive information about an IP address associated with an interface. In Linux 2.2, an interface can carry multiple IP addresses, this replaces the alias device concept in Linux 2.0. In Linux 2.2, these messages support IPv4 and IPv6 addresses. They contain an *ifaddrmsg* structure, optionally followed by *rtattr* routing attributes.

```
struct ifaddrmsg {
    unsigned char ifa_family;    /* Address type */
    unsigned char ifa_prefixlen; /* Prefixlength of address */
    unsigned char ifa_flags;     /* Address flags */
    unsigned char ifa_scope;     /* Address scope */
    unsigned int  ifa_index;     /* Interface index */
};
```

ifa_family is the address family type (currently **AF_INET** or **AF_INET6**), *ifa_prefixlen* is the length of the address mask of the address if defined for the family (like for IPv4), *ifa_scope* is the address scope, *ifa_index* is the interface index of the interface the address is associated with. *ifa_flags* is a flag word of **IFA_F_SECONDARY** for secondary address (old alias interface), **IFA_F_PERMANENT** for a permanent address set by the user and other undocumented flags.

Attributes

rta_type	Value type	Description
IFA_UNSPEC	-	unspecified
IFA_ADDRESS	raw protocol address	interface address
IFA_LOCAL	raw protocol address	local address
IFA_LABEL	asciiz string	name of the interface
IFA_BROADCAST	raw protocol address	broadcast address
IFA_ANYCAST	raw protocol address	anycast address
IFA_CACHEINFO	struct ifa_cacheinfo	Address information

RTM_NEWROUTE
RTM_DELROUTE
RTM_GETROUTE

Create, remove, or receive information about a network route. These messages contain an *rtmsg* structure with an optional sequence of *rtattr* structures following. For **RTM_GETROUTE**, setting *rtm_dst_len* and *rtm_src_len* to 0 means you get all entries for the specified routing table. For the other fields, except *rtm_table* and *rtm_protocol*, 0 is the wildcard.

```
struct rtmsg {
    unsigned char rtm_family;   /* Address family of route */
    unsigned char rtm_dst_len;  /* Length of destination */
    unsigned char rtm_src_len;  /* Length of source */
    unsigned char rtm_tos;      /* TOS filter */
    unsigned char rtm_table;    /* Routing table ID;
                                   see RTA_TABLE below */
    unsigned char rtm_protocol; /* Routing protocol; see below */
    unsigned char rtm_scope;    /* See below */
    unsigned char rtm_type;     /* See below */

    unsigned int  rtm_flags;
};
```

rtm_type	Route type
RTN_UNSPEC	unknown route
RTN_UNICAST	a gateway or direct route
RTN_LOCAL	a local interface route
RTN_BROADCAST	a local broadcast route (sent as a broadcast)

RTN_ANYCAST	a local broadcast route (sent as a unicast)
RTN_MULTICAST	a multicast route
RTN_BLACKHOLE	a packet dropping route
RTN_UNREACHABLE	an unreachable destination
RTN_PROHIBIT	a packet rejection route
RTN_THROW	continue routing lookup in another table
RTN_NAT	a network address translation rule
RTN_XRESOLVE	refer to an external resolver (not implemented)

rtm_protocol	Route origin
RTPROT_UNSPEC	unknown
RTPROT_REDIRECT	by an ICMP redirect (currently unused)
RTPROT_KERNEL	by the kernel
RTPROT_BOOT	during boot
RTPROT_STATIC	by the administrator

Values larger than **RTPROT_STATIC** are not interpreted by the kernel, they are just for user information. They may be used to tag the source of a routing information or to distinguish between multiple routing daemons. See <*linux/rtnetlink.h*> for the routing daemon identifiers which are already assigned.

rtm_scope is the distance to the destination:

RT_SCOPE_UNIVERSE	global route
RT_SCOPE_SITE	interior route in the local autonomous system
RT_SCOPE_LINK	route on this link
RT_SCOPE_HOST	route on the local host
RT_SCOPE_NOWHERE	destination doesn't exist

The values between **RT_SCOPE_UNIVERSE** and **RT_SCOPE_SITE** are available to the user.

The *rtm_flags* have the following meanings:

RTM_F_NOTIFY	if the route changes, notify the user via rtnetlink
RTM_F_CLONED	route is cloned from another route
RTM_F_EQUALIZE	a multipath equalizer (not yet implemented)

rtm_table specifies the routing table

RT_TABLE_UNSPEC	an unspecified routing table
RT_TABLE_DEFAULT	the default table
RT_TABLE_MAIN	the main table
RT_TABLE_LOCAL	the local table

The user may assign arbitrary values between **RT_TABLE_UNSPEC** and **RT_TABLE_DEFAULT**.

Attributes

rta_type	Value type	Description
RTA_UNSPEC	-	ignored
RTA_DST	protocol address	Route destination address
RTA_SRC	protocol address	Route source address
RTA_IIF	int	Input interface index
RTA_OIF	int	Output interface index
RTA_GATEWAY	protocol address	The gateway of the route
RTA_PRIORITY	int	Priority of route
RTA_PREFSRC	protocol address	Preferred source address
RTA_METRICS	int	Route metric
RTA_MULTIPATH		Multipath nexthop data br (see below).
RTA_PROTOINFO		No longer used
RTA_FLOW	int	Route realm
RTA_CACHEINFO	struct rta_cacheinfo	(see linux/rtnetlink.h)
RTA_SESSION		No longer used
RTA_MP_ALGO		No longer used
RTA_TABLE	int	Routing table ID; if set, rtm_table is ignored
RTA_MARK	int	
RTA_MFC_STATS	struct rta_mfc_stats	(see linux/rtnetlink.h)
RTA_VIA	struct rtvia	Gateway in different AF (see below)
RTA_NEWDST	protocol address	Change packet destination address
RTA_PREF	char	RFC4191 IPv6 router preference (see below)
RTA_ENCAP_TYPE	short	Encapsulation type for lwtunnels (see below)
RTA_ENCAP		Defined by RTA_ENCAP_TYPE
RTA_EXPIRES	int	Expire time for IPv6 routes (in seconds)

RTA_MULTIPATH contains several packed instances of *struct rtnexthop* together with nested RTAs (**RTA_GATEWAY**):

```
struct rtnexthop {
    unsigned short rtnh_len;     /* Length of struct + length
                                    of RTAs */
    unsigned char  rtnh_flags;   /* Flags (see
                                    linux/rtnetlink.h) */
    unsigned char  rtnh_hops;    /* Nexthop priority */
    int            rtnh_ifindex; /* Interface index for this
                                    nexthop */
}
```

There exist a bunch of **RTNH_*** macros similar to **RTA_*** and **NLHDR_*** macros useful to handle these structures.

```
struct rtvia {
    unsigned short rtvia_family;
    unsigned char  rtvia_addr[0];
};
```

rtvia_addr is the address, *rtvia_family* is its family type.

RTA_PREF may contain values **ICMPV6_ROUTER_PREF_LOW**, **ICMPV6_ROUTER_PREF_MEDIUM**, and **ICMPV6_ROUTER_PREF_HIGH** defined incw <*linux/icmpv6.h*>.

RTA_ENCAP_TYPE may contain values **LWTUNNEL_ENCAP_MPLS**, **LWTUNNEL_ENCAP_IP**, **LWTUNNEL_ENCAP_ILA**, or **LWTUNNEL_ENCAP_IP6** defined in

<*linux/lwtunnel.h*>.

Fill these values in!

RTM_NEWNEIGH
RTM_DELNEIGH
RTM_GETNEIGH

Add, remove, or receive information about a neighbor table entry (e.g., an ARP entry). The message contains an *ndmsg* structure.

```
struct ndmsg {
    unsigned char ndm_family;
    int           ndm_ifindex;  /* Interface index */
    __u16         ndm_state;    /* State */
    __u8          ndm_flags;    /* Flags */
    __u8          ndm_type;
};

struct nda_cacheinfo {
    __u32         ndm_confirmed;
    __u32         ndm_used;
    __u32         ndm_updated;
    __u32         ndm_refcnt;
};
```

ndm_state is a bit mask of the following states:

NUD_INCOMPLETE a currently resolving cache entry
NUD_REACHABLE a confirmed working cache entry
NUD_STALE an expired cache entry
NUD_DELAY an entry waiting for a timer
NUD_PROBE a cache entry that is currently reprobed
NUD_FAILED an invalid cache entry
NUD_NOARP a device with no destination cache
NUD_PERMANENT a static entry

Valid *ndm_flags* are:

NTF_PROXY a proxy arp entry
NTF_ROUTER an IPv6 router

The *rtattr* struct has the following meanings for the *rta_type* field:

NDA_UNSPEC unknown type
NDA_DST a neighbor cache n/w layer destination address
NDA_LLADDR a neighbor cache link layer address
NDA_CACHEINFO cache statistics

If the *rta_type* field is **NDA_CACHEINFO**, then a *struct nda_cacheinfo* header follows.

RTM_NEWRULE
RTM_DELRULE
RTM_GETRULE

Add, delete, or retrieve a routing rule. Carries a *struct rtmsg*

RTM_NEWQDISC
RTM_DELQDISC
RTM_GETQDISC

Add, remove, or get a queueing discipline. The message contains a *struct tcmsg* and may be followed by a series of attributes.

```
struct tcmsg {
    unsigned char    tcm_family;
    int              tcm_ifindex;   /* interface index */
    __u32            tcm_handle;    /* Qdisc handle */
    __u32            tcm_parent;    /* Parent qdisc */
    __u32            tcm_info;
};
```

Attributes

rta_type	Value type	Description
TCA_UNSPEC	-	unspecified
TCA_KIND	asciiz string	Name of queueing discipline
TCA_OPTIONS	byte sequence	Qdisc-specific options follow
TCA_STATS	struct tc_stats	Qdisc statistics
TCA_XSTATS	qdisc-specific	Module-specific statistics
TCA_RATE	struct tc_estimator	Rate limit

In addition, various other qdisc-module-specific attributes are allowed. For more information see the appropriate include files.

RTM_NEWTCLASS
RTM_DELTCLASS
RTM_GETTCLASS
Add, remove, or get a traffic class. These messages contain a *struct tcmsg* as described above.

RTM_NEWTFILTER
RTM_DELTFILTER
RTM_GETTFILTER
Add, remove, or receive information about a traffic filter. These messages contain a *struct tcmsg* as described above.

VERSIONS

rtnetlink is a new feature of Linux 2.2.

BUGS

This manual page is incomplete.

SEE ALSO

cmsg(3), *rtnetlink*(3), *ip*(7), *netlink*(7)

NAME

sched − overview of CPU scheduling

DESCRIPTION

Since Linux 2.6.23, the default scheduler is CFS, the "Completely Fair Scheduler". The CFS scheduler replaced the earlier "O(1)" scheduler.

API summary

Linux provides the following system calls for controlling the CPU scheduling behavior, policy, and priority of processes (or, more precisely, threads).

nice(2) Set a new nice value for the calling thread, and return the new nice value.

getpriority(2)
: Return the nice value of a thread, a process group, or the set of threads owned by a specified user.

setpriority(2)
: Set the nice value of a thread, a process group, or the set of threads owned by a specified user.

sched_setscheduler(2)
: Set the scheduling policy and parameters of a specified thread.

sched_getscheduler(2)
: Return the scheduling policy of a specified thread.

sched_setparam(2)
: Set the scheduling parameters of a specified thread.

sched_getparam(2)
: Fetch the scheduling parameters of a specified thread.

sched_get_priority_max(2)
: Return the maximum priority available in a specified scheduling policy.

sched_get_priority_min(2)
: Return the minimum priority available in a specified scheduling policy.

sched_rr_get_interval(2)
: Fetch the quantum used for threads that are scheduled under the "round-robin" scheduling policy.

sched_yield(2)
: Cause the caller to relinquish the CPU, so that some other thread be executed.

sched_setaffinity(2)
: (Linux-specific) Set the CPU affinity of a specified thread.

sched_getaffinity(2)
: (Linux-specific) Get the CPU affinity of a specified thread.

sched_setattr(2)
: Set the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of *sched_setscheduler*(2) and *sched_setparam*(2).

sched_getattr(2)
: Fetch the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of *sched_getscheduler*(2) and *sched_getparam*(2).

Scheduling policies

The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a *static* scheduling priority, *sched_priority*. The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all threads on the system.

For threads scheduled under one of the normal scheduling policies (**SCHED_OTHER**, **SCHED_IDLE**, **SCHED_BATCH**), *sched_priority* is not used in scheduling decisions (it must be specified as 0).

Processes scheduled under one of the real-time policies (**SCHED_FIFO**, **SCHED_RR**) have a *sched_priority* value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always

have higher priority than normal threads.) Note well: POSIX.1 requires an implementation to support only a minimum 32 distinct priority levels for the real-time policies, and some systems supply just this minimum. Portable programs should use *sched_get_priority_min*(2) and *sched_get_priority_max*(2) to find the range of priorities supported for a particular policy.

Conceptually, the scheduler maintains a list of runnable threads for each possible *sched_priority* value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and selects the thread at the head of this list.

A thread's scheduling policy determines where it will be inserted into the list of threads with equal static priority and how it will move inside this list.

All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. The scheduling policy determines the ordering only within the list of runnable threads with equal static priority.

SCHED_FIFO: First in-first out scheduling

SCHED_FIFO can be used only with static priorities higher than 0, which means that when a **SCHED_FIFO** thread becomes runnable, it will always immediately preempt any currently running **SCHED_OTHER**, **SCHED_BATCH**, or **SCHED_IDLE** thread. **SCHED_FIFO** is a simple scheduling algorithm without time slicing. For threads scheduled under the **SCHED_FIFO** policy, the following rules apply:

- A running **SCHED_FIFO** thread that has been preempted by another thread of higher priority will stay at the head of the list for its priority and will resume execution as soon as all threads of higher priority are blocked again.
- When a blocked **SCHED_FIFO** thread becomes runnable, it will be inserted at the end of the list for its priority.
- If a call to *sched_setscheduler*(2), *sched_setparam*(2), *sched_setattr*(2), *pthread_setschedparam*(3), or *pthread_setschedprio*(3) changes the priority of the running or runnable **SCHED_FIFO** thread identified by *pid* the effect on the thread's position in the list depends on the direction of the change to the thread's priority:

  (a) If the thread's priority is raised, it is placed at the end of the list for its new priority. As a consequence, it may preempt a currently running thread with the same priority.

  (b) If the thread's priority is unchanged, its position in the run list is unchanged.

  (c) If the thread's priority is lowered, it is placed at the front of the list for its new priority.

  According to POSIX.1-2008, changes to a thread's priority (or policy) using any mechanism other than *pthread_setschedprio*(3) should result in the thread being placed at the end of the list for its priority.
- A thread calling *sched_yield*(2) will be put at the end of the list.

No other events will move a thread scheduled under the **SCHED_FIFO** policy in the wait list of runnable threads with equal static priority.

A **SCHED_FIFO** thread runs until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls *sched_yield*(2).

SCHED_RR: Round-robin scheduling

SCHED_RR is a simple enhancement of **SCHED_FIFO**. Everything described above for **SCHED_FIFO** also applies to **SCHED_RR**, except that each thread is allowed to run only for a maximum time quantum. If a **SCHED_RR** thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A **SCHED_RR** thread that has been preempted by a higher priority thread and subsequently resumes execution as a running thread will complete the unexpired portion of its round-robin time quantum. The length of the time quantum can be retrieved using *sched_rr_get_interval*(2).

SCHED_DEADLINE: Sporadic task model deadline scheduling

Since Linux 3.14, Linux provides a deadline scheduling policy (**SCHED_DEADLINE**). This policy is currently implemented using GEDF (Global Earliest Deadline First) in conjunction with CBS (Constant Bandwidth Server). To set and fetch this policy and associated attributes, one must use the Linux-

specific *sched_setattr*(2) and *sched_getattr*(2) system calls.

A sporadic task is one that has a sequence of jobs, where each job is activated at most once per period. Each job also has a *relative deadline*, before which it should finish execution, and a *computation time*, which is the CPU time necessary for executing the job. The moment when a task wakes up because a new job has to be executed is called the *arrival time* (also referred to as the request time or release time). The *start time* is the time at which a task starts its execution. The *absolute deadline* is thus obtained by adding the relative deadline to the arrival time.

The following diagram clarifies these terms:

```
arrival/wakeup                    absolute deadline
     |    start time                    |
     |        |                        |
     v        v                        v
-----x--------xooooooooooooooooo--------x--------x---
              |<- comp. time ->|
     |<------- relative deadline ------>|
     |<-------------- period ------------------>|
```

When setting a **SCHED_DEADLINE** policy for a thread using *sched_setattr*(2), one can specify three parameters: *Runtime*, *Deadline*, and *Period*. These parameters do not necessarily correspond to the aforementioned terms: usual practice is to set Runtime to something bigger than the average computation time (or worst-case execution time for hard real-time tasks), Deadline to the relative deadline, and Period to the period of the task. Thus, for **SCHED_DEADLINE** scheduling, we have:

```
arrival/wakeup                    absolute deadline
     |    start time                    |
     |        |                        |
     v        v                        v
-----x--------xooooooooooooooooo--------x--------x---
              |<-- Runtime ------->|
     |<----------- Deadline ----------->|
     |<-------------- Period ------------------>|
```

The three deadline-scheduling parameters correspond to the *sched_runtime*, *sched_deadline*, and *sched_period* fields of the *sched_attr* structure; see *sched_setattr*(2). These fields express values in nanoseconds. If *sched_period* is specified as 0, then it is made the same as *sched_deadline*.

The kernel requires that:

```
sched_runtime <= sched_deadline <= sched_period
```

In addition, under the current implementation, all of the parameter values must be at least 1024 (i.e., just over one microsecond, which is the resolution of the implementation), and less than 2^63. If any of these checks fails, *sched_setattr*(2) fails with the error **EINVAL**.

The CBS guarantees non-interference between tasks, by throttling threads that attempt to over-run their specified Runtime.

To ensure deadline scheduling guarantees, the kernel must prevent situations where the set of **SCHED_DEADLINE** threads is not feasible (schedulable) within the given constraints. The kernel thus performs an admittance test when setting or changing **SCHED_DEADLINE** policy and attributes. This admission test calculates whether the change is feasible; if it is not, *sched_setattr*(2) fails with the error **EBUSY**.

For example, it is required (but not necessarily sufficient) for the total utilization to be less than or equal to the total number of CPUs available, where, since each thread can maximally run for Runtime per Period, that thread's utilization is its Runtime divided by its Period.

In order to fulfill the guarantees that are made when a thread is admitted to the **SCHED_DEADLINE** policy, **SCHED_DEADLINE** threads are the highest priority (user controllable) threads in the system; if any **SCHED_DEADLINE** thread is runnable, it will preempt any thread scheduled under one of the other policies.

A call to *fork*(2) by a thread scheduled under the **SCHED_DEADLINE** policy fails with the error **EAGAIN**, unless the thread has its reset-on-fork flag set (see below).

A **SCHED_DEADLINE** thread that calls *sched_yield*(2) will yield the current job and wait for a new period to begin.

SCHED_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can be used at only static priority 0 (i.e., threads under real-time policies always have priority over **SCHED_OTHER** processes). **SCHED_OTHER** is the standard Linux time-sharing scheduler that is intended for all threads that do not require the special real-time mechanisms.

The thread to run is chosen from the static priority 0 list based on a *dynamic* priority that is determined only inside this list. The dynamic priority is based on the nice value (see below) and is increased for each time quantum the thread is ready to run, but denied to run by the scheduler. This ensures fair progress among all **SCHED_OTHER** threads.

In the Linux kernel source code, the **SCHED_OTHER** policy is actually named **SCHED_NORMAL**.

The nice value

The nice value is an attribute that can be used to influence the CPU scheduler to favor or disfavor a process in scheduling decisions. It affects the scheduling of **SCHED_OTHER** and **SCHED_BATCH** (see below) processes. The nice value can be modified using *nice*(2), *setpriority*(2), or *sched_setattr*(2).

According to POSIX.1, the nice value is a per-process attribute; that is, the threads in a process should share a nice value. However, on Linux, the nice value is a per-thread attribute: different threads in the same process may have different nice values.

The range of the nice value varies across UNIX systems. On modern Linux, the range is −20 (high priority) to +19 (low priority). On some other systems, the range is −20..20. Very early Linux kernels (before Linux 2.0) had the range −infinity..15.

The degree to which the nice value affects the relative scheduling of **SCHED_OTHER** processes likewise varies across UNIX systems and across Linux kernel versions.

With the advent of the CFS scheduler in Linux 2.6.23, Linux adopted an algorithm that causes relative differences in nice values to have a much stronger effect. In the current implementation, each unit of difference in the nice values of two processes results in a factor of 1.25 in the degree to which the scheduler favors the higher priority process. This causes very low nice values (+19) to truly provide little CPU to a process whenever there is any other higher priority load on the system, and makes high nice values (−20) deliver most of the CPU to applications that require it (e.g., some audio applications).

On Linux, the **RLIMIT_NICE** resource limit can be used to define a limit to which an unprivileged process's nice value can be raised; see *setrlimit*(2) for details.

For further details on the nice value, see the subsections on the autogroup feature and group scheduling, below.

SCHED_BATCH: Scheduling batch processes

(Since Linux 2.6.16.) **SCHED_BATCH** can be used only at static priority 0. This policy is similar to **SCHED_OTHER** in that it schedules the thread according to its dynamic priority (based on the nice value). The difference is that this policy will cause the scheduler to always assume that the thread is CPU-intensive. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behavior, so that this thread is mildly disfavored in scheduling decisions.

This policy is useful for workloads that are noninteractive, but do not want to lower their nice value, and for workloads that want a deterministic scheduling policy without interactivity causing extra preemptions (between the workload's tasks).

SCHED_IDLE: Scheduling very low priority jobs

(Since Linux 2.6.23.) **SCHED_IDLE** can be used only at static priority 0; the process nice value has no influence for this policy.

This policy is intended for running jobs at extremely low priority (lower even than a +19 nice value with the **SCHED_OTHER** or **SCHED_BATCH** policies).

Resetting scheduling policy for child processes

Each thread has a reset-on-fork scheduling flag. When this flag is set, children created by *fork*(2) do not inherit privileged scheduling policies. The reset-on-fork flag can be set by either:

- ORing the **SCHED_RESET_ON_FORK** flag into the *policy* argument when calling *sched_setscheduler*(2) (since Linux 2.6.32); or
- specifying the **SCHED_FLAG_RESET_ON_FORK** flag in *attr.sched_flags* when calling *sched_setattr*(2).

Note that the constants used with these two APIs have different names. The state of the reset-on-fork flag can analogously be retrieved using *sched_getscheduler*(2) and *sched_getattr*(2).

The reset-on-fork feature is intended for media-playback applications, and can be used to prevent applications evading the **RLIMIT_RTTIME** resource limit (see *getrlimit*(2)) by creating multiple child processes.

More precisely, if the reset-on-fork flag is set, the following rules apply for subsequently created children:

- If the calling thread has a scheduling policy of **SCHED_FIFO** or **SCHED_RR**, the policy is reset to **SCHED_OTHER** in child processes.
- If the calling process has a negative nice value, the nice value is reset to zero in child processes.

After the reset-on-fork flag has been enabled, it can be reset only if the thread has the **CAP_SYS_NICE** capability. This flag is disabled in child processes created by *fork*(2).

Privileges and resource limits

Before Linux 2.6.12, only privileged (**CAP_SYS_NICE**) threads can set a nonzero static priority (i.e., set a real-time scheduling policy). The only change that an unprivileged thread can make is to set the **SCHED_OTHER** policy, and this can be done only if the effective user ID of the caller matches the real or effective user ID of the target thread (i.e., the thread specified by *pid*) whose policy is being changed.

A thread must be privileged (**CAP_SYS_NICE**) in order to set or modify a **SCHED_DEADLINE** policy.

Since Linux 2.6.12, the **RLIMIT_RTPRIO** resource limit defines a ceiling on an unprivileged thread's static priority for the **SCHED_RR** and **SCHED_FIFO** policies. The rules for changing scheduling policy and priority are as follows:

- If an unprivileged thread has a nonzero **RLIMIT_RTPRIO** soft limit, then it can change its scheduling policy and priority, subject to the restriction that the priority cannot be set to a value higher than the maximum of its current priority and its **RLIMIT_RTPRIO** soft limit.
- If the **RLIMIT_RTPRIO** soft limit is 0, then the only permitted changes are to lower the priority, or to switch to a non-real-time policy.
- Subject to the same rules, another unprivileged thread can also make these changes, as long as the effective user ID of the thread making the change matches the real or effective user ID of the target thread.
- Special rules apply for the **SCHED_IDLE** policy. Before Linux 2.6.39, an unprivileged thread operating under this policy cannot change its policy, regardless of the value of its **RLIMIT_RTPRIO** resource limit. Since Linux 2.6.39, an unprivileged thread can switch to either the **SCHED_BATCH** or the **SCHED_OTHER** policy so long as its nice value falls within the range permitted by its **RLIMIT_NICE** resource limit (see *getrlimit*(2)).

Privileged (**CAP_SYS_NICE**) threads ignore the **RLIMIT_RTPRIO** limit; as with older kernels, they can make arbitrary changes to scheduling policy and priority. See *getrlimit*(2) for further information on **RLIMIT_RTPRIO**.

Limiting the CPU usage of real-time and deadline processes

A nonblocking infinite loop in a thread scheduled under the **SCHED_FIFO**, **SCHED_RR**, or **SCHED_DEADLINE** policy can potentially block all other threads from accessing the CPU forever. Before Linux 2.6.25, the only way of preventing a runaway real-time process from freezing the system was to run (at the console) a shell scheduled under a higher static priority than the tested application. This allows an emergency kill of tested real-time applications that do not block or terminate as expected.

Since Linux 2.6.25, there are other techniques for dealing with runaway real-time and deadline processes. One of these is to use the **RLIMIT_RTTIME** resource limit to set a ceiling on the CPU

time that a real-time process may consume. See *getrlimit*(2) for details.

Since Linux 2.6.25, Linux also provides two */proc* files that can be used to reserve a certain amount of CPU time to be used by non-real-time processes. Reserving CPU time in this fashion allows some CPU time to be allocated to (say) a root shell that can be used to kill a runaway process. Both of these files specify time values in microseconds:

/proc/sys/kernel/sched_rt_period_us
: This file specifies a scheduling period that is equivalent to 100% CPU bandwidth. The value in this file can range from 1 to **INT_MAX**, giving an operating range of 1 microsecond to around 35 minutes. The default value in this file is 1,000,000 (1 second).

/proc/sys/kernel/sched_rt_runtime_us
: The value in this file specifies how much of the "period" time can be used by all real-time and deadline scheduled processes on the system. The value in this file can range from −1 to **INT_MAX**-1. Specifying −1 makes the run time the same as the period; that is, no CPU time is set aside for non-real-time processes (which was the behavior before Linux 2.6.25). The default value in this file is 950,000 (0.95 seconds), meaning that 5% of the CPU time is reserved for processes that don't run under a real-time or deadline scheduling policy.

Response time

A blocked high priority thread waiting for I/O has a certain response time before it is scheduled again. The device driver writer can greatly reduce this response time by using a "slow interrupt" interrupt handler.

Miscellaneous

Child processes inherit the scheduling policy and parameters across a *fork*(2). The scheduling policy and parameters are preserved across *execve*(2).

Memory locking is usually needed for real-time processes to avoid paging delays; this can be done with *mlock*(2) or *mlockall*(2).

The autogroup feature

Since Linux 2.6.38, the kernel provides a feature known as autogrouping to improve interactive desktop performance in the face of multiprocess, CPU-intensive workloads such as building the Linux kernel with large numbers of parallel build processes (i.e., the *make*(1) **−j** flag).

This feature operates in conjunction with the CFS scheduler and requires a kernel that is configured with **CONFIG_SCHED_AUTOGROUP**. On a running system, this feature is enabled or disabled via the file */proc/sys/kernel/sched_autogroup_enabled*; a value of 0 disables the feature, while a value of 1 enables it. The default value in this file is 1, unless the kernel was booted with the *noautogroup* parameter.

A new autogroup is created when a new session is created via *setsid*(2); this happens, for example, when a new terminal window is started. A new process created by *fork*(2) inherits its parent's autogroup membership. Thus, all of the processes in a session are members of the same autogroup. An autogroup is automatically destroyed when the last process in the group terminates.

When autogrouping is enabled, all of the members of an autogroup are placed in the same kernel scheduler "task group". The CFS scheduler employs an algorithm that equalizes the distribution of CPU cycles across task groups. The benefits of this for interactive desktop performance can be described via the following example.

Suppose that there are two autogroups competing for the same CPU (i.e., presume either a single CPU system or the use of *taskset*(1) to confine all the processes to the same CPU on an SMP system). The first group contains ten CPU-bound processes from a kernel build started with *make −j10*. The other contains a single CPU-bound process: a video player. The effect of autogrouping is that the two groups will each receive half of the CPU cycles. That is, the video player will receive 50% of the CPU cycles, rather than just 9% of the cycles, which would likely lead to degraded video playback. The situation on an SMP system is more complex, but the general effect is the same: the scheduler distributes CPU cycles across task groups such that an autogroup that contains a large number of CPU-bound processes does not end up hogging CPU cycles at the expense of the other jobs on the system.

A process's autogroup (task group) membership can be viewed via the file */proc/*pid*/autogroup*:

```
$ cat /proc/1/autogroup
```

```
/autogroup-1 nice 0
```

This file can also be used to modify the CPU bandwidth allocated to an autogroup. This is done by writing a number in the "nice" range to the file to set the autogroup's nice value. The allowed range is from +19 (low priority) to −20 (high priority). (Writing values outside of this range causes *write*(2) to fail with the error **EINVAL**.)

The autogroup nice setting has the same meaning as the process nice value, but applies to distribution of CPU cycles to the autogroup as a whole, based on the relative nice values of other autogroups. For a process inside an autogroup, the CPU cycles that it receives will be a product of the autogroup's nice value (compared to other autogroups) and the process's nice value (compared to other processes in the same autogroup.

The use of the *cgroups*(7) CPU controller to place processes in cgroups other than the root CPU cgroup overrides the effect of autogrouping.

The autogroup feature groups only processes scheduled under non-real-time policies (**SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE**). It does not group processes scheduled under real-time and deadline policies. Those processes are scheduled according to the rules described earlier.

The nice value and group scheduling

When scheduling non-real-time processes (i.e., those scheduled under the **SCHED_OTHER**, **SCHED_BATCH**, and **SCHED_IDLE** policies), the CFS scheduler employs a technique known as "group scheduling", if the kernel was configured with the **CONFIG_FAIR_GROUP_SCHED** option (which is typical).

Under group scheduling, threads are scheduled in "task groups". Task groups have a hierarchical relationship, rooted under the initial task group on the system, known as the "root task group". Task groups are formed in the following circumstances:

- All of the threads in a CPU cgroup form a task group. The parent of this task group is the task group of the corresponding parent cgroup.
- If autogrouping is enabled, then all of the threads that are (implicitly) placed in an autogroup (i.e., the same session, as created by *setsid*(2)) form a task group. Each new autogroup is thus a separate task group. The root task group is the parent of all such autogroups.
- If autogrouping is enabled, then the root task group consists of all processes in the root CPU cgroup that were not otherwise implicitly placed into a new autogroup.
- If autogrouping is disabled, then the root task group consists of all processes in the root CPU cgroup.
- If group scheduling was disabled (i.e., the kernel was configured without **CONFIG_FAIR_GROUP_SCHED**), then all of the processes on the system are notionally placed in a single task group.

Under group scheduling, a thread's nice value has an effect for scheduling decisions *only relative to other threads in the same task group*. This has some surprising consequences in terms of the traditional semantics of the nice value on UNIX systems. In particular, if autogrouping is enabled (which is the default in various distributions), then employing *setpriority*(2) or *nice*(1) on a process has an effect only for scheduling relative to other processes executed in the same session (typically: the same terminal window).

Conversely, for two processes that are (for example) the sole CPU-bound processes in different sessions (e.g., different terminal windows, each of whose jobs are tied to different autogroups), *modifying the nice value of the process in one of the sessions has no effect* in terms of the scheduler's decisions relative to the process in the other session. A possibly useful workaround here is to use a command such as the following to modify the autogroup nice value for *all* of the processes in a terminal session:

```
$ echo 10 > /proc/self/autogroup
```

Real-time features in the mainline Linux kernel

Since Linux 2.6.18, Linux is gradually becoming equipped with real-time capabilities, most of which are derived from the former *realtime-preempt* patch set. Until the patches have been completely merged into the mainline kernel, they must be installed to achieve the best real-time performance. These patches are named:

patch-*kernelversion*-rt*patchversion*

and can be downloaded from .

Without the patches and prior to their full inclusion into the mainline kernel, the kernel configuration offers only the three preemption classes **CONFIG_PREEMPT_NONE**, **CONFIG_PREEMPT_VOLUNTARY**, and **CONFIG_PREEMPT_DESKTOP** which respectively provide no, some, and considerable reduction of the worst-case scheduling latency.

With the patches applied or after their full inclusion into the mainline kernel, the additional configuration item **CONFIG_PREEMPT_RT** becomes available. If this is selected, Linux is transformed into a regular real-time operating system. The FIFO and RR scheduling policies are then used to run a thread with true real-time priority and a minimum worst-case scheduling latency.

NOTES

The *cgroups*(7) CPU controller can be used to limit the CPU consumption of groups of processes.

Originally, Standard Linux was intended as a general-purpose operating system being able to handle background processes, interactive applications, and less demanding real-time applications (applications that need to usually meet timing deadlines). Although the Linux 2.6 allowed for kernel preemption and the newly introduced O(1) scheduler ensures that the time needed to schedule is fixed and deterministic irrespective of the number of active tasks, true real-time computing was not possible up to Linux 2.6.17.

SEE ALSO

chcpu(1), *chrt*(1), *lscpu*(1), *ps*(1), *taskset*(1), *top*(1), *getpriority*(2), *mlock*(2), *mlockall*(2), *munlock*(2), *munlockall*(2), *nice*(2), *sched_get_priority_max*(2), *sched_get_priority_min*(2), *sched_getaffinity*(2), *sched_getparam*(2), *sched_getscheduler*(2), *sched_rr_get_interval*(2), *sched_setaffinity*(2), *sched_setparam*(2), *sched_setscheduler*(2), *sched_yield*(2), *setpriority*(2), *pthread_getaffinity_np*(3), *pthread_getschedparam*(3), *pthread_setaffinity_np*(3), *sched_getcpu*(3), *capabilities*(7), *cpuset*(7)

Programming for the real world − POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0.

The Linux kernel source files *Documentation/scheduler/sched−deadline.txt*, *Documentation/scheduler/sched−rt−group.txt*, *Documentation/scheduler/sched−design−CFS.txt*, and *Documentation/scheduler/sched−nice−design.txt*

NAME

sem_overview − overview of POSIX semaphores

DESCRIPTION

POSIX semaphores allow processes and threads to synchronize their actions.

A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (**sem_post**(3)); and decrement the semaphore value by one (**sem_wait**(3)). If the value of a semaphore is currently zero, then a *sem_wait*(3) operation will block until the value becomes greater than zero.

POSIX semaphores come in two forms: named semaphores and unnamed semaphores.

Named semaphores

A named semaphore is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX**−*4* (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to *sem_open*(3).

The *sem_open*(3) function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using *sem_post*(3) and *sem_wait*(3). When a process has finished using the semaphore, it can use *sem_close*(3) to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using *sem_unlink*(3).

Unnamed semaphores (memory-based semaphores)

An unnamed semaphore does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (a *thread-shared semaphore*) or processes (a *process-shared semaphore*). A thread-shared semaphore is placed in an area of memory shared between the threads of a process, for example, a global variable. A process-shared semaphore must be placed in a shared memory region (e.g., a System V shared memory segment created using *shmget*(2), or a POSIX shared memory object built created using *shm_open*(3)).

Before being used, an unnamed semaphore must be initialized using *sem_init*(3). It can then be operated on using *sem_post*(3) and *sem_wait*(3). When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using *sem_destroy*(3).

The remainder of this section describes some specific details of the Linux implementation of POSIX semaphores.

Versions

Before Linux 2.6, Linux supported only unnamed, thread-shared semaphores. On a system with Linux 2.6 and a glibc that provides the NPTL threading implementation, a complete implementation of POSIX semaphores is provided.

Persistence

POSIX named semaphores have kernel persistence: if not removed by *sem_unlink*(3), a semaphore will exist until the system is shut down.

Linking

Programs using the POSIX semaphores API must be compiled with *cc −pthread* to link against the real-time library, *librt*.

Accessing named semaphores via the filesystem

On Linux, named semaphores are created in a virtual filesystem, normally mounted under */dev/shm*, with names of the form **sem.***somename*. (This is the reason that semaphore names are limited to **NAME_MAX**−*4* rather than **NAME_MAX** characters.)

Since Linux 2.6.19, ACLs can be placed on files under this directory, to control object permissions on a per-user and per-group basis.

NOTES

System V semaphores (**semget**(2), *semop*(2), etc.) are an older semaphore API. POSIX semaphores provide a simpler, and better designed interface than System V semaphores; on the other hand POSIX semaphores are less widely available (especially on older systems) than System V semaphores.

EXAMPLES

An example of the use of various POSIX semaphore functions is shown in *sem_wait*(3).

SEE ALSO

sem_close(3), *sem_destroy*(3), *sem_getvalue*(3), *sem_init*(3), *sem_open*(3), *sem_post*(3), *sem_unlink*(3), *sem_wait*(3), *pthreads*(7), *shm_overview*(7)

NAME

session-keyring − session shared process keyring

DESCRIPTION

The session keyring is a keyring used to anchor keys on behalf of a process. It is typically created by *pam_keyinit*(8) when a user logs in and a link will be added that refers to the *user-keyring*(7). Optionally, *PAM*(7) may revoke the session keyring on logout. (In typical configurations, PAM does do this revocation.) The session keyring has the name (description) *_ses*.

A special serial number value, **KEY_SPEC_SESSION_KEYRING**, is defined that can be used in lieu of the actual serial number of the calling process's session keyring.

From the *keyctl*(1) utility, '**@s**' can be used instead of a numeric key ID in much the same way.

A process's session keyring is inherited across *clone*(2), *fork*(2), and *vfork*(2). The session keyring is preserved across *execve*(2), even when the executable is set-user-ID or set-group-ID or has capabilities. The session keyring is destroyed when the last process that refers to it exits.

If a process doesn't have a session keyring when it is accessed, then, under certain circumstances, the *user-session-keyring*(7) will be attached as the session keyring and under others a new session keyring will be created. (See *user-session-keyring*(7) for further details.)

Special operations

The *keyutils* library provides the following special operations for manipulating session keyrings:

keyctl_join_session_keyring(3)
: This operation allows the caller to change the session keyring that it subscribes to. The caller can join an existing keyring with a specified name (description), create a new keyring with a given name, or ask the kernel to create a new "anonymous" session keyring with the name "_ses". (This function is an interface to the *keyctl*(2) **KEYCTL_JOIN_SESSION_KEYRING** operation.)

keyctl_session_to_parent(3)
: This operation allows the caller to make the parent process's session keyring to the same as its own. For this to succeed, the parent process must have identical security attributes and must be single threaded. (This function is an interface to the *keyctl*(2) **KEYCTL_SESSION_TO_PARENT** operation.)

These operations are also exposed through the *keyctl*(1) utility as:

```
keyctl session
keyctl session - [<prog> <arg1> <arg2> ...]
keyctl session <name> [<prog> <arg1> <arg2> ...]
```

and:

```
keyctl new_session
```

SEE ALSO

keyctl(1), *keyctl*(3), *keyctl_join_session_keyring*(3), *keyctl_session_to_parent*(3), *keyrings*(7), *PAM*(7), *persistent-keyring*(7), *process-keyring*(7), *thread-keyring*(7), *user-keyring*(7), *user-session-keyring*(7), *pam_keyinit*(8)

NAME

shm_overview − overview of POSIX shared memory

DESCRIPTION

The POSIX shared memory API allows processes to communicate information by sharing a region of memory.

The interfaces employed in the API are:

shm_open(3) Create and open a new object, or open an existing object. This is analogous to *open*(2). The call returns a file descriptor for use by the other interfaces listed below.

ftruncate(2) Set the size of the shared memory object. (A newly created shared memory object has a length of zero.)

mmap(2) Map the shared memory object into the virtual address space of the calling process.

munmap(2) Unmap the shared memory object from the virtual address space of the calling process.

shm_unlink(3) Remove a shared memory object name.

close(2) Close the file descriptor allocated by *shm_open*(3) when it is no longer needed.

fstat(2) Obtain a *stat* structure that describes the shared memory object. Among the information returned by this call are the object's size (*st_size*), permissions (*st_mode*), owner (*st_uid*), and group (*st_gid*).

fchown(2) To change the ownership of a shared memory object.

fchmod(2) To change the permissions of a shared memory object.

Versions

POSIX shared memory is supported since Linux 2.4 and glibc 2.2.

Persistence

POSIX shared memory objects have kernel persistence: a shared memory object will exist until the system is shut down, or until all processes have unmapped the object and it has been deleted with *shm_unlink*(3)

Linking

Programs using the POSIX shared memory API must be compiled with *cc −lrt* to link against the real-time library, *librt*.

Accessing shared memory objects via the filesystem

On Linux, shared memory objects are created in a (**tmpfs**(5)) virtual filesystem, normally mounted under */dev/shm*. Since Linux 2.6.19, Linux supports the use of access control lists (ACLs) to control the permissions of objects in the virtual filesystem.

NOTES

Typically, processes must synchronize their access to a shared memory object, using, for example, POSIX semaphores.

System V shared memory (**shmget**(2), *shmop*(2), etc.) is an older shared memory API. POSIX shared memory provides a simpler, and better designed interface; on the other hand POSIX shared memory is somewhat less widely available (especially on older systems) than System V shared memory.

SEE ALSO

fchmod(2), *fchown*(2), *fstat*(2), *ftruncate*(2), *memfd_create*(2), *mmap*(2), *mprotect*(2), *munmap*(2), *shmget*(2), *shmop*(2), *shm_open*(3), *shm_unlink*(3), *sem_overview*(7)

NAME

signal − overview of signals

DESCRIPTION

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the table below specify the default disposition for each signal, as follows:

Term Default action is to terminate the process.

Ign Default action is to ignore the signal.

Core Default action is to terminate the process and dump core (see *core*(5)).

Stop Default action is to stop the process.

Cont Default action is to continue the process if it is currently stopped.

A process can change the disposition of a signal using *sigaction*(2) or *signal*(2). (The latter is less portable when establishing a signal handler; see *signal*(2) for details.) Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the signal with a *signal handler*, a programmer-defined function that is automatically invoked when the signal is delivered.

By default, a signal handler is invoked on the normal process stack. It is possible to arrange that the signal handler uses an alternate stack; see *sigaltstack*(2) for a discussion of how to do this and when it might be useful.

The signal disposition is a per-process attribute: in a multithreaded application, the disposition of a particular signal is the same for all threads.

A child created via *fork*(2) inherits a copy of its parent's signal dispositions. During an *execve*(2), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

Sending a signal

The following system calls and library functions allow the caller to send a signal:

raise(3)
Sends a signal to the calling thread.

kill(2) Sends a signal to a specified process, to all members of a specified process group, or to all processes on the system.

pidfd_send_signal(2)
Sends a signal to a process identified by a PID file descriptor.

killpg(3)
Sends a signal to all of the members of a specified process group.

pthread_kill(3)
Sends a signal to a specified POSIX thread in the same process as the caller.

tgkill(2)
Sends a signal to a specified thread within a specific process. (This is the system call used to implement *pthread_kill*(3).)

sigqueue(3)
Sends a real-time signal with accompanying data to a specified process.

Waiting for a signal to be caught

The following system calls suspend execution of the calling thread until a signal is caught (or an unhandled signal terminates the process):

pause(2)
: Suspends execution until any signal is caught.

sigsuspend(2)
: Temporarily changes the signal mask (see below) and suspends execution until one of the unmasked signals is caught.

Synchronously accepting a signal

Rather than asynchronously catching a signal via a signal handler, it is possible to synchronously accept the signal, that is, to block execution until the signal is delivered, at which point the kernel returns information about the signal to the caller. There are two general ways to do this:

- *sigwaitinfo*(2), *sigtimedwait*(2), and *sigwait*(3) suspend execution until one of the signals in a specified set is delivered. Each of these calls returns information about the delivered signal.
- *signalfd*(2) returns a file descriptor that can be used to read information about signals that are delivered to the caller. Each *read*(2) from this file descriptor blocks until one of the signals in the set specified in the *signalfd*(2) call is delivered to the caller. The buffer returned by *read*(2) contains a structure describing the signal.

Signal mask and pending signals

A signal may be *blocked*, which means that it will not be delivered until it is later unblocked. Between the time when it is generated and when it is delivered a signal is said to be *pending*.

Each thread in a process has an independent *signal mask*, which indicates the set of signals that the thread is currently blocking. A thread can manipulate its signal mask using *pthread_sigmask*(3). In a traditional single-threaded application, *sigprocmask*(2) can be used to manipulate the signal mask.

A child created via *fork*(2) inherits a copy of its parent's signal mask; the signal mask is preserved across *execve*(2).

A signal may be process-directed or thread-directed. A process-directed signal is one that is targeted at (and thus pending for) the process as a whole. A signal may be process-directed because it was generated by the kernel for reasons other than a hardware exception, or because it was sent using *kill*(2) or *sigqueue*(3). A thread-directed signal is one that is targeted at a specific thread. A signal may be thread-directed because it was generated as a consequence of executing a specific machine-language instruction that triggered a hardware exception (e.g., **SIGSEGV** for an invalid memory access, or **SIGFPE** for a math error), or because it was targeted at a specific thread using interfaces such as *tgkill*(2) or *pthread_kill*(3).

A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal.

A thread can obtain the set of signals that it currently has pending using *sigpending*(2). This set will consist of the union of the set of pending process-directed signals and the set of signals pending for the calling thread.

A child created via *fork*(2) initially has an empty pending signal set; the pending signal set is preserved across an *execve*(2).

Execution of signal handlers

Whenever there is a transition from kernel-mode to user-mode execution (e.g., on return from a system call or scheduling of a thread onto the CPU), the kernel checks whether there is a pending unblocked signal for which the process has established a signal handler. If there is such a pending signal, the following steps occur:

(1) The kernel performs the necessary preparatory steps for execution of the signal handler:

  (1.1) The signal is removed from the set of pending signals.

  (1.2) If the signal handler was installed by a call to *sigaction*(2) that specified the **SA_ONSTACK** flag and the thread has defined an alternate signal stack (using *sigaltstack*(2)), then that stack is installed.

  (1.3) Various pieces of signal-related context are saved into a special frame that is created on the stack. The saved information includes:

- the program counter register (i.e., the address of the next instruction in the main program that should be executed when the signal handler returns);
- architecture-specific register state required for resuming the interrupted program;
- the thread's current signal mask;
- the thread's alternate signal stack settings.

(If the signal handler was installed using the *sigaction*(2) **SA_SIGINFO** flag, then the above information is accessible via the *ucontext_t* object that is pointed to by the third argument of the signal handler.)

(1.4) Any signals specified in *act->sa_mask* when registering the handler with *sigprocmask*(2) are added to the thread's signal mask. The signal being delivered is also added to the signal mask, unless **SA_NODEFER** was specified when registering the handler. These signals are thus blocked while the handler executes.

(2) The kernel constructs a frame for the signal handler on the stack. The kernel sets the program counter for the thread to point to the first instruction of the signal handler function, and configures the return address for that function to point to a piece of user-space code known as the signal trampoline (described in *sigreturn*(2)).

(3) The kernel passes control back to user-space, where execution commences at the start of the signal handler function.

(4) When the signal handler returns, control passes to the signal trampoline code.

(5) The signal trampoline calls *sigreturn*(2), a system call that uses the information in the stack frame created in step 1 to restore the thread to its state before the signal handler was called. The thread's signal mask and alternate signal stack settings are restored as part of this procedure. Upon completion of the call to *sigreturn*(2), the kernel transfers control back to user space, and the thread recommences execution at the point where it was interrupted by the signal handler.

Note that if the signal handler does not return (e.g., control is transferred out of the handler using *siglongjmp*(3), or the handler executes a new program with *execve*(2)), then the final step is not performed. In particular, in such scenarios it is the programmer's responsibility to restore the state of the signal mask (using *sigprocmask*(2)), if it is desired to unblock the signals that were blocked on entry to the signal handler. (Note that *siglongjmp*(3) may or may not restore the signal mask, depending on the *savesigs* value that was specified in the corresponding call to *sigsetjmp*(3).)

From the kernel's point of view, execution of the signal handler code is exactly the same as the execution of any other user-space code. That is to say, the kernel does not record any special state information indicating that the thread is currently executing inside a signal handler. All necessary state information is maintained in user-space registers and the user-space stack. The depth to which nested signal handlers may be invoked is thus limited only by the user-space stack (and sensible software design!).

Standard signals

Linux supports the standard signals listed below. The second column of the table indicates which standard (if any) specified the signal: "P1990" indicates that the signal is described in the original POSIX.1-1990 standard; "P2001" indicates that the signal was added in SUSv2 and POSIX.1-2001.

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from *abort*(3)
SIGALRM	P1990	Term	Timer signal from *alarm*(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	−	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	−	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	−		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard

SIGIO	−	Term	I/O now possible (4.2BSD)
SIGIOT	−	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	−	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see *pipe*(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	−	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	−	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also *seccomp*(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	−	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see *setrlimit*(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see *setrlimit*(2)
SIGWINCH	−	Ign	Window resize signal (4.3BSD, Sun)

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Up to and including Linux 2.2, the default behavior for **SIGSYS**, **SIGXCPU**, **SIGXFSZ**, and (on architectures other than SPARC and MIPS) **SIGBUS** was to terminate the process (without a core dump). (On some other UNIX systems the default action for **SIGXCPU** and **SIGXFSZ** is to terminate the process without a core dump.) Linux 2.4 conforms to the POSIX.1-2001 requirements for these signals, terminating the process with a core dump.

SIGEMT is not specified in POSIX.1-2001, but nevertheless appears on most other UNIX systems, where its default action is typically to terminate the process with a core dump.

SIGPWR (which is not specified in POSIX.1-2001) is typically ignored by default on those other UNIX systems where it appears.

SIGIO (which is not specified in POSIX.1-2001) is ignored by default on several other UNIX systems.

Queueing and delivery semantics for standard signals

If multiple standard signals are pending for a process, the order in which the signals are delivered is unspecified.

Standard signals do not queue. If multiple instances of a standard signal are generated while that signal is blocked, then only one instance of the signal is marked as pending (and the signal will be delivered just once when it is unblocked). In the case where a standard signal is already pending, the *siginfo_t* structure (see *sigaction*(2)) associated with that signal is not overwritten on arrival of subsequent instances of the same signal. Thus, the process will receive the information associated with the first instance of the signal.

Signal numbering for standard signals

The numeric value for each signal is given in the table below. As shown in the table, many signals have different numeric values on different architectures. The first numeric value in each table row shows the signal number on x86, ARM, and most other architectures; the second value is for Alpha and SPARC;

the third is for MIPS; and the last is for PARISC. A dash (−) denotes that a signal is absent on the corresponding architecture.

Signal	x86/ARM most others	Alpha/ SPARC	MIPS	PARISC	Notes
SIGHUP	1	1	1	1	
SIGINT	2	2	2	2	
SIGQUIT	3	3	3	3	
SIGILL	4	4	4	4	
SIGTRAP	5	5	5	5	
SIGABRT	6	6	6	6	
SIGIOT	6	6	6	6	
SIGBUS	7	10	10	10	
SIGEMT	−	7	7	-	
SIGFPE	8	8	8	8	
SIGKILL	9	9	9	9	
SIGUSR1	10	30	16	16	
SIGSEGV	11	11	11	11	
SIGUSR2	12	31	17	17	
SIGPIPE	13	13	13	13	
SIGALRM	14	14	14	14	
SIGTERM	15	15	15	15	
SIGSTKFLT	16	−	−	7	
SIGCHLD	17	20	18	18	
SIGCLD	−	−	18	−	
SIGCONT	18	19	25	26	
SIGSTOP	19	17	23	24	
SIGTSTP	20	18	24	25	
SIGTTIN	21	21	26	27	
SIGTTOU	22	22	27	28	
SIGURG	23	16	21	29	
SIGXCPU	24	24	30	12	
SIGXFSZ	25	25	31	30	
SIGVTALRM	26	26	28	20	
SIGPROF	27	27	29	21	
SIGWINCH	28	28	20	23	
SIGIO	29	23	22	22	
SIGPOLL					Same as SIGIO
SIGPWR	30	29/−	19	19	
SIGINFO	−	29/−	−	−	
SIGLOST	−	−/29	−	−	
SIGSYS	31	12	12	31	
SIGUNUSED	31	−	−	31	

Note the following:

- Where defined, **SIGUNUSED** is synonymous with **SIGSYS**. Since glibc 2.26, **SIGUNUSED** is no longer defined on any architecture.
- Signal 29 is **SIGINFO**/**SIGPWR** (synonyms for the same value) on Alpha but **SIGLOST** on SPARC.

Real-time signals

Starting with Linux 2.2, Linux supports real-time signals as originally defined in the POSIX.1b real-time extensions (and now included in POSIX.1-2001). The range of supported real-time signals is defined by the macros **SIGRTMIN** and **SIGRTMAX**. POSIX.1-2001 requires that an implementation support at least **_POSIX_RTSIG_MAX** (8) real-time signals.

The Linux kernel supports a range of 33 different real-time signals, numbered 32 to 64. However, the glibc POSIX threads implementation internally uses two (for NPTL) or three (for LinuxThreads) real-time signals (see *pthreads*(7)), and adjusts the value of **SIGRTMIN** suitably (to 34 or 35). Because the range of available real-time signals varies according to the glibc threading implementation (and this

variation can occur at run time according to the available kernel and glibc), and indeed the range of real-time signals varies across UNIX systems, programs should *never refer to real-time signals using hard-coded numbers*, but instead should always refer to real-time signals using the notation **SIGRTMIN**+n, and include suitable (run-time) checks that **SIGRTMIN**+n does not exceed **SIGRTMAX**.

Unlike standard signals, real-time signals have no predefined meanings: the entire set of real-time signals can be used for application-defined purposes.

The default action for an unhandled real-time signal is to terminate the receiving process.

Real-time signals are distinguished by the following:

- Multiple instances of real-time signals can be queued. By contrast, if multiple instances of a standard signal are delivered while that signal is currently blocked, then only one instance is queued.
- If the signal is sent using *sigqueue*(3), an accompanying value (either an integer or a pointer) can be sent with the signal. If the receiving process establishes a handler for this signal using the **SA_SIGINFO** flag to *sigaction*(2), then it can obtain this data via the *si_value* field of the *siginfo_t* structure passed as the second argument to the handler. Furthermore, the *si_pid* and *si_uid* fields of this structure can be used to obtain the PID and real user ID of the process sending the signal.
- Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal. (I.e., low-numbered signals have highest priority.) By contrast, if multiple standard signals are pending for a process, the order in which they are delivered is unspecified.

If both standard and real-time signals are pending for a process, POSIX leaves it unspecified which is delivered first. Linux, like many other implementations, gives priority to standard signals in this case.

According to POSIX, an implementation should permit at least **_POSIX_SIGQUEUE_MAX** (32) real-time signals to be queued to a process. However, Linux does things differently. Up to and including Linux 2.6.7, Linux imposes a system-wide limit on the number of queued real-time signals for all processes. This limit can be viewed and (with privilege) changed via the */proc/sys/kernel/rtsig-max* file. A related file, */proc/sys/kernel/rtsig-nr*, can be used to find out how many real-time signals are currently queued. In Linux 2.6.8, these */proc* interfaces were replaced by the **RLIMIT_SIGPENDING** resource limit, which specifies a per-user limit for queued signals; see *setrlimit*(2) for further details.

The addition of real-time signals required the widening of the signal set structure (*sigset_t*) from 32 to 64 bits. Consequently, various system calls were superseded by new system calls that supported the larger signal sets. The old and new system calls are as follows:

Linux 2.0 and earlier	**Linux 2.2 and later**
sigaction(2)	*rt_sigaction*(2)
sigpending(2)	*rt_sigpending*(2)
sigprocmask(2)	*rt_sigprocmask*(2)
sigreturn(2)	*rt_sigreturn*(2)
sigsuspend(2)	*rt_sigsuspend*(2)
sigtimedwait(2)	*rt_sigtimedwait*(2)

Interruption of system calls and library functions by signal handlers

If a signal handler is invoked while a system call or library function call is blocked, then either:

- the call is automatically restarted after the signal handler returns; or
- the call fails with the error **EINTR**.

Which of these two behaviors occurs depends on the interface and whether or not the signal handler was established using the **SA_RESTART** flag (see *sigaction*(2)). The details vary across UNIX systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted by a signal handler, then the call is automatically restarted after the signal handler returns if the **SA_RESTART** flag was used; otherwise the call fails with the error **EINTR**:

- *read*(2), *readv*(2), *write*(2), *writev*(2), and *ioctl*(2) calls on "slow" devices. A "slow" device is one where the I/O call may block for an indefinite time, for example, a terminal, pipe, or socket. If an I/O call on a slow device has already transferred some data by the time it is interrupted by a signal

handler, then the call will return a success status (normally, the number of bytes transferred). Note that a (local) disk is not a slow device according to this definition; I/O operations on disk devices are not interrupted by signals.

- *open*(2), if it can block (e.g., when opening a FIFO; see *fifo*(7)).
- *wait*(2), *wait3*(2), *wait4*(2), *waitid*(2), and *waitpid*(2).
- Socket interfaces: *accept*(2), *connect*(2), *recv*(2), *recvfrom*(2), *recvmmsg*(2), *recvmsg*(2), *send*(2), *sendto*(2), and *sendmsg*(2), unless a timeout has been set on the socket (see below).
- File locking interfaces: *flock*(2) and the **F_SETLKW** and **F_OFD_SETLKW** operations of *fcntl*(2)
- POSIX message queue interfaces: *mq_receive*(3), *mq_timedreceive*(3), *mq_send*(3), and *mq_timedsend*(3).
- *futex*(2) **FUTEX_WAIT** (since Linux 2.6.22; beforehand, always failed with **EINTR**).
- *getrandom*(2).
- *pthread_mutex_lock*(3), *pthread_cond_wait*(3), and related APIs.
- *futex*(2) **FUTEX_WAIT_BITSET**.
- POSIX semaphore interfaces: *sem_wait*(3) and *sem_timedwait*(3) (since Linux 2.6.22; beforehand, always failed with **EINTR**).
- *read*(2) from an *inotify*(7) file descriptor (since Linux 3.8; beforehand, always failed with **EINTR**).

The following interfaces are never restarted after being interrupted by a signal handler, regardless of the use of **SA_RESTART**; they always fail with the error **EINTR** when interrupted by a signal handler:

- "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using *setsockopt*(2): *accept*(2), *recv*(2), *recvfrom*(2), *recvmmsg*(2) (also with a non-NULL *timeout* argument), and *recvmsg*(2).
- "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using *setsockopt*(2): *connect*(2), *send*(2), *sendto*(2), and *sendmsg*(2).
- Interfaces used to wait for signals: *pause*(2), *sigsuspend*(2), *sigtimedwait*(2), and *sigwaitinfo*(2).
- File descriptor multiplexing interfaces: *epoll_wait*(2), *epoll_pwait*(2), *poll*(2), *ppoll*(2), *select*(2), and *pselect*(2).
- System V IPC interfaces: *msgrcv*(2), *msgsnd*(2), *semop*(2), and *semtimedop*(2).
- Sleep interfaces: *clock_nanosleep*(2), *nanosleep*(2), and *usleep*(3).
- *io_getevents*(2).

The *sleep*(3) function is also never restarted if interrupted by a handler, but gives a success return: the number of seconds remaining to sleep.

In certain circumstances, the *seccomp*(2) user-space notification feature can lead to restarting of system calls that would otherwise never be restarted by **SA_RESTART**; for details, see *seccomp_unotify*(2).

Interruption of system calls and library functions by stop signals

On Linux, even in the absence of signal handlers, certain blocking interfaces can fail with the error **EINTR** after the process is stopped by one of the stop signals and then resumed via **SIGCONT**. This behavior is not sanctioned by POSIX.1, and doesn't occur on other systems.

The Linux interfaces that display this behavior are:

- "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using *setsockopt*(2): *accept*(2), *recv*(2), *recvfrom*(2), *recvmmsg*(2) (also with a non-NULL *timeout* argument), and *recvmsg*(2).
- "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using *setsockopt*(2): *connect*(2), *send*(2), *sendto*(2), and *sendmsg*(2), if a send timeout (**SO_SNDTIMEO**) has been set.

- *epoll_wait*(2), *epoll_pwait*(2).
- *semop*(2), *semtimedop*(2).
- *sigtimedwait*(2), *sigwaitinfo*(2).
- Linux 3.7 and earlier: *read*(2) from an *inotify*(7) file descriptor
- Linux 2.6.21 and earlier: *futex*(2) **FUTEX_WAIT**, *sem_timedwait*(3), *sem_wait*(3).
- Linux 2.6.8 and earlier: *msgrcv*(2), *msgsnd*(2).
- Linux 2.4 and earlier: *nanosleep*(2).

STANDARDS

POSIX.1, except as noted.

NOTES

For a discussion of async-signal-safe functions, see *signal-safety*(7).

The */proc/*pid*/task/*tid*/status* file contains various fields that show the signals that a thread is blocking (*SigBlk*), catching (*SigCgt*), or ignoring (*SigIgn*). (The set of signals that are caught or ignored will be the same across all threads in a process.) Other fields show the set of pending signals that are directed to the thread (*SigPnd*) as well as the set of pending signals that are directed to the process as a whole (*ShdPnd*). The corresponding fields in */proc/*pid*/status* show the information for the main thread. See *proc*(5) for further details.

BUGS

There are six signals that can be delivered as a consequence of a hardware exception: **SIGBUS**, **SIGEMT**, **SIGFPE**, **SIGILL**, **SIGSEGV**, and **SIGTRAP**. Which of these signals is delivered, for any given hardware exception, is not documented and does not always make sense.

For example, an invalid memory access that causes delivery of **SIGSEGV** on one CPU architecture may cause delivery of **SIGBUS** on another architecture, or vice versa.

For another example, using the x86 *int* instruction with a forbidden argument (any number other than 3 or 128) causes delivery of **SIGSEGV**, even though **SIGILL** would make more sense, because of how the CPU reports the forbidden operation to the kernel.

SEE ALSO

kill(1), *clone*(2), *getrlimit*(2), *kill*(2), *pidfd_send_signal*(2), *restart_syscall*(2), *rt_sigqueueinfo*(2), *setitimer*(2), *setrlimit*(2), *sgetmask*(2), *sigaction*(2), *sigaltstack*(2), *signal*(2), *signalfd*(2), *sigpending*(2), *sigprocmask*(2), *sigreturn*(2), *sigsuspend*(2), *sigwaitinfo*(2), *abort*(3), *bsd_signal*(3), *killpg*(3), *longjmp*(3), *pthread_sigqueue*(3), *raise*(3), *sigqueue*(3), *sigset*(3), *sigsetops*(3), *sigvec*(3), *sigwait*(3), *strsignal*(3), *swapcontext*(3), *sysv_signal*(3), *core*(5), *proc*(5), *nptl*(7), *pthreads*(7), *sigevent*(3type)

NAME

signal-safety − async-signal-safe functions

DESCRIPTION

An *async-signal-safe* function is one that can be safely called from within a signal handler. Many functions are *not* async-signal-safe. In particular, nonreentrant functions are generally unsafe to call from a signal handler.

The kinds of issues that render a function unsafe can be quickly understood when one considers the implementation of the *stdio* library, all of whose functions are not async-signal-safe.

When performing buffered I/O on a file, the *stdio* functions must maintain a statically allocated data buffer along with associated counters and indexes (or pointers) that record the amount of data and the current position in the buffer. Suppose that the main program is in the middle of a call to a *stdio* function such as *printf*(3) where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls *printf*(3), then the second call to *printf*(3) will operate on inconsistent data, with unpredictable results.

To avoid problems with unsafe functions, there are two possible choices:

(a) Ensure that (1) the signal handler calls only async-signal-safe functions, and (2) the signal handler itself is reentrant with respect to global variables in the main program.

(b) Block signal delivery in the main program when calling functions that are unsafe or operating on global data that is also accessed by the signal handler.

Generally, the second choice is difficult in programs of any complexity, so the first choice is taken.

POSIX.1 specifies a set of functions that an implementation must make async-signal-safe. (An implementation may provide safe implementations of additional functions, but this is not required by the standard and other implementations may not provide the same guarantees.)

In general, a function is async-signal-safe either because it is reentrant or because it is atomic with respect to signals (i.e., its execution can't be interrupted by a signal handler).

The set of functions required to be async-signal-safe by POSIX.1 is shown in the following table. The functions not otherwise noted were required to be async-signal-safe in POSIX.1-2001; the table details changes in the subsequent standards.

Function	**Notes**
abort(3)	Added in POSIX.1-2001 TC1
accept(2)	
access(2)	
aio_error(3)	
aio_return(3)	
aio_suspend(3)	See notes below
alarm(2)	
bind(2)	
cfgetispeed(3)	
cfgetospeed(3)	
cfsetispeed(3)	
cfsetospeed(3)	
chdir(2)	
chmod(2)	
chown(2)	
clock_gettime(2)	
close(2)	
connect(2)	
creat(2)	
dup(2)	
dup2(2)	
execl(3)	Added in POSIX.1-2008; see notes below
execle(3)	See notes below

execv(3)	Added in POSIX.1-2008
execve(2)	
_exit(2)	
_Exit(2)	
faccessat(2)	Added in POSIX.1-2008
fchdir(2)	Added in POSIX.1-2008 TC1
fchmod(2)	
fchmodat(2)	Added in POSIX.1-2008
fchown(2)	
fchownat(2)	Added in POSIX.1-2008
fcntl(2)	
fdatasync(2)	
fexecve(3)	Added in POSIX.1-2008
ffs(3)	Added in POSIX.1-2008 TC2
fork(2)	See notes below
fstat(2)	
fstatat(2)	Added in POSIX.1-2008
fsync(2)	
ftruncate(2)	
futimens(3)	Added in POSIX.1-2008
getegid(2)	
geteuid(2)	
getgid(2)	
getgroups(2)	
getpeername(2)	
getpgrp(2)	
getpid(2)	
getppid(2)	
getsockname(2)	
getsockopt(2)	
getuid(2)	
htonl(3)	Added in POSIX.1-2008 TC2
htons(3)	Added in POSIX.1-2008 TC2
kill(2)	
link(2)	
linkat(2)	Added in POSIX.1-2008
listen(2)	
longjmp(3)	Added in POSIX.1-2008 TC2; see notes below
lseek(2)	
lstat(2)	
memccpy(3)	Added in POSIX.1-2008 TC2
memchr(3)	Added in POSIX.1-2008 TC2
memcmp(3)	Added in POSIX.1-2008 TC2
memcpy(3)	Added in POSIX.1-2008 TC2
memmove(3)	Added in POSIX.1-2008 TC2
memset(3)	Added in POSIX.1-2008 TC2
mkdir(2)	
mkdirat(2)	Added in POSIX.1-2008
mkfifo(3)	
mkfifoat(3)	Added in POSIX.1-2008
mknod(2)	Added in POSIX.1-2008
mknodat(2)	Added in POSIX.1-2008
ntohl(3)	Added in POSIX.1-2008 TC2
ntohs(3)	Added in POSIX.1-2008 TC2
open(2)	
openat(2)	Added in POSIX.1-2008
pause(2)	

pipe(2)	
poll(2)	
posix_trace_event(3)	
pselect(2)	
pthread_kill(3)	Added in POSIX.1-2008 TC1
pthread_self(3)	Added in POSIX.1-2008 TC1
pthread_sigmask(3)	Added in POSIX.1-2008 TC1
raise(3)	
read(2)	
readlink(2)	
readlinkat(2)	Added in POSIX.1-2008
recv(2)	
recvfrom(2)	
recvmsg(2)	
rename(2)	
renameat(2)	Added in POSIX.1-2008
rmdir(2)	
select(2)	
sem_post(3)	
send(2)	
sendmsg(2)	
sendto(2)	
setgid(2)	
setpgid(2)	
setsid(2)	
setsockopt(2)	
setuid(2)	
shutdown(2)	
sigaction(2)	
sigaddset(3)	
sigdelset(3)	
sigemptyset(3)	
sigfillset(3)	
sigismember(3)	
siglongjmp(3)	Added in POSIX.1-2008 TC2; see notes below
signal(2)	
sigpause(3)	
sigpending(2)	
sigprocmask(2)	
sigqueue(2)	
sigset(3)	
sigsuspend(2)	
sleep(3)	
sockatmark(3)	Added in POSIX.1-2001 TC2
socket(2)	
socketpair(2)	
stat(2)	
stpcpy(3)	Added in POSIX.1-2008 TC2
stpncpy(3)	Added in POSIX.1-2008 TC2
strcat(3)	Added in POSIX.1-2008 TC2
strchr(3)	Added in POSIX.1-2008 TC2
strcmp(3)	Added in POSIX.1-2008 TC2
strcpy(3)	Added in POSIX.1-2008 TC2
strcspn(3)	Added in POSIX.1-2008 TC2
strlen(3)	Added in POSIX.1-2008 TC2
strncat(3)	Added in POSIX.1-2008 TC2
strncmp(3)	Added in POSIX.1-2008 TC2

strncpy(3)	Added in POSIX.1-2008 TC2
strnlen(3)	Added in POSIX.1-2008 TC2
strpbrk(3)	Added in POSIX.1-2008 TC2
strrchr(3)	Added in POSIX.1-2008 TC2
strspn(3)	Added in POSIX.1-2008 TC2
strstr(3)	Added in POSIX.1-2008 TC2
strtok_r(3)	Added in POSIX.1-2008 TC2
symlink(2)	
symlinkat(2)	Added in POSIX.1-2008
tcdrain(3)	
tcflow(3)	
tcflush(3)	
tcgetattr(3)	
tcgetpgrp(3)	
tcsendbreak(3)	
tcsetattr(3)	
tcsetpgrp(3)	
time(2)	
timer_getoverrun(2)	
timer_gettime(2)	
timer_settime(2)	
times(2)	
umask(2)	
uname(2)	
unlink(2)	
unlinkat(2)	Added in POSIX.1-2008
utime(2)	
utimensat(2)	Added in POSIX.1-2008
utimes(2)	Added in POSIX.1-2008
wait(2)	
waitpid(2)	
wcpcpy(3)	Added in POSIX.1-2008 TC2
wcpncpy(3)	Added in POSIX.1-2008 TC2
wcscat(3)	Added in POSIX.1-2008 TC2
wcschr(3)	Added in POSIX.1-2008 TC2
wcscmp(3)	Added in POSIX.1-2008 TC2
wcscpy(3)	Added in POSIX.1-2008 TC2
wcscspn(3)	Added in POSIX.1-2008 TC2
wcslen(3)	Added in POSIX.1-2008 TC2
wcsncat(3)	Added in POSIX.1-2008 TC2
wcsncmp(3)	Added in POSIX.1-2008 TC2
wcsncpy(3)	Added in POSIX.1-2008 TC2
wcsnlen(3)	Added in POSIX.1-2008 TC2
wcspbrk(3)	Added in POSIX.1-2008 TC2
wcsrchr(3)	Added in POSIX.1-2008 TC2
wcsspn(3)	Added in POSIX.1-2008 TC2
wcsstr(3)	Added in POSIX.1-2008 TC2
wcstok(3)	Added in POSIX.1-2008 TC2
wmemchr(3)	Added in POSIX.1-2008 TC2
wmemcmp(3)	Added in POSIX.1-2008 TC2
wmemcpy(3)	Added in POSIX.1-2008 TC2
wmemmove(3)	Added in POSIX.1-2008 TC2
wmemset(3)	Added in POSIX.1-2008 TC2
write(2)	

Notes:

- POSIX.1-2001 and POSIX.1-2001 TC2 required the functions *fpathconf*(3), *pathconf*(3), and *sysconf*(3) to be async-signal-safe, but this requirement was removed in POSIX.1-2008.

- If a signal handler interrupts the execution of an unsafe function, and the handler terminates via a call to *longjmp*(3) or *siglongjmp*(3) and the program subsequently calls an unsafe function, then the behavior of the program is undefined.
- POSIX.1-2001 TC1 clarified that if an application calls *fork*(2) from a signal handler and any of the fork handlers registered by *pthread_atfork*(3) calls a function that is not async-signal-safe, the behavior is undefined. A future revision of the standard is likely to remove *fork*(2) from the list of async-signal-safe functions.
- Asynchronous signal handlers that call functions which are cancelation points and nest over regions of deferred cancelation may trigger cancelation whose behavior is as if asynchronous cancelation had occurred and may cause application state to become inconsistent.

errno

Fetching and setting the value of *errno* is async-signal-safe provided that the signal handler saves *errno* on entry and restores its value before returning.

Deviations in the GNU C library

The following known deviations from the standard occur in the GNU C library:

- Before glibc 2.24, *execl*(3) and *execle*(3) employed *realloc*(3) internally and were consequently not async-signal-safe. This was fixed in glibc 2.24.
- The glibc implementation of *aio_suspend*(3) is not async-signal-safe because it uses *pthread_mutex_lock*(3) internally.

SEE ALSO

sigaction(2), *signal*(7), *standards*(7)

NAME

sock_diag − obtaining information about sockets

SYNOPSIS

#include <sys/socket.h>
#include <linux/sock_diag.h>
#include <linux/unix_diag.h> /* for UNIX domain sockets */
#include <linux/inet_diag.h> /* for IPv4 and IPv6 sockets */

diag_socket = socket(AF_NETLINK, *socket_type***, NETLINK_SOCK_DIAG);**

DESCRIPTION

The sock_diag netlink subsystem provides a mechanism for obtaining information about sockets of various address families from the kernel. This subsystem can be used to obtain information about individual sockets or request a list of sockets.

In the request, the caller can specify additional information it would like to obtain about the socket, for example, memory information or information specific to the address family.

When requesting a list of sockets, the caller can specify filters that would be applied by the kernel to select a subset of sockets to report. For now, there is only the ability to filter sockets by state (connected, listening, and so on.)

Note that sock_diag reports only those sockets that have a name; that is, either sockets bound explicitly with *bind*(2) or sockets that were automatically bound to an address (e.g., by *connect*(2)). This is the same set of sockets that is available via */proc/net/unix*, */proc/net/tcp*, */proc/net/udp*, and so on.

Request

The request starts with a *struct nlmsghdr* header described in *netlink*(7) with *nlmsg_type* field set to **SOCK_DIAG_BY_FAMILY**. It is followed by a header specific to the address family that starts with a common part shared by all address families:

```
struct sock_diag_req {
    __u8 sdiag_family;
    __u8 sdiag_protocol;
};
```

The fields of this structure are as follows:

sdiag_family
: An address family. It should be set to the appropriate **AF_*** constant.

sdiag_protocol
: Depends on *sdiag_family*. It should be set to the appropriate **IPPROTO_*** constant for **AF_INET** and **AF_INET6**, and to 0 otherwise.

If the *nlmsg_flags* field of the *struct nlmsghdr* header has the **NLM_F_DUMP** flag set, it means that a list of sockets is being requested; otherwise it is a query about an individual socket.

Response

The response starts with a *struct nlmsghdr* header and is followed by an array of objects specific to the address family. The array is to be accessed with the standard **NLMSG_*** macros from the *netlink*(3) API.

Each object is the NLA (netlink attributes) list that is to be accessed with the **RTA_*** macros from *rtnetlink*(3) API.

UNIX domain sockets

For UNIX domain sockets the request is represented in the following structure:

```
struct unix_diag_req {
    __u8    sdiag_family;
    __u8    sdiag_protocol;
    __u16   pad;
    __u32   udiag_states;
    __u32   udiag_ino;
    __u32   udiag_show;
    __u32   udiag_cookie[2];
```

```
};
```

The fields of this structure are as follows:

sdiag_family
: The address family; it should be set to **AF_UNIX**.

sdiag_protocol
pad
: These fields should be set to 0.

udiag_states
: This is a bit mask that defines a filter of sockets states. Only those sockets whose states are in this mask will be reported. Ignored when querying for an individual socket. Supported values are:

1 << **TCP_ESTABLISHED**

1 << **TCP_LISTEN**

udiag_ino
: This is an inode number when querying for an individual socket. Ignored when querying for a list of sockets.

udiag_show
: This is a set of flags defining what kind of information to report. Each requested kind of information is reported back as a netlink attribute as described below:

UDIAG_SHOW_NAME
: The attribute reported in answer to this request is **UNIX_DIAG_NAME**. The payload associated with this attribute is the pathname to which the socket was bound (a sequence of bytes up to **UNIX_PATH_MAX** length).

UDIAG_SHOW_VFS
: The attribute reported in answer to this request is **UNIX_DIAG_VFS**. The payload associated with this attribute is represented in the following structure:

```
struct unix_diag_vfs {
    __u32 udiag_vfs_dev;
    __u32 udiag_vfs_ino;
};
```

The fields of this structure are as follows:

udiag_vfs_dev
: The device number of the corresponding on-disk socket inode.

udiag_vfs_ino
: The inode number of the corresponding on-disk socket inode.

UDIAG_SHOW_PEER
: The attribute reported in answer to this request is **UNIX_DIAG_PEER**. The payload associated with this attribute is a __u32 value which is the peer's inode number. This attribute is reported for connected sockets only.

UDIAG_SHOW_ICONS
: The attribute reported in answer to this request is **UNIX_DIAG_ICONS**. The payload associated with this attribute is an array of __u32 values which are inode numbers of sockets that has passed the *connect*(2) call, but hasn't been processed with *accept*(2) yet. This attribute is reported for listening sockets only.

UDIAG_SHOW_RQLEN
: The attribute reported in answer to this request is **UNIX_DIAG_RQLEN**. The payload associated with this attribute is represented in the following structure:

```
struct unix_diag_rqlen {
    __u32 udiag_rqueue;
    __u32 udiag_wqueue;
};
```

The fields of this structure are as follows:

udiag_rqueue
: For listening sockets: the number of pending connections. The length of the array associated with the **UNIX_DIAG_ICONS** response attribute is equal to this value.

  For established sockets: the amount of data in incoming queue.

udiag_wqueue
: For listening sockets: the backlog length which equals to the value passed as the second argument to *listen*(2).

  For established sockets: the amount of memory available for sending.

UDIAG_SHOW_MEMINFO
: The attribute reported in answer to this request is **UNIX_DIAG_MEMINFO**. The payload associated with this attribute is an array of __u32 values described below in the subsection "Socket memory information".

The following attributes are reported back without any specific request:

UNIX_DIAG_SHUTDOWN
: The payload associated with this attribute is __u8 value which represents bits of *shutdown*(2) state.

udiag_cookie
: This is an array of opaque identifiers that could be used along with *udiag_ino* to specify an individual socket. It is ignored when querying for a list of sockets, as well as when all its elements are set to −1.

The response to a query for UNIX domain sockets is represented as an array of

```
struct unix_diag_msg {
    __u8    udiag_family;
    __u8    udiag_type;
    __u8    udiag_state;
    __u8    pad;
    __u32   udiag_ino;
    __u32   udiag_cookie[2];
};
```

followed by netlink attributes.

The fields of this structure are as follows:

udiag_family
: This field has the same meaning as in *struct unix_diag_req*.

udiag_type
: This is set to one of **SOCK_PACKET**, **SOCK_STREAM**, or **SOCK_SEQPACKET**.

udiag_state
: This is set to one of **TCP_LISTEN** or **TCP_ESTABLISHED**.

pad This field is set to 0.

udiag_ino
: This is the socket inode number.

udiag_cookie
: This is an array of opaque identifiers that could be used in subsequent queries.

IPv4 and IPv6 sockets

For IPv4 and IPv6 sockets, the request is represented in the following structure:

```
struct inet_diag_req_v2 {
    __u8    sdiag_family;
    __u8    sdiag_protocol;
    __u8    idiag_ext;
    __u8    pad;
```

```
        __u32   idiag_states;
        struct inet_diag_sockid id;
    };
```

where *struct inet_diag_sockid* is defined as follows:

```
    struct inet_diag_sockid {
        __be16  idiag_sport;
        __be16  idiag_dport;
        __be32  idiag_src[4];
        __be32  idiag_dst[4];
        __u32   idiag_if;
        __u32   idiag_cookie[2];
    };
```

The fields of *struct inet_diag_req_v2* are as follows:

sdiag_family
: This should be set to either **AF_INET** or **AF_INET6** for IPv4 or IPv6 sockets respectively.

sdiag_protocol
: This should be set to one of **IPPROTO_TCP**, **IPPROTO_UDP**, or **IPPROTO_UDPLITE**.

idiag_ext
: This is a set of flags defining what kind of extended information to report. Each requested kind of information is reported back as a netlink attribute as described below:

  INET_DIAG_TOS
  : The payload associated with this attribute is a __u8 value which is the TOS of the socket.

  INET_DIAG_TCLASS
  : The payload associated with this attribute is a __u8 value which is the TClass of the socket. IPv6 sockets only. For LISTEN and CLOSE sockets, this is followed by **INET_DIAG_SKV6ONLY** attribute with associated __u8 payload value meaning whether the socket is IPv6-only or not.

  INET_DIAG_MEMINFO
  : The payload associated with this attribute is represented in the following structure:

    ```
    struct inet_diag_meminfo {
        __u32 idiag_rmem;
        __u32 idiag_wmem;
        __u32 idiag_fmem;
        __u32 idiag_tmem;
    };
    ```

    The fields of this structure are as follows:

    idiag_rmem The amount of data in the receive queue.

    idiag_wmem The amount of data that is queued by TCP but not yet sent.

    idiag_fmem The amount of memory scheduled for future use (TCP only).

    idiag_tmem The amount of data in send queue.

  INET_DIAG_SKMEMINFO
  : The payload associated with this attribute is an array of __u32 values described below in the subsection "Socket memory information".

  INET_DIAG_INFO
  : The payload associated with this attribute is specific to the address family. For TCP sockets, it is an object of type *struct tcp_info*.

  INET_DIAG_CONG
  : The payload associated with this attribute is a string that describes the congestion control algorithm used. For TCP sockets only.

pad This should be set to 0.

idiag_states
: This is a bit mask that defines a filter of socket states. Only those sockets whose states are in this mask will be reported. Ignored when querying for an individual socket.

id This is a socket ID object that is used in dump requests, in queries about individual sockets, and is reported back in each response. Unlike UNIX domain sockets, IPv4 and IPv6 sockets are identified using addresses and ports. All values are in network byte order.

The fields of *struct inet_diag_sockid* are as follows:

idiag_sport
: The source port.

idiag_dport
: The destination port.

idiag_src
: The source address.

idiag_dst
: The destination address.

idiag_if
: The interface number the socket is bound to.

idiag_cookie
: This is an array of opaque identifiers that could be used along with other fields of this structure to specify an individual socket. It is ignored when querying for a list of sockets, as well as when all its elements are set to −1.

The response to a query for IPv4 or IPv6 sockets is represented as an array of

```
struct inet_diag_msg {
    __u8    idiag_family;
    __u8    idiag_state;
    __u8    idiag_timer;
    __u8    idiag_retrans;

    struct inet_diag_sockid id;

    __u32   idiag_expires;
    __u32   idiag_rqueue;
    __u32   idiag_wqueue;
    __u32   idiag_uid;
    __u32   idiag_inode;
};
```

followed by netlink attributes.

The fields of this structure are as follows:

idiag_family
: This is the same field as in *struct inet_diag_req_v2*.

idiag_state
: This denotes socket state as in *struct inet_diag_req_v2*.

idiag_timer
: For TCP sockets, this field describes the type of timer that is currently active for the socket. It is set to one of the following constants:

  0 no timer is active
  1 a retransmit timer
  2 a keep-alive timer
  3 a TIME_WAIT timer

4 a zero window probe timer

For non-TCP sockets, this field is set to 0.

idiag_retrans
For *idiag_timer* values 1, 2, and 4, this field contains the number of retransmits. For other *idiag_timer* values, this field is set to 0.

idiag_expires
For TCP sockets that have an active timer, this field describes its expiration time in milliseconds. For other sockets, this field is set to 0.

idiag_rqueue
For listening sockets: the number of pending connections.

For other sockets: the amount of data in the incoming queue.

idiag_wqueue
For listening sockets: the backlog length.

For other sockets: the amount of memory available for sending.

idiag_uid
This is the socket owner UID.

idiag_inode
This is the socket inode number.

Socket memory information

The payload associated with **UNIX_DIAG_MEMINFO** and **INET_DIAG_SKMEMINFO** netlink attributes is an array of the following __u32 values:

SK_MEMINFO_RMEM_ALLOC
The amount of data in receive queue.

SK_MEMINFO_RCVBUF
The receive socket buffer as set by **SO_RCVBUF**.

SK_MEMINFO_WMEM_ALLOC
The amount of data in send queue.

SK_MEMINFO_SNDBUF
The send socket buffer as set by **SO_SNDBUF**.

SK_MEMINFO_FWD_ALLOC
The amount of memory scheduled for future use (TCP only).

SK_MEMINFO_WMEM_QUEUED
The amount of data queued by TCP, but not yet sent.

SK_MEMINFO_OPTMEM
The amount of memory allocated for the socket's service needs (e.g., socket filter).

SK_MEMINFO_BACKLOG
The amount of packets in the backlog (not yet processed).

VERSIONS

NETLINK_INET_DIAG was introduced in Linux 2.6.14 and supported **AF_INET** and **AF_INET6** sockets only. In Linux 3.3, it was renamed to **NETLINK_SOCK_DIAG** and extended to support **AF_UNIX** sockets.

UNIX_DIAG_MEMINFO and **INET_DIAG_SKMEMINFO** were introduced in Linux 3.6.

STANDARDS

Linux.

EXAMPLES

The following example program prints inode number, peer's inode number, and name of all UNIX domain sockets in the current namespace.

```
#include <errno.h>
#include <stdio.h>
```

```
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <linux/sock_diag.h>
#include <linux/unix_diag.h>

static int
send_query(int fd)
{
    struct sockaddr_nl nladdr = {
        .nl_family = AF_NETLINK
    };
    struct
    {
        struct nlmsghdr nlh;
        struct unix_diag_req udr;
    } req = {
        .nlh = {
            .nlmsg_len = sizeof(req),
            .nlmsg_type = SOCK_DIAG_BY_FAMILY,
            .nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP
        },
        .udr = {
            .sdiag_family = AF_UNIX,
            .udiag_states = -1,
            .udiag_show = UDIAG_SHOW_NAME | UDIAG_SHOW_PEER
        }
    };
    struct iovec iov = {
        .iov_base = &req,
        .iov_len = sizeof(req)
    };
    struct msghdr msg = {
        .msg_name = &nladdr,
        .msg_namelen = sizeof(nladdr),
        .msg_iov = &iov,
        .msg_iovlen = 1
    };

    for (;;) {
        if (sendmsg(fd, &msg, 0) < 0) {
            if (errno == EINTR)
                continue;

            perror("sendmsg");
            return -1;
        }

        return 0;
    }
}

static int
print_diag(const struct unix_diag_msg *diag, unsigned int len)
{
    if (len < NLMSG_LENGTH(sizeof(*diag))) {
```

```
            fputs("short response\n", stderr);
            return −1;
        }
        if (diag−>udiag_family != AF_UNIX) {
            fprintf(stderr, "unexpected family %u\n", diag−>udiag_family);
            return −1;
        }

        unsigned int rta_len = len − NLMSG_LENGTH(sizeof(*diag));
        unsigned int peer = 0;
        size_t path_len = 0;
        char path[sizeof(((struct sockaddr_un *) 0)−>sun_path) + 1];

        for (struct rtattr *attr = (struct rtattr *) (diag + 1);
             RTA_OK(attr, rta_len); attr = RTA_NEXT(attr, rta_len)) {
            switch (attr−>rta_type) {
            case UNIX_DIAG_NAME:
                if (!path_len) {
                    path_len = RTA_PAYLOAD(attr);
                    if (path_len > sizeof(path) − 1)
                        path_len = sizeof(path) − 1;
                    memcpy(path, RTA_DATA(attr), path_len);
                    path[path_len] = '\0';
                }
                break;

            case UNIX_DIAG_PEER:
                if (RTA_PAYLOAD(attr) >= sizeof(peer))
                    peer = *(unsigned int *) RTA_DATA(attr);
                break;
            }
        }

        printf("inode=%u", diag−>udiag_ino);

        if (peer)
            printf(", peer=%u", peer);

        if (path_len)
            printf(", name=%s%s", *path ? "" : "@",
                    *path ? path : path + 1);

        putchar('\n');
        return 0;
    }

    static int
    receive_responses(int fd)
    {
        long buf[8192 / sizeof(long)];
        struct sockaddr_nl nladdr;
        struct iovec iov = {
            .iov_base = buf,
            .iov_len = sizeof(buf)
        };
        int flags = 0;

        for (;;) {
            struct msghdr msg = {
```

```
                .msg_name = &nladdr,
                .msg_namelen = sizeof(nladdr),
                .msg_iov = &iov,
                .msg_iovlen = 1
            };

            ssize_t ret = recvmsg(fd, &msg, flags);

            if (ret < 0) {
                if (errno == EINTR)
                    continue;

                perror("recvmsg");
                return -1;
            }
            if (ret == 0)
                return 0;

            if (nladdr.nl_family != AF_NETLINK) {
                fputs("!AF_NETLINK\n", stderr);
                return -1;
            }

            const struct nlmsghdr *h = (struct nlmsghdr *) buf;

            if (!NLMSG_OK(h, ret)) {
                fputs("!NLMSG_OK\n", stderr);
                return -1;
            }

            for (; NLMSG_OK(h, ret); h = NLMSG_NEXT(h, ret)) {
                if (h->nlmsg_type == NLMSG_DONE)
                    return 0;

                if (h->nlmsg_type == NLMSG_ERROR) {
                    const struct nlmsgerr *err = NLMSG_DATA(h);

                    if (h->nlmsg_len < NLMSG_LENGTH(sizeof(*err))) {
                        fputs("NLMSG_ERROR\n", stderr);
                    } else {
                        errno = -err->error;
                        perror("NLMSG_ERROR");
                    }

                    return -1;
                }

                if (h->nlmsg_type != SOCK_DIAG_BY_FAMILY) {
                    fprintf(stderr, "unexpected nlmsg_type %u\n",
                            (unsigned) h->nlmsg_type);
                    return -1;
                }

                if (print_diag(NLMSG_DATA(h), h->nlmsg_len))
                    return -1;
            }
        }
    }
```

```
int
main(void)
{
    int fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG);

    if (fd < 0) {
        perror("socket");
        return 1;
    }

    int ret = send_query(fd) || receive_responses(fd);

    close(fd);
    return ret;
}
```

SEE ALSO

netlink(3), *rtnetlink*(3), *netlink*(7), *tcp*(7)

NAME

socket − Linux socket interface

SYNOPSIS

#include <sys/socket.h>

sockfd = **socket(int** *socket_family***, int** *socket_type***, int** *protocol***);**

DESCRIPTION

This manual page describes the Linux networking socket layer user interface. The BSD compatible sockets are the uniform interface between the user process and the network protocol stacks in the kernel. The protocol modules are grouped into *protocol families* such as **AF_INET**, **AF_IPX**, and **AF_PACKET**, and *socket types* such as **SOCK_STREAM** or **SOCK_DGRAM**. See *socket*(2) for more information on families and types.

Socket-layer functions

These functions are used by the user process to send or receive packets and to do other socket operations. For more information, see their respective manual pages.

socket(2) creates a socket, *connect*(2) connects a socket to a remote socket address, the *bind*(2) function binds a socket to a local socket address, *listen*(2) tells the socket that new connections shall be accepted, and *accept*(2) is used to get a new socket with a new incoming connection. *socketpair*(2) returns two connected anonymous sockets (implemented only for a few local families like **AF_UNIX**)

send(2), *sendto*(2), and *sendmsg*(2) send data over a socket, and *recv*(2), *recvfrom*(2), *recvmsg*(2) receive data from a socket. *poll*(2) and *select*(2) wait for arriving data or a readiness to send data. In addition, the standard I/O operations like *write*(2), *writev*(2), *sendfile*(2), *read*(2), and *readv*(2) can be used to read and write data.

getsockname(2) returns the local socket address and *getpeername*(2) returns the remote socket address. *getsockopt*(2) and *setsockopt*(2) are used to set or get socket layer or protocol options. *ioctl*(2) can be used to set or read some other options.

close(2) is used to close a socket. *shutdown*(2) closes parts of a full-duplex socket connection.

Seeking, or calling *pread*(2) or *pwrite*(2) with a nonzero position is not supported on sockets.

It is possible to do nonblocking I/O on sockets by setting the **O_NONBLOCK** flag on a socket file descriptor using *fcntl*(2). Then all operations that would block will (usually) return with **EAGAIN** (operation should be retried later); *connect*(2) will return **EINPROGRESS** error. The user can then wait for various events via *poll*(2) or *select*(2).

I/O events		
Event	Poll flag	Occurrence
Read	POLLIN	New data arrived.
Read	POLLIN	A connection setup has been completed (for connection-oriented sockets)
Read	POLLHUP	A disconnection request has been initiated by the other end.
Read	POLLHUP	A connection is broken (only for connection-oriented protocols). When the socket is written **SIGPIPE** is also sent.
Write	POLLOUT	Socket has enough send buffer space for writing new data.
Read/Write	POLLIN | POLLOUT	An outgoing *connect*(2) finished.
Read/Write	POLLERR	An asynchronous error occurred.
Read/Write	POLLHUP	The other end has shut down one direction.
Exception	POLLPRI	Urgent data arrived. **SIGURG** is sent then.

An alternative to *poll*(2) and *select*(2) is to let the kernel inform the application about events via a **SIGIO** signal. For that the **O_ASYNC** flag must be set on a socket file descriptor via *fcntl*(2) and a valid signal handler for **SIGIO** must be installed via *sigaction*(2). See the *Signals* discussion below.

Socket address structures

Each socket domain has its own format for socket addresses, with a domain-specific address structure. Each of these structures begins with an integer "family" field (typed as *sa_family_t*) that indicates the type of the address structure. This allows the various system calls (e.g., *connect*(2), *bind*(2), *accept*(2),

getsockname(2), *getpeername*(2)), which are generic to all socket domains, to determine the domain of a particular socket address.

To allow any type of socket address to be passed to interfaces in the sockets API, the type *struct sockaddr* is defined. The purpose of this type is purely to allow casting of domain-specific socket address types to a "generic" type, so as to avoid compiler warnings about type mismatches in calls to the sockets API.

In addition, the sockets API provides the data type *struct sockaddr_storage*. This type is suitable to accommodate all supported domain-specific socket address structures; it is large enough and is aligned properly. (In particular, it is large enough to hold IPv6 socket addresses.) The structure includes the following field, which can be used to identify the type of socket address actually stored in the structure:

```
sa_family_t ss_family;
```

The *sockaddr_storage* structure is useful in programs that must handle socket addresses in a generic way (e.g., programs that must deal with both IPv4 and IPv6 socket addresses).

Socket options

The socket options listed below can be set by using *setsockopt*(2) and read with *getsockopt*(2) with the socket level set to **SOL_SOCKET** for all sockets. Unless otherwise noted, *optval* is a pointer to an *int*.

SO_ACCEPTCONN
: Returns a value indicating whether or not this socket has been marked to accept connections with *listen*(2). The value 0 indicates that this is not a listening socket, the value 1 indicates that this is a listening socket. This socket option is read-only.

SO_ATTACH_FILTER (since Linux 2.2)
SO_ATTACH_BPF (since Linux 3.19)
: Attach a classic BPF (**SO_ATTACH_FILTER**) or an extended BPF (**SO_ATTACH_BPF**) program to the socket for use as a filter of incoming packets. A packet will be dropped if the filter program returns zero. If the filter program returns a nonzero value which is less than the packet's data length, the packet will be truncated to the length returned. If the value returned by the filter is greater than or equal to the packet's data length, the packet is allowed to proceed unmodified.

  The argument for **SO_ATTACH_FILTER** is a *sock_fprog* structure, defined in <*linux/filter.h*>:

```
struct sock_fprog {
    unsigned short      len;
    struct sock_filter *filter;
};
```

  The argument for **SO_ATTACH_BPF** is a file descriptor returned by the *bpf*(2) system call and must refer to a program of type **BPF_PROG_TYPE_SOCKET_FILTER**.

  These options may be set multiple times for a given socket, each time replacing the previous filter program. The classic and extended versions may be called on the same socket, but the previous filter will always be replaced such that a socket never has more than one filter defined.

  Both classic and extended BPF are explained in the kernel source file *Documentation/networking/filter.txt*

SO_ATTACH_REUSEPORT_CBPF
SO_ATTACH_REUSEPORT_EBPF
: For use with the **SO_REUSEPORT** option, these options allow the user to set a classic BPF (**SO_ATTACH_REUSEPORT_CBPF**) or an extended BPF (**SO_ATTACH_REUSEPORT_EBPF**) program which defines how packets are assigned to the sockets in the reuseport group (that is, all sockets which have **SO_REUSEPORT** set and are using the same local address to receive packets).

  The BPF program must return an index between 0 and N−1 representing the socket which should receive the packet (where N is the number of sockets in the group). If the BPF program returns an invalid index, socket selection will fall back to the plain **SO_REUSEPORT**

mechanism.

Sockets are numbered in the order in which they are added to the group (that is, the order of *bind*(2) calls for UDP sockets or the order of *listen*(2) calls for TCP sockets). New sockets added to a reuseport group will inherit the BPF program. When a socket is removed from a reuseport group (via *close*(2)), the last socket in the group will be moved into the closed socket's position.

These options may be set repeatedly at any time on any socket in the group to replace the current BPF program used by all sockets in the group.

SO_ATTACH_REUSEPORT_CBPF takes the same argument type as **SO_ATTACH_FILTER** and **SO_ATTACH_REUSEPORT_EBPF** takes the same argument type as **SO_ATTACH_BPF**.

UDP support for this feature is available since Linux 4.5; TCP support is available since Linux 4.6.

SO_BINDTODEVICE
Bind this socket to a particular device like "eth0", as specified in the passed interface name. If the name is an empty string or the option length is zero, the socket device binding is removed. The passed option is a variable-length null-terminated interface name string with the maximum size of **IFNAMSIZ**. If a socket is bound to an interface, only packets received from that particular interface are processed by the socket. Note that this works only for some socket types, particularly **AF_INET** sockets. It is not supported for packet sockets (use normal *bind*(2) there).

Before Linux 3.8, this socket option could be set, but could not retrieved with *getsockopt*(2). Since Linux 3.8, it is readable. The *optlen* argument should contain the buffer size available to receive the device name and is recommended to be **IFNAMSIZ** bytes. The real device name length is reported back in the *optlen* argument.

SO_BROADCAST
Set or get the broadcast flag. When enabled, datagram sockets are allowed to send packets to a broadcast address. This option has no effect on stream-oriented sockets.

SO_BSDCOMPAT
Enable BSD bug-to-bug compatibility. This is used by the UDP protocol module in Linux 2.0 and 2.2. If enabled, ICMP errors received for a UDP socket will not be passed to the user program. In later kernel versions, support for this option has been phased out: Linux 2.4 silently ignores it, and Linux 2.6 generates a kernel warning (printk()) if a program uses this option. Linux 2.0 also enabled BSD bug-to-bug compatibility options (random header changing, skipping of the broadcast flag) for raw sockets with this option, but that was removed in Linux 2.2.

SO_DEBUG
Enable socket debugging. Allowed only for processes with the **CAP_NET_ADMIN** capability or an effective user ID of 0.

SO_DETACH_FILTER (since Linux 2.2)
SO_DETACH_BPF (since Linux 3.19)
These two options, which are synonyms, may be used to remove the classic or extended BPF program attached to a socket with either **SO_ATTACH_FILTER** or **SO_ATTACH_BPF**. The option value is ignored.

SO_DOMAIN (since Linux 2.6.32)
Retrieves the socket domain as an integer, returning a value such as **AF_INET6**. See *socket*(2) for details. This socket option is read-only.

SO_ERROR
Get and clear the pending socket error. This socket option is read-only. Expects an integer.

SO_DONTROUTE
Don't send via a gateway, send only to directly connected hosts. The same effect can be achieved by setting the **MSG_DONTROUTE** flag on a socket *send*(2) operation. Expects an integer boolean flag.

SO_INCOMING_CPU (gettable since Linux 3.19, settable since Linux 4.4)
Sets or gets the CPU affinity of a socket. Expects an integer flag.

```
int cpu = 1;
setsockopt(fd, SOL_SOCKET, SO_INCOMING_CPU, &cpu,
           sizeof(cpu));
```

Because all of the packets for a single stream (i.e., all packets for the same 4-tuple) arrive on the single RX queue that is associated with a particular CPU, the typical use case is to employ one listening process per RX queue, with the incoming flow being handled by a listener on the same CPU that is handling the RX queue. This provides optimal NUMA behavior and keeps CPU caches hot.

SO_INCOMING_NAPI_ID (gettable since Linux 4.12)
Returns a system-level unique ID called NAPI ID that is associated with a RX queue on which the last packet associated with that socket is received.

This can be used by an application to split the incoming flows among worker threads based on the RX queue on which the packets associated with the flows are received. It allows each worker thread to be associated with a NIC HW receive queue and service all the connection requests received on that RX queue. This mapping between an app thread and a HW NIC queue streamlines the flow of data from the NIC to the application.

SO_KEEPALIVE
Enable sending of keep-alive messages on connection-oriented sockets. Expects an integer boolean flag.

SO_LINGER
Sets or gets the **SO_LINGER** option. The argument is a *linger* structure.

```
struct linger {
    int l_onoff;    /* linger active */
    int l_linger;   /* how many seconds to linger for */
};
```

When enabled, a *close*(2) or *shutdown*(2) will not return until all queued messages for the socket have been successfully sent or the linger timeout has been reached. Otherwise, the call returns immediately and the closing is done in the background. When the socket is closed as part of *exit*(2), it always lingers in the background.

SO_LOCK_FILTER
When set, this option will prevent changing the filters associated with the socket. These filters include any set using the socket options **SO_ATTACH_FILTER**, **SO_ATTACH_BPF**, **SO_ATTACH_REUSEPORT_CBPF**, and **SO_ATTACH_REUSEPORT_EBPF**.

The typical use case is for a privileged process to set up a raw socket (an operation that requires the **CAP_NET_RAW** capability), apply a restrictive filter, set the **SO_LOCK_FILTER** option, and then either drop its privileges or pass the socket file descriptor to an unprivileged process via a UNIX domain socket.

Once the **SO_LOCK_FILTER** option has been enabled, attempts to change or remove the filter attached to a socket, or to disable the **SO_LOCK_FILTER** option will fail with the error **EPERM**.

SO_MARK (since Linux 2.6.25)
Set the mark for each packet sent through this socket (similar to the netfilter MARK target but socket-based). Changing the mark can be used for mark-based routing without netfilter or for packet filtering. Setting this option requires the **CAP_NET_ADMIN** or **CAP_NET_RAW** (since Linux 5.17) capability.

SO_OOBINLINE
If this option is enabled, out-of-band data is directly placed into the receive data stream. Otherwise, out-of-band data is passed only when the **MSG_OOB** flag is set during receiving.

SO_PASSCRED
Enable or disable the receiving of the **SCM_CREDENTIALS** control message. For more information, see *unix*(7).

SO_PASSSEC
Enable or disable the receiving of the **SCM_SECURITY** control message. For more information, see *unix*(7).

SO_PEEK_OFF (since Linux 3.4)
This option, which is currently supported only for *unix*(7) sockets, sets the value of the "peek offset" for the *recv*(2) system call when used with **MSG_PEEK** flag.

When this option is set to a negative value (it is set to −1 for all new sockets), traditional behavior is provided: *recv*(2) with the **MSG_PEEK** flag will peek data from the front of the queue.

When the option is set to a value greater than or equal to zero, then the next peek at data queued in the socket will occur at the byte offset specified by the option value. At the same time, the "peek offset" will be incremented by the number of bytes that were peeked from the queue, so that a subsequent peek will return the next data in the queue.

If data is removed from the front of the queue via a call to *recv*(2) (or similar) without the **MSG_PEEK** flag, the "peek offset" will be decreased by the number of bytes removed. In other words, receiving data without the **MSG_PEEK** flag will cause the "peek offset" to be adjusted to maintain the correct relative position in the queued data, so that a subsequent peek will retrieve the data that would have been retrieved had the data not been removed.

For datagram sockets, if the "peek offset" points to the middle of a packet, the data returned will be marked with the **MSG_TRUNC** flag.

The following example serves to illustrate the use of **SO_PEEK_OFF**. Suppose a stream socket has the following queued input data:

```
aabbccddeeff
```

The following sequence of *recv*(2) calls would have the effect noted in the comments:

```
int ov = 4;                  // Set peek offset to 4
setsockopt(fd, SOL_SOCKET, SO_PEEK_OFF, &ov, sizeof(ov));

recv(fd, buf, 2, MSG_PEEK);  // Peeks "cc"; offset set to 6
recv(fd, buf, 2, MSG_PEEK);  // Peeks "dd"; offset set to 8
recv(fd, buf, 2, 0);         // Reads "aa"; offset set to 6
recv(fd, buf, 2, MSG_PEEK);  // Peeks "ee"; offset set to 8
```

SO_PEERCRED
Return the credentials of the peer process connected to this socket. For further details, see *unix*(7).

SO_PEERSEC (since Linux 2.6.2)
Return the security context of the peer socket connected to this socket. For further details, see *unix*(7) and *ip*(7).

SO_PRIORITY
Set the protocol-defined priority for all packets to be sent on this socket. Linux uses this value to order the networking queues: packets with a higher priority may be processed first depending on the selected device queueing discipline. Setting a priority outside the range 0 to 6 requires the **CAP_NET_ADMIN** capability.

SO_PROTOCOL (since Linux 2.6.32)
Retrieves the socket protocol as an integer, returning a value such as **IPPROTO_SCTP**. See *socket*(2) for details. This socket option is read-only.

SO_RCVBUF
Sets or gets the maximum socket receive buffer in bytes. The kernel doubles this value (to allow space for bookkeeping overhead) when it is set using *setsockopt*(2), and this doubled value is returned by *getsockopt*(2). The default value is set by the */proc/sys/net/core/rmem_default* file, and the maximum allowed value is set by the */proc/sys/net/core/rmem_max* file. The minimum (doubled) value for this option is 256.

SO_RCVBUFFORCE (since Linux 2.6.14)
Using this socket option, a privileged (**CAP_NET_ADMIN**) process can perform the same task as **SO_RCVBUF**, but the *rmem_max* limit can be overridden.

SO_RCVLOWAT and **SO_SNDLOWAT**
Specify the minimum number of bytes in the buffer until the socket layer will pass the data to the protocol (**SO_SNDLOWAT**) or the user on receiving (**SO_RCVLOWAT**). These two values are initialized to 1. **SO_SNDLOWAT** is not changeable on Linux (**setsockopt**(2) fails with the error **ENOPROTOOPT**). **SO_RCVLOWAT** is changeable only since Linux 2.4.

Before Linux 2.6.28 *select*(2), *poll*(2), and *epoll*(7) did not respect the **SO_RCVLOWAT** setting on Linux, and indicated a socket as readable when even a single byte of data was available. A subsequent read from the socket would then block until **SO_RCVLOWAT** bytes are available. Since Linux 2.6.28, *select*(2), *poll*(2), and *epoll*(7) indicate a socket as readable only if at least **SO_RCVLOWAT** bytes are available.

SO_RCVTIMEO and **SO_SNDTIMEO**
Specify the receiving or sending timeouts until reporting an error. The argument is a *struct timeval*. If an input or output function blocks for this period of time, and data has been sent or received, the return value of that function will be the amount of data transferred; if no data has been transferred and the timeout has been reached, then −1 is returned with *errno* set to **EAGAIN** or **EWOULDBLOCK**, or **EINPROGRESS** (for *connect*(2)) just as if the socket was specified to be nonblocking. If the timeout is set to zero (the default), then the operation will never timeout. Timeouts only have effect for system calls that perform socket I/O (e.g., *accept*(2), *connect*(2), *read*(2), *recvmsg*(2), *send*(2), *sendmsg*(2)); timeouts have no effect for *select*(2), *poll*(2), *epoll_wait*(2), and so on.

SO_REUSEADDR
Indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. For **AF_INET** sockets this means that a socket may bind, except when there is an active listening socket bound to the address. When the listening socket is bound to **INADDR_ANY** with a specific port then it is not possible to bind to this port for any local address. Argument is an integer boolean flag.

SO_REUSEPORT (since Linux 3.9)
Permits multiple **AF_INET** or **AF_INET6** sockets to be bound to an identical socket address. This option must be set on each socket (including the first socket) prior to calling *bind*(2) on the socket. To prevent port hijacking, all of the processes binding to the same address must have the same effective UID. This option can be employed with both TCP and UDP sockets.

For TCP sockets, this option allows *accept*(2) load distribution in a multi-threaded server to be improved by using a distinct listener socket for each thread. This provides improved load distribution as compared to traditional techniques such using a single *accept*(2)ing thread that distributes connections, or having multiple threads that compete to *accept*(2) from the same socket.

For UDP sockets, the use of this option can provide better distribution of incoming datagrams to multiple processes (or threads) as compared to the traditional technique of having multiple processes compete to receive datagrams on the same socket.

SO_RXQ_OVFL (since Linux 2.6.33)
Indicates that an unsigned 32-bit value ancillary message (cmsg) should be attached to received skbs indicating the number of packets dropped by the socket since its creation.

SO_SELECT_ERR_QUEUE (since Linux 3.10)
When this option is set on a socket, an error condition on a socket causes notification not only via the *exceptfds* set of *select*(2). Similarly, *poll*(2) also returns a **POLLPRI** whenever an **POLLERR** event is returned.

Background: this option was added when waking up on an error condition occurred only via the *readfds* and *writefds* sets of *select*(2). The option was added to allow monitoring for error conditions via the *exceptfds* argument without simultaneously having to receive notifications (via *readfds*) for regular data that can be read from the socket. After changes in Linux 4.16, the use of this flag to achieve the desired notifications is no longer necessary. This option is nevertheless retained for backwards compatibility.

SO_SNDBUF
Sets or gets the maximum socket send buffer in bytes. The kernel doubles this value (to allow space for bookkeeping overhead) when it is set using *setsockopt*(2), and this doubled value is returned by *getsockopt*(2). The default value is set by the */proc/sys/net/core/wmem_default* file and the maximum allowed value is set by the */proc/sys/net/core/wmem_max* file. The minimum (doubled) value for this option is 2048.

SO_SNDBUFFORCE (since Linux 2.6.14)
Using this socket option, a privileged (**CAP_NET_ADMIN**) process can perform the same task as **SO_SNDBUF**, but the *wmem_max* limit can be overridden.

SO_TIMESTAMP
Enable or disable the receiving of the **SO_TIMESTAMP** control message. The timestamp control message is sent with level **SOL_SOCKET** and a *cmsg_type* of **SCM_TIMESTAMP**. The *cmsg_data* field is a *struct timeval* indicating the reception time of the last packet passed to the user in this call. See *cmsg*(3) for details on control messages.

SO_TIMESTAMPNS (since Linux 2.6.22)
Enable or disable the receiving of the **SO_TIMESTAMPNS** control message. The timestamp control message is sent with level **SOL_SOCKET** and a *cmsg_type* of **SCM_TIMESTAMPNS**. The *cmsg_data* field is a *struct timespec* indicating the reception time of the last packet passed to the user in this call. The clock used for the timestamp is **CLOCK_REALTIME**. See *cmsg*(3) for details on control messages.

A socket cannot mix **SO_TIMESTAMP** and **SO_TIMESTAMPNS**: the two modes are mutually exclusive.

SO_TYPE
Gets the socket type as an integer (e.g., **SOCK_STREAM**). This socket option is read-only.

SO_BUSY_POLL (since Linux 3.11)
Sets the approximate time in microseconds to busy poll on a blocking receive when there is no data. Increasing this value requires **CAP_NET_ADMIN**. The default for this option is controlled by the */proc/sys/net/core/busy_read* file.

The value in the */proc/sys/net/core/busy_poll* file determines how long *select*(2) and *poll*(2) will busy poll when they operate on sockets with **SO_BUSY_POLL** set and no events to report are found.

In both cases, busy polling will only be done when the socket last received data from a network device that supports this option.

While busy polling may improve latency of some applications, care must be taken when using it since this will increase both CPU utilization and power usage.

Signals

When writing onto a connection-oriented socket that has been shut down (by the local or the remote end) **SIGPIPE** is sent to the writing process and **EPIPE** is returned. The signal is not sent when the write call specified the **MSG_NOSIGNAL** flag.

When requested with the **FIOSETOWN** *fcntl*(2) or **SIOCSPGRP** *ioctl*(2), **SIGIO** is sent when an I/O event occurs. It is possible to use *poll*(2) or *select*(2) in the signal handler to find out which socket the event occurred on. An alternative (in Linux 2.2) is to set a real-time signal using the **F_SETSIG** *fcntl*(2); the handler of the real time signal will be called with the file descriptor in the *si_fd* field of its *siginfo_t*. See *fcntl*(2) for more information.

Under some circumstances (e.g., multiple processes accessing a single socket), the condition that caused the **SIGIO** may have already disappeared when the process reacts to the signal. If this happens, the process should wait again because Linux will resend the signal later.

/proc interfaces

The core socket networking parameters can be accessed via files in the directory */proc/sys/net/core/*.

rmem_default
contains the default setting in bytes of the socket receive buffer.

rmem_max
: contains the maximum socket receive buffer size in bytes which a user may set by using the **SO_RCVBUF** socket option.

wmem_default
: contains the default setting in bytes of the socket send buffer.

wmem_max
: contains the maximum socket send buffer size in bytes which a user may set by using the **SO_SNDBUF** socket option.

message_cost and *message_burst*
: configure the token bucket filter used to load limit warning messages caused by external network events.

netdev_max_backlog
: Maximum number of packets in the global input queue.

optmem_max
: Maximum length of ancillary data and user control data like the iovecs per socket.

Ioctls

These operations can be accessed using *ioctl*(2):

```
error = ioctl(ip_socket, ioctl_type, &value_result);
```

SIOCGSTAMP
: Return a *struct timeval* with the receive timestamp of the last packet passed to the user. This is useful for accurate round trip time measurements. See *setitimer*(2) for a description of *struct timeval*. This ioctl should be used only if the socket options **SO_TIMESTAMP** and **SO_TIMESTAMPNS** are not set on the socket. Otherwise, it returns the timestamp of the last packet that was received while **SO_TIMESTAMP** and **SO_TIMESTAMPNS** were not set, or it fails if no such packet has been received, (i.e., *ioctl*(2) returns −1 with *errno* set to **ENOENT**).

SIOCSPGRP
: Set the process or process group that is to receive **SIGIO** or **SIGURG** signals when I/O becomes possible or urgent data is available. The argument is a pointer to a *pid_t*. For further details, see the description of **F_SETOWN** in *fcntl*(2).

FIOASYNC
: Change the **O_ASYNC** flag to enable or disable asynchronous I/O mode of the socket. Asynchronous I/O mode means that the **SIGIO** signal or the signal set with **F_SETSIG** is raised when a new I/O event occurs.

  Argument is an integer boolean flag. (This operation is synonymous with the use of *fcntl*(2) to set the **O_ASYNC** flag.)

SIOCGPGRP
: Get the current process or process group that receives **SIGIO** or **SIGURG** signals, or 0 when none is set.

Valid *fcntl*(2) operations:

FIOGETOWN
: The same as the **SIOCGPGRP** *ioctl*(2).

FIOSETOWN
: The same as the **SIOCSPGRP** *ioctl*(2).

VERSIONS

SO_BINDTODEVICE was introduced in Linux 2.0.30. **SO_PASSCRED** is new in Linux 2.2. The */proc* interfaces were introduced in Linux 2.2. **SO_RCVTIMEO** and **SO_SNDTIMEO** are supported since Linux 2.3.41. Earlier, timeouts were fixed to a protocol-specific setting, and could not be read or written.

NOTES

Linux assumes that half of the send/receive buffer is used for internal kernel structures; thus the values in the corresponding */proc* files are twice what can be observed on the wire.

Linux will allow port reuse only with the **SO_REUSEADDR** option when this option was set both in the previous program that performed a *bind*(2) to the port and in the program that wants to reuse the port. This differs from some implementations (e.g., FreeBSD) where only the later program needs to set the **SO_REUSEADDR** option. Typically this difference is invisible, since, for example, a server program is designed to always set this option.

SEE ALSO

wireshark(1), *bpf*(2), *connect*(2), *getsockopt*(2), *setsockopt*(2), *socket*(2), *pcap*(3), *address_families*(7), *capabilities*(7), *ddp*(7), *ip*(7), *ipv6*(7), *packet*(7), *tcp*(7), *udp*(7), *unix*(7), *tcpdump*(8)

NAME

spufs − SPU filesystem

DESCRIPTION

The SPU filesystem is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs).

The filesystem provides a name space similar to POSIX shared memory or message queues. Users that have write permissions on the filesystem can use *spu_create*(2) to establish SPU contexts under the **spufs** root directory.

Every SPU context is represented by a directory containing a predefined set of files. These files can be used for manipulating the state of the logical SPU. Users can change permissions on the files, but can't add or remove files.

Mount options

uid=<uid>
: Set the user owning the mount point; the default is 0 (root).

gid=<gid>
: Set the group owning the mount point; the default is 0 (root).

mode=<mode>
: Set the mode of the top-level directory in **spufs**, as an octal mode string. The default is 0775.

Files

The files in **spufs** mostly follow the standard behavior for regular system calls like *read*(2) or *write*(2), but often support only a subset of the operations supported on regular filesystems. This list details the supported operations and the deviations from the standard behavior described in the respective man pages.

All files that support the *read*(2) operation also support *readv*(2) and all files that support the *write*(2) operation also support *writev*(2). All files support the *access*(2) and *stat*(2) family of operations, but for the latter call, the only fields of the returned *stat* structure that contain reliable information are *st_mode*, *st_nlink*, *st_uid*, and *st_gid*.

All files support the *chmod*(2)/*fchmod*(2) and *chown*(2)/*fchown*(2) operations, but will not be able to grant permissions that contradict the possible operations (e.g., read access on the *wbox* file).

The current set of files is:

/capabilities
: Contains a comma-delimited string representing the capabilities of this SPU context. Possible capabilities are:

  sched This context may be scheduled.

  step This context can be run in single-step mode, for debugging.

  New capabilities flags may be added in the future.

/mem
: the contents of the local storage memory of the SPU. This can be accessed like a regular shared memory file and contains both code and data in the address space of the SPU. The possible operations on an open *mem* file are:

  read(2)
  pread(2)
  write(2)
  pwrite(2)
  lseek(2)
  : These operate as usual, with the exception that *lseek*(2), *write*(2), and *pwrite*(2) are not supported beyond the end of the file. The file size is the size of the local storage of the SPU, which is normally 256 kilobytes.

  mmap(2)
  : Mapping *mem* into the process address space provides access to the SPU local storage within the process address space. Only **MAP_SHARED** mappings are allowed.

/regs Contains the saved general-purpose registers of the SPU context. This file contains the 128-bit values of each register, from register 0 to register 127, in order. This allows the general-purpose registers to be inspected for debugging.

Reading to or writing from this file requires that the context is scheduled out, so use of this file is not recommended in normal program operation.

The *regs* file is not present on contexts that have been created with the **SPU_CREATE_NOSCHED** flag.

/mbox The first SPU-to-CPU communication mailbox. This file is read-only and can be read in units of 4 bytes. The file can be used only in nonblocking mode − even *poll*(2) cannot be used to block on this file. The only possible operation on an open *mbox* file is:

read(2) If *count* is smaller than four, *read*(2) returns −1 and sets *errno* to **EINVAL**. If there is no data available in the mailbox (i.e., the SPU has not sent a mailbox message), the return value is set to −1 and *errno* is set to **EAGAIN**. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

/ibox The second SPU-to-CPU communication mailbox. This file is similar to the first mailbox file, but can be read in blocking I/O mode, thus calling *read*(2) on an open *ibox* file will block until the SPU has written data to its interrupt mailbox channel (unless the file has been opened with **O_NONBLOCK**, see below). Also, *poll*(2) and similar system calls can be used to monitor for the presence of mailbox data.

The possible operations on an open *ibox* file are:

read(2) If *count* is smaller than four, *read*(2) returns −1 and sets *errno* to **EINVAL**. If there is no data available in the mailbox and the file descriptor has been opened with **O_NONBLOCK**, the return value is set to −1 and *errno* is set to **EAGAIN**.

If there is no data available in the mailbox and the file descriptor has been opened without **O_NONBLOCK**, the call will block until the SPU writes to its interrupt mailbox channel. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

poll(2) Poll on the *ibox* file returns *(POLLIN | POLLRDNORM)* whenever data is available for reading.

/wbox The CPU-to-SPU communication mailbox. It is write-only and can be written in units of four bytes. If the mailbox is full, *write*(2) will block, and *poll*(2) can be used to block until the mailbox is available for writing again. The possible operations on an open *wbox* file are:

write(2)

If *count* is smaller than four, *write*(2) returns −1 and sets *errno* to **EINVAL**. If there is no space available in the mailbox and the file descriptor has been opened with **O_NONBLOCK**, the return value is set to −1 and *errno* is set to **EAGAIN**.

If there is no space available in the mailbox and the file descriptor has been opened without **O_NONBLOCK**, the call will block until the SPU reads from its PPE (PowerPC Processing Element) mailbox channel. When data has been written successfully, the system call returns four as its function result.

poll(2) A poll on the *wbox* file returns *(POLLOUT | POLLWRNORM)* whenever space is available for writing.

/mbox_stat
/ibox_stat
/wbox_stat

These are read-only files that contain the length of the current queue of each mailbox—that is, how many words can be read from *mbox* or *ibox* or how many words can be written to *wbox* without blocking. The files can be read only in four-byte units and return a big-endian binary integer number. The only possible operation on an open **box_stat* file is:

read(2) If *count* is smaller than four, *read*(2) returns −1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer. This value is the number of elements that can be read from (for *mbox_stat* and *ibox_stat*) or written to (for *wbox_stat*) the respective mailbox without blocking or returning an **EAGAIN** error.

/npc
/decr
/decr_status
/spu_tag_mask
/event_mask
/event_status
/srr0
/lslr Internal registers of the SPU. These files contain an ASCII string representing the hex value of the specified register. Reads and writes on these files (except for *npc*, see below) require that the SPU context be scheduled out, so frequent access to these files is not recommended for normal program operation.

The contents of these files are:

npc	Next Program Counter − valid only when the SPU is in a stopped state.
decr	SPU Decrementer
decr_status	Decrementer Status
spu_tag_mask	MFC tag mask for SPU DMA
event_mask	Event mask for SPU interrupts
event_status	Number of SPU events pending (read-only)
srr0	Interrupt Return address register
lslr	Local Store Limit Register

The possible operations on these files are:

read(2) Reads the current register value. If the register value is larger than the buffer passed to the *read*(2) system call, subsequent reads will continue reading from the same buffer, until the end of the buffer is reached.

When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read a new value.

write(2)
A *write*(2) operation on the file sets the register to the value given in the string. The string is parsed from the beginning until the first nonnumeric character or the end of the buffer. Subsequent writes to the same file descriptor overwrite the previous setting.

Except for the *npc* file, these files are not present on contexts that have been created with the **SPU_CREATE_NOSCHED** flag.

/fpcr This file provides access to the Floating Point Status and Control Register (fcpr) as a binary, four-byte file. The operations on the *fpcr* file are:

read(2) If *count* is smaller than four, *read*(2) returns −1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer; this is the current value of the *fpcr* register.

write(2)
If *count* is smaller than four, *write*(2) returns −1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is copied from the data buffer, updating the value of the *fpcr* register.

/signal1
/signal2
The files provide access to the two signal notification channels of an SPU. These are read-write files that operate on four-byte words. Writing to one of these files triggers an interrupt on the SPU. The value written to the signal files can be read from the SPU through a channel read or from host user space through the file. After the value has been read by the SPU, it is reset to zero. The possible operations on an open *signal1* or *signal2* file are:

read(2) If *count* is smaller than four, *read*(2) returns −1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer; this is the current value of the specified signal notification register.

write(2)
If *count* is smaller than four, *write*(2) returns −1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is copied from the data buffer, updating the value of the specified signal notification register. The signal notification register will either be replaced with the input data or will be updated to the bitwise OR operation of the old value and the input data, depending on the contents of the *signal1_type* or *signal2_type* files respectively.

/signal1_type
/signal2_type
These two files change the behavior of the *signal1* and *signal2* notification files. They contain a numeric ASCII string which is read as either "1" or "0". In mode 0 (overwrite), the hardware replaces the contents of the signal channel with the data that is written to it. In mode 1 (logical OR), the hardware accumulates the bits that are subsequently written to it. The possible operations on an open *signal1_type* or *signal2_type* file are:

read(2) When the count supplied to the *read*(2) call is shorter than the required length for the digit (plus a newline character), subsequent reads from the same file descriptor will complete the string. When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read the value again.

write(2)
A *write*(2) operation on the file sets the register to the value given in the string. The string is parsed from the beginning until the first nonnumeric character or the end of the buffer. Subsequent writes to the same file descriptor overwrite the previous setting.

/mbox_info
/ibox_info
/wbox_info
/dma_into
/proxydma_info
Read-only files that contain the saved state of the SPU mailboxes and DMA queues. This allows the SPU status to be inspected, mainly for debugging. The *mbox_info* and *ibox_info* files each contain the four-byte mailbox message that has been written by the SPU. If no message has been written to these mailboxes, then contents of these files is undefined. The *mbox_stat*, *ibox_stat*, and *wbox_stat* files contain the available message count.

The *wbox_info* file contains an array of four-byte mailbox messages, which have been sent to the SPU. With current CBEA machines, the array is four items in length, so up to 4 * 4 = 16 bytes can be read from this file. If any mailbox queue entry is empty, then the bytes read at the corresponding location are undefined.

The *dma_info* file contains the contents of the SPU MFC DMA queue, represented as the following structure:

```
struct spu_dma_info {
    uint64_t         dma_info_type;
    uint64_t         dma_info_mask;
    uint64_t         dma_info_status;
    uint64_t         dma_info_stall_and_notify;
    uint64_t         dma_info_atomic_command_status;
    struct mfc_cq_sr dma_info_command_data[16];
};
```

The last member of this data structure is the actual DMA queue, containing 16 entries. The *mfc_cq_sr* structure is defined as:

```
struct mfc_cq_sr {
    uint64_t mfc_cq_data0_RW;
```

```
    uint64_t mfc_cq_data1_RW;
    uint64_t mfc_cq_data2_RW;
    uint64_t mfc_cq_data3_RW;
};
```

The *proxydma_info* file contains similar information, but describes the proxy DMA queue (i.e., DMAs initiated by entities outside the SPU) instead. The file is in the following format:

```
struct spu_proxydma_info {
    uint64_t         proxydma_info_type;
    uint64_t         proxydma_info_mask;
    uint64_t         proxydma_info_status;
    struct mfc_cq_sr proxydma_info_command_data[8];
};
```

Accessing these files requires that the SPU context is scheduled out - frequent use can be inefficient. These files should not be used for normal program operation.

These files are not present on contexts that have been created with the **SPU_CREATE_NOSCHED** flag.

/cntl This file provides access to the SPU Run Control and SPU status registers, as an ASCII string. The following operations are supported:

read(2) Reads from the *cntl* file will return an ASCII string with the hex value of the SPU Status register.

write(2)
Writes to the *cntl* file will set the context's SPU Run Control register.

/mfc Provides access to the Memory Flow Controller of the SPU. Reading from the file returns the contents of the SPU's MFC Tag Status register, and writing to the file initiates a DMA from the MFC. The following operations are supported:

write(2)
Writes to this file need to be in the format of a MFC DMA command, defined as follows:

```
struct mfc_dma_command {
    int32_t  pad;    /* reserved */
    uint32_t lsa;    /* local storage address */
    uint64_t ea;     /* effective address */
    uint16_t size;   /* transfer size */
    uint16_t tag;    /* command tag */
    uint16_t class;  /* class ID */
    uint16_t cmd;    /* command opcode */
};
```

Writes are required to be exactly *sizeof(struct mfc_dma_command)* bytes in size. The command will be sent to the SPU's MFC proxy queue, and the tag stored in the kernel (see below).

read(2) Reads the contents of the tag status register. If the file is opened in blocking mode (i.e., without **O_NONBLOCK**), then the read will block until a DMA tag (as performed by a previous write) is complete. In nonblocking mode, the MFC tag status register will be returned without waiting.

poll(2) Calling *poll*(2) on the *mfc* file will block until a new DMA can be started (by checking for **POLLOUT**) or until a previously started DMA (by checking for **POLLIN**) has been completed.

/mss Provides access to the MFC MultiSource Synchronization (MSS) facility. By *mmap*(2)-ing this file, processes can access the MSS area of the SPU.

The following operations are supported:

mmap(2)
: Mapping **mss** into the process address space gives access to the SPU MSS area within the process address space. Only **MAP_SHARED** mappings are allowed.

/psmap Provides access to the whole problem-state mapping of the SPU. Applications can use this area to interface to the SPU, rather than writing to individual register files in **spufs**.

The following operations are supported:

mmap(2)
: Mapping **psmap** gives a process a direct map of the SPU problem state area. Only **MAP_SHARED** mappings are supported.

/phys−id
: Read-only file containing the physical SPU number that the SPU context is running on. When the context is not running, this file contains the string "−1".

The physical SPU number is given by an ASCII hex string.

/object−id
: Allows applications to store (or retrieve) a single 64-bit ID into the context. This ID is later used by profiling tools to uniquely identify the context.

write(2)
: By writing an ASCII hex value into this file, applications can set the object ID of the SPU context. Any previous value of the object ID is overwritten.

read(2) Reading this file gives an ASCII hex string representing the object ID for this SPU context.

EXAMPLES

To automatically *mount*(8) the SPU filesystem when booting, at the location */spu* chosen by the user, put this line into the *fstab*(5) configuration file:

```
none /spu spufs gid=spu 0 0
```

SEE ALSO

close(2), *spu_create*(2), *spu_run*(2), *capabilities*(7)

The Cell Broadband Engine Architecture (CBEA) specification

NAME

standards − C and UNIX Standards

DESCRIPTION

The STANDARDS section that appears in many manual pages identifies various standards to which the documented interface conforms. The following list briefly describes these standards.

V7 Version 7 (also known as Seventh Edition) UNIX, released by AT&T/Bell Labs in 1979. After this point, UNIX systems diverged into two main dialects: BSD and System V.

4.2BSD

This is an implementation standard defined by the 4.2 release of the *Berkeley Software Distribution*, released by the University of California at Berkeley. This was the first Berkeley release that contained a TCP/IP stack and the sockets API. 4.2BSD was released in 1983.

Earlier major BSD releases included *3BSD* (1980), *4BSD* (1980), and *4.1BSD* (1981).

4.3BSD

The successor to 4.2BSD, released in 1986.

4.4BSD

The successor to 4.3BSD, released in 1993. This was the last major Berkeley release.

System V

This is an implementation standard defined by AT&T's milestone 1983 release of its commercial System V (five) release. The previous major AT&T release was *System III*, released in 1981.

System V release 2 (SVr2)

This was the next System V release, made in 1985. The SVr2 was formally described in the *System V Interface Definition version 1* (*SVID 1*) published in 1985.

System V release 3 (SVr3)

This was the successor to SVr2, released in 1986. This release was formally described in the *System V Interface Definition version 2* (*SVID 2*).

System V release 4 (SVr4)

This was the successor to SVr3, released in 1989. This version of System V is described in the "Programmer's Reference Manual: Operating System API (Intel processors)" (Prentice-Hall 1992, ISBN 0-13-951294-2) This release was formally described in the *System V Interface Definition version 3* (*SVID 3*), and is considered the definitive System V release.

SVID 4

System V Interface Definition version 4, issued in 1995. Available online at .

C89 This was the first C language standard, ratified by ANSI (American National Standards Institute) in 1989 (*X3.159-1989*). Sometimes this is known as *ANSI C*, but since C99 is also an ANSI standard, this term is ambiguous. This standard was also ratified by ISO (International Standards Organization) in 1990 (*ISO/IEC 9899:1990*), and is thus occasionally referred to as *ISO C90*.

C99 This revision of the C language standard was ratified by ISO in 1999 (*ISO/IEC 9899:1999*). Available online at .

C11 This revision of the C language standard was ratified by ISO in 2011 (*ISO/IEC 9899:2011*).

LFS The Large File Summit specification, completed in 1996. This specification defined mechanisms that allowed 32-bit systems to support the use of large files (i.e., 64-bit file offsets). See .

POSIX.1-1988

This was the first POSIX standard, ratified by IEEE as IEEE Std 1003.1-1988, and subsequently adopted (with minor revisions) as an ISO standard in 1990. The term "POSIX" was coined by Richard Stallman.

POSIX.1-1990

"Portable Operating System Interface for Computing Environments". IEEE 1003.1-1990 part 1, ratified by ISO in 1990 (*ISO/IEC 9945-1:1990*).

POSIX.2
IEEE Std 1003.2-1992, describing commands and utilities, ratified by ISO in 1993 (*ISO/IEC 9945-2:1993*).

POSIX.1b (formerly known as *POSIX.4*)
IEEE Std 1003.1b-1993, describing real-time facilities for portable operating systems, ratified by ISO in 1996 (*ISO/IEC 9945-1:1996*).

POSIX.1c (formerly known as *POSIX.4a*)
IEEE Std 1003.1c-1995, which describes the POSIX threads interfaces.

POSIX.1d
IEEE Std 1003.1d-1999, which describes additional real-time extensions.

POSIX.1g
IEEE Std 1003.1g-2000, which describes networking APIs (including sockets).

POSIX.1j
IEEE Std 1003.1j-2000, which describes advanced real-time extensions.

POSIX.1-1996
A 1996 revision of POSIX.1 which incorporated POSIX.1b and POSIX.1c.

XPG3 Released in 1989, this was the first release of the X/Open Portability Guide to be based on a POSIX standard (POSIX.1-1988). This multivolume guide was developed by the X/Open Group, a multivendor consortium.

XPG4 A revision of the X/Open Portability Guide, released in 1992. This revision incorporated POSIX.2.

XPG4v2
A 1994 revision of XPG4. This is also referred to as *Spec 1170*, where 1170 referred to the number of interfaces defined by this standard.

SUS (SUSv1)
Single UNIX Specification. This was a repackaging of XPG4v2 and other X/Open standards (X/Open Curses Issue 4 version 2, X/Open Networking Service (XNS) Issue 4). Systems conforming to this standard can be branded *UNIX 95*.

SUSv2 Single UNIX Specification version 2. Sometimes also referred to (incorrectly) as *XPG5*. This standard appeared in 1997. Systems conforming to this standard can be branded *UNIX 98*. See also .)

POSIX.1-2001

SUSv3 This was a 2001 revision and consolidation of the POSIX.1, POSIX.2, and SUS standards into a single document, conducted under the auspices of the Austin Group . The standard is available online at .

The standard defines two levels of conformance: *POSIX conformance*, which is a baseline set of interfaces required of a conforming system; and *XSI Conformance*, which additionally mandates a set of interfaces (the "XSI extension") which are only optional for POSIX conformance. XSI-conformant systems can be branded *UNIX 03*.

The POSIX.1-2001 document is broken into four parts:

XBD: Definitions, terms, and concepts, header file specifications.

XSH: Specifications of functions (i.e., system calls and library functions in actual implementations).

XCU: Specifications of commands and utilities (i.e., the area formerly described by POSIX.2).

XRAT: Informative text on the other parts of the standard.

POSIX.1-2001 is aligned with C99, so that all of the library functions standardized in C99 are also standardized in POSIX.1-2001.

The Single UNIX Specification version 3 (SUSv3) comprises the Base Specifications containing XBD, XSH, XCU, and XRAT as above, plus X/Open Curses Issue 4 version 2 as an extra volume that is not in POSIX.1-2001.

Two Technical Corrigenda (minor fixes and improvements) of the original 2001 standard have occurred: TC1 in 2003 and TC2 in 2004.

POSIX.1-2008

SUSv4 Work on the next revision of POSIX.1/SUS was completed and ratified in 2008. The standard is available online at .

The changes in this revision are not as large as those that occurred for POSIX.1-2001/SUSv3, but a number of new interfaces are added and various details of existing specifications are modified. Many of the interfaces that were optional in POSIX.1-2001 become mandatory in the 2008 revision of the standard. A few interfaces that are present in POSIX.1-2001 are marked as obsolete in POSIX.1-2008, or removed from the standard altogether.

The revised standard is structured in the same way as its predecessor. The Single UNIX Specification version 4 (SUSv4) comprises the Base Specifications containing XBD, XSH, XCU, and XRAT, plus X/Open Curses Issue 7 as an extra volume that is not in POSIX.1-2008.

Again there are two levels of conformance: the baseline *POSIX Conformance*, and *XSI Conformance*, which mandates an additional set of interfaces beyond those in the base specification.

In general, where the STANDARDS section of a manual page lists POSIX.1-2001, it can be assumed that the interface also conforms to POSIX.1-2008, unless otherwise noted.

Technical Corrigendum 1 (minor fixes and improvements) of this standard was released in 2013.

Technical Corrigendum 2 of this standard was released in 2016.

Further information can be found on the Austin Group web site, .

SUSv4 2016 edition

This is equivalent to POSIX.1-2008, with the addition of Technical Corrigenda 1 and 2 and the XCurses specification.

POSIX.1-2017

This revision of POSIX is technically identical to POSIX.1-2008 with Technical Corrigenda 1 and 2 applied.

SUSv4 2018 edition

This is equivalent to POSIX.1-2017, with the addition of the XCurses specification.

The interfaces documented in POSIX.1/SUS are available as manual pages under sections 0p (header files), 1p (commands), and 3p (functions); thus one can write "man 3p open".

SEE ALSO

getconf (1), *confstr*(3), *pathconf*(3), *sysconf*(3), *attributes*(7), *feature_test_macros*(7), *libc*(7), *posixoptions*(7), *system_data_types*(7)

NAME

stpcpy, strcpy, strcat, stpecpy, strtcpy, strlcpy, strlcat, stpncpy, strncpy, strncat − copying strings and character sequences

SYNOPSIS

Strings

```
// Chain-copy a string.
char *stpcpy(char *restrict dst, const char *restrict src);

// Copy/catenate a string.
char *strcpy(char *restrict dst, const char *restrict src);
char *strcat(char *restrict dst, const char *restrict src);

// Chain-copy a string with truncation.
char *stpecpy(char *dst, char end[0], const char *restrict src);

// Copy/catenate a string with truncation.
ssize_t strtcpy(char dst[restrict .dsize], const char *restrict src,
                size_t dsize);
size_t strlcpy(char dst[restrict .dsize], const char *restrict src,
               size_t dsize);
size_t strlcat(char dst[restrict .dsize], const char *restrict src,
               size_t dsize);
```

Null-padded character sequences

```
// Fill a fixed-size buffer with characters from a string
// and pad with null bytes.
char *strncpy(char dst[restrict .dsize], const char *restrict src,
              size_t dsize);
char *stpncpy(char dst[restrict .dsize], const char *restrict src,
              size_t dsize);

// Chain-copy a null-padded character sequence into a character sequence.
mempcpy(dst, src, strnlen(src, NITEMS(src)));

// Chain-copy a null-padded character sequence into a string.
stpcpy(mempcpy(dst, src, strnlen(src, NITEMS(src))), "");

// Catenate a null-padded character sequence into a string.
char *strncat(char *restrict dst, const char src[restrict .ssize],
              size_t ssize);
```

Known-length character sequences

```
// Chain-copy a known-length character sequence.
void *mempcpy(void dst[restrict .len], const void src[restrict .len],
              size_t len);

// Chain-copy a known-length character sequence into a string.
stpcpy(mempcpy(dst, src, len), "");
```

DESCRIPTION

Terms (and abbreviations)

string (*str*)
: is a sequence of zero or more non-null characters followed by a null character.

character sequence
: is a sequence of zero or more non-null characters. A program should never use a character sequence where a string is required. However, with appropriate care, a string can be used in the place of a character sequence.

  null-padded character sequence
  : Character sequences can be contained in fixed-size buffers, which contain padding null bytes after the character sequence, to fill the rest of the buffer without affecting the character sequence; however, those padding null bytes are not part of the character sequence. Don’t confuse null-padded with null-terminated: null-padded means 0 or more padding null bytes, while null-terminated means exactly 1 terminating null

character.

known-length character sequence
Character sequence delimited by its length. It may be a slice of a larger character sequence, or even of a string.

length (*len*)
is the number of non-null characters in a string or character sequence. It is the return value of *strlen(str)* and of *strnlen(buf, size)*.

size refers to the entire buffer where the string or character sequence is contained.

end is the name of a pointer to one past the last element of a buffer. It is equivalent to *&str[size]*. It is used as a sentinel value, to be able to truncate strings or character sequences instead of overrunning the containing buffer.

copy This term is used when the writing starts at the first element pointed to by *dst*.

catenate
This term is used when a function first finds the terminating null character in *dst*, and then starts writing at that position.

chain This term is used when it's the programmer who provides a pointer to the terminating null character in the string *dst* (or one after the last character in a character sequence), and the function starts writing at that location. The function returns a pointer to the new location of the terminating null character (or one after the last character in a character sequence) after the call, so that the programmer can use it to chain such calls.

Copy, catenate, and chain-copy

Originally, there was a distinction between functions that copy and those that catenate. However, newer functions that copy while allowing chaining cover both use cases with a single API. They are also algorithmically faster, since they don't need to search for the terminating null character of the existing string. However, functions that catenate have a much simpler use, so if performance is not important, it can make sense to use them for improving readability.

The pointer returned by functions that allow chaining is a byproduct of the copy operation, so it has no performance costs. Functions that return such a pointer, and thus can be chained, have names of the form *__stp__*(), since it's common to name the pointer just *p*.

Chain-copying functions that truncate should accept a pointer to the end of the destination buffer, and have names of the form *__stpe__*(). This allows not having to recalculate the remaining size after each call.

Truncate or not?

The first thing to note is that programmers should be careful with buffers, so they always have the correct size, and truncation is not necessary.

In most cases, truncation is not desired, and it is simpler to just do the copy. Simpler code is safer code. Programming against programming mistakes by adding more code just adds more points where mistakes can be made.

Nowadays, compilers can detect most programmer errors with features like compiler warnings, static analyzers, and **_FORTIFY_SOURCE** (see *ftm*(7)). Keeping the code simple helps these overflow-detection features be more precise.

When validating user input, code should normally not truncate, but instead fail and prevent the copy at all.

In some cases, however, it makes sense to truncate.

Functions that truncate:

- **stpecpy**()
- **strtcpy**()
- *strlcpy*(3bsd) and *strlcat*(3bsd) are similar, but have important performance problems; see BUGS.
- *stpncpy*(3) and *strncpy*(3) also truncate, but they don't write strings, but rather null-padded character sequences.

Null-padded character sequences

For historic reasons, some standard APIs and file formats, such as *utmpx*(5) and *tar*(1), use null-padded character sequences in fixed-size buffers. To interface with them, specialized functions need to be used.

To copy bytes from strings into these buffers, use *strncpy*(3) or *stpncpy*(3).

To read a null-padded character sequence, use *strnlen(src, NITEMS(src))*, and then you can treat it as a known-length character sequence; or use *strncat*(3) directly.

Known-length character sequences

The simplest character sequence copying function is *mempcpy*(3). It requires always knowing the length of your character sequences, for which structures can be used. It makes the code much faster, since you always know the length of your character sequences, and can do the minimal copies and length measurements. *mempcpy*(3) copies character sequences, so you need to explicitly set the terminating null character if you need a string.

In programs that make considerable use of strings or character sequences, and need the best performance, using overlapping character sequences can make a big difference. It allows holding subsequences of a larger character sequence, while not duplicating memory nor using time to do a copy.

However, this is delicate, since it requires using character sequences. C library APIs use strings, so programs that use character sequences will have to take care of differentiating strings from character sequences.

To copy a known-length character sequence, use *mempcpy*(3).

To copy a known-length character sequence into a string, use *stpcpy(mempcpy(dst, src, len), "")*.

A string is also accepted as input, because *mempcpy*(3) asks for the length, and a string is composed of a character sequence of the same length plus a terminating null character.

String vs character sequence

Some functions only operate on strings. Those require that the input *src* is a string, and guarantee an output string (even when truncation occurs). Functions that catenate also require that *dst* holds a string before the call. List of functions:

- *stpcpy*(3)
- *strcpy*(3), *strcat*(3)
- **stpecpy**()
- **strtcpy**()
- *strlcpy*(3bsd), *strlcat*(3bsd)

Other functions require an input string, but create a character sequence as output. These functions have confusing names, and have a long history of misuse. List of functions:

- *stpncpy*(3)
- *strncpy*(3)

Other functions operate on an input character sequence, and create an output string. Functions that catenate also require that *dst* holds a string before the call. *strncat*(3) has an even more misleading name than the functions above. List of functions:

- *strncat*(3)

Other functions operate on an input character sequence to create an output character sequence. List of functions:

- *mempcpy*(3)

Functions

stpcpy(3)

Copy the input string into a destination string. The programmer is responsible for allocating a buffer large enough. It returns a pointer suitable for chaining.

strcpy(3)
strcat(3)

Copy and catenate the input string into a destination string. The programmer is responsible for allocating a buffer large enough. The return value is useless.

stpcpy(3) is a faster alternative to these functions.

stpecpy()
: Chain-copy the input string into a destination string. If the destination buffer, limited by a pointer to its end, isn't large enough to hold the copy, the resulting string is truncated (but it is guaranteed to be null-terminated). It returns a pointer suitable for chaining. Truncation needs to be detected only once after the last chained call.

  This function is not provided by any library; see EXAMPLES for a reference implementation.

strtcpy()
: Copy the input string into a destination string. If the destination buffer isn't large enough to hold the copy, the resulting string is truncated (but it is guaranteed to be null-terminated). It returns the length of the string, or −1 if it truncated.

  This function is not provided by any library; see EXAMPLES for a reference implementation.

strlcpy(3bsd)
strlcat(3bsd)
: Copy and catenate the input string into a destination string. If the destination buffer, limited by its size, isn't large enough to hold the copy, the resulting string is truncated (but it is guaranteed to be null-terminated). They return the length of the total string they tried to create.

  Check BUGS before using these functions.

  strtcpy() and **stpecpy**() are better alternatives to these functions.

stpncpy(3)
: Copy the input string into a destination null-padded character sequence in a fixed-size buffer. If the destination buffer, limited by its size, isn't large enough to hold the copy, the resulting character sequence is truncated. Since it creates a character sequence, it doesn't need to write a terminating null character. It's impossible to distinguish truncation by the result of the call, from a character sequence that just fits the destination buffer; truncation should be detected by comparing the length of the input string with the size of the destination buffer.

strncpy(3)
: This function is identical to *stpncpy*(3) except for the useless return value.

  stpncpy(3) is a more useful alternative to this function.

strncat(3)
: Catenate the input character sequence, contained in a null-padded fixed-size buffer, into a destination string. The programmer is responsible for allocating a buffer large enough. The return value is useless.

  Do not confuse this function with *strncpy*(3); they are not related at all.

  stpcpy(mempcpy(dst, src, strnlen(src, NITEMS(src))), "") is a faster alternative to this function.

mempcpy(3)
: Copy the input character sequence, limited by its length, into a destination character sequence. The programmer is responsible for allocating a buffer large enough. It returns a pointer suitable for chaining.

RETURN VALUE

stpcpy(3)
: A pointer to the terminating null character in the destination string.

stpecpy()
: A pointer to the terminating null character in the destination string, on success. On error, NULL is returned, and *errno* is set to indicate the error.

mempcpy(3)
stpncpy(3)
: A pointer to one after the last character in the destination character sequence.

strtcpy()
: The length of the string, on success. On error, −1 is returned, and *errno* is set to indicate the error.

strlcpy(3bsd)
strlcat(3bsd)
: The length of the total string that they tried to create (as if truncation didn't occur).

strcpy(3)
strcat(3)
strncpy(3)
strncat(3)
: The *dst* pointer, which is useless.

ERRORS

Most of these functions don't set *errno*.

stpecpy()
strtcpy()

ENOBUFS
: *dsize* was **0**.

E2BIG The string has been truncated.

NOTES

The Linux kernel has an internal function for copying strings, *strscpy*(9), which is identical to **strtcpy**(), except that it returns **−E2BIG** instead of −1 and it doesn't set *errno*.

CAVEATS

Don't mix chain calls to truncating and non-truncating functions. It is conceptually wrong unless you know that the first part of a copy will always fit. Anyway, the performance difference will probably be negligible, so it will probably be more clear if you use consistent semantics: either truncating or non-truncating. Calling a non-truncating function after a truncating one is necessarily wrong.

BUGS

All catenation functions share the same performance problem: Shlemiel the painter. As a mitigation, compilers are able to transform some calls to catenation functions into normal copy functions, since *strlen(dst)* is usually a byproduct of the previous copy.

strlcpy(3) and *strlcat*(3) need to read the entire *src* string, even if the destination buffer is small. This makes them vulnerable to Denial of Service (DoS) attacks if an attacker can control the length of the *src* string. And if not, they're still unnecessarily slow.

EXAMPLES

The following are examples of correct use of each of these functions.

stpcpy(3)

```
p = buf;
p = stpcpy(p, "Hello ");
p = stpcpy(p, "world");
p = stpcpy(p, "!");
len = p - buf;
puts(buf);
```

strcpy(3)
strcat(3)

```
strcpy(buf, "Hello ");
strcat(buf, "world");
strcat(buf, "!");
len = strlen(buf);
puts(buf);
```

stpecpy()

```
end = buf + NITEMS(buf);
p = buf;
p = stpecpy(p, end, "Hello ");
p = stpecpy(p, end, "world");
p = stpecpy(p, end, "!");
if (p == NULL) {
```

```
            len = NITEMS(buf) - 1;
            goto toolong;
        }
        len = p - buf;
        puts(buf);
```

strtcpy()

```
        len = strtcpy(buf, "Hello world!", NITEMS(buf));
        if (len == -1)
            goto toolong;
        puts(buf);
```

strlcpy(3bsd)
strlcat(3bsd)

```
        if (strlcpy(buf, "Hello ", NITEMS(buf)) >= NITEMS(buf))
            goto toolong;
        if (strlcat(buf, "world", NITEMS(buf)) >= NITEMS(buf))
            goto toolong;
        len = strlcat(buf, "!", NITEMS(buf));
        if (len >= NITEMS(buf))
            goto toolong;
        puts(buf);
```

stpncpy(3)

```
        p = stpncpy(u->ut_user, "alx", NITEMS(u->ut_user));
        if (NITEMS(u->ut_user) < strlen("alx"))
            goto toolong;
        len = p - u->ut_user;
        fwrite(u->ut_user, 1, len, stdout);
```

strncpy(3)

```
        strncpy(u->ut_user, "alx", NITEMS(u->ut_user));
        if (NITEMS(u->ut_user) < strlen("alx"))
            goto toolong;
        len = strnlen(u->ut_user, NITEMS(u->ut_user));
        fwrite(u->ut_user, 1, len, stdout);
```

mempcpy(dst, src, strnlen(src, NITEMS(src)))

```
        char  buf[NITEMS(u->ut_user)];
        p = buf;
        p = mempcpy(p, u->ut_user, strnlen(u->ut_user, NITEMS(u->ut_user)));
        len = p - buf;
        fwrite(buf, 1, len, stdout);
```

stpcpy(mempcpy(dst, src, strnlen(src, NITEMS(src))), "")

```
        char  buf[NITEMS(u->ut_user) + 1];
        p = buf;
        p = mempcpy(p, u->ut_user, strnlen(u->ut_user, NITEMS(u->ut_user)));
        p = stpcpy(p, "");
        len = p - buf;
        puts(buf);
```

strncat(3)

```
        char  buf[NITEMS(u->ut_user) + 1];
        strcpy(buf, "");
        strncat(buf, u->ut_user, NITEMS(u->ut_user));
        len = strlen(buf);
        puts(buf);
```

mempcpy(3)

```
        p = buf;
        p = mempcpy(p, "Hello ", 6);
        p = mempcpy(p, "world", 5);
```

```
    p = mempcpy(p, "!", 1);
    len = p - buf;
    fwrite(buf, 1, len, stdout);
```

stpcpy(mempcpy(dst, src, len), "")

```
    p = buf;
    p = mempcpy(p, "Hello ", 6);
    p = mempcpy(p, "world", 5);
    p = mempcpy(p, "!", 1);
    p = stpcpy(p, "");
    len = p - buf;
    puts(buf);
```

Implementations

Here are reference implementations for functions not provided by libc.

```
/* This code is in the public domain. */

char *
stpecpy(char *dst, char end[0], const char *restrict src)
{
    size_t  dlen;

    if (dst == NULL)
        return NULL;

    dlen = strtcpy(dst, src, end - dst);
    return (dlen == -1) ? NULL : dst + dlen;
}

ssize_t
strtcpy(char *restrict dst, const char *restrict src, size_t dsize)
{
    bool    trunc;
    size_t  dlen, slen;

    if (dsize == 0) {
        errno = ENOBUFS;
        return -1;
    }

    slen = strnlen(src, dsize);
    trunc = (slen == dsize);
    dlen = slen - trunc;

    stpcpy(mempcpy(dst, src, dlen), "");
    if (trunc)
        errno = E2BIG;
    return trunc ? -1 : slen;
}
```

SEE ALSO

bzero(3), *memcpy*(3), *memccpy*(3), *mempcpy*(3), *stpcpy*(3), *strlcpy*(3bsd), *strncat*(3), *stpncpy*(3), *string*(3)

NAME

suffixes − list of file suffixes

DESCRIPTION

It is customary to indicate the contents of a file with the file suffix, which (typically) consists of a period, followed by one or more letters. Many standard utilities, such as compilers, use this to recognize the type of file they are dealing with. The *make*(1) utility is driven by rules based on file suffix.

Following is a list of suffixes which are likely to be found on a Linux system.

Suffix	File type
,v	files for RCS (Revision Control System)
-	backup file
.C	C++ source code, equivalent to *.cc*
.F	Fortran source with*cpp*(1) directives or file compressed using freeze
.S	assembler source with*cpp*(1) directives
.Y	file compressed using yabba
.Z	file compressed using*compress*(1)
.[0−9]+gf	TeX generic font files
.[0−9]+pk	TeX packed font files
.[1−9]	manual page for the corresponding section
.[1−9][a-z]	manual page for section plus subsection
.a	static object code library
.ad	X application default resource file
.ada	Ada source (may be body, spec, or combination)
.adb	Ada body source
.ads	Ada spec source
.afm	PostScript font metrics
.al	Perl autoload file
.am	*automake*(1) input file
.arc	*arc*(1) archive
.arj	*arj*(1) archive
.asc	PGP ASCII-armored data
.asm	(GNU) assembler source file
.au	Audio sound file
.aux	LaTeX auxiliary file
.avi	(msvideo) movie
.awk	AWK language program
.b	LILO boot loader image
.bak	backup file
.bash	*bash*(1) shell script
.bb	basic block list data produced by gcc −ftest−coverage
.bbg	basic block graph data produced by gcc −ftest−coverage
.bbl	BibTeX output
.bdf	X font file
.bib	TeX bibliographic database, BibTeX input
.bm	bitmap source
.bmp	bitmap
.bz2	file compressed using*bzip2*(1)
.c	C source
.cat	message catalog files
.cc	C++ source
.cf	configuration file
.cfg	configuration file
.cgi	WWW content generating script or program
.cls	LaTeX Class definition

.class	Java compiled byte-code
.conf	configuration file
.config	configuration file
.cpp	equivalent to *.cc*
.csh	*csh*(1) shell script
.cxx	equivalent to *.cc*
.dat	data file
.deb	Debian software package
.def	Modula-2 source for definition modules
.def	other definition files
.desc	initial part of mail message unpacked with *munpack*(1)
.diff	file differences (*diff*(1) command output)
.dir	dbm data base directory file
.doc	documentation file
.dsc	Debian Source Control (source package)
.dtx	LaTeX package source file
.dvi	TeX's device independent output
.el	Emacs-Lisp source
.elc	compiled Emacs-Lisp source
.eps	encapsulated PostScript
.exp	Expect source code
.f	Fortran source
.f77	Fortran 77 source
.f90	Fortran 90 source
.fas	precompiled Common-Lisp
.fi	Fortran include files
.fig	FIG image file (used by*xfig*(1))
.fmt	TeX format file
.gif	Compuserve Graphics Image File format
.gmo	GNU format message catalog
.gsf	Ghostscript fonts
.gz	file compressed using*gzip*(1)
.h	C or C++ header files
.help	help file
.hf	equivalent to *.help*
.hlp	equivalent to *.help*
.htm	poor man's *.html*
.html	HTML document used with the World Wide Web
.hqx	7-bit encoded Macintosh file
.i	C source after preprocessing
.icon	bitmap source
.idx	reference or datum-index file for hypertext or database system
.image	bitmap source
.in	configuration template, especially for GNU Autoconf
.info	files for the Emacs info browser
.info-[0−9]+	split info files
.ins	LaTeX package install file for docstrip
.itcl	itcl source code; itcl ([incr Tcl]) is an OO extension of tcl
.java	a Java source file
.jpeg	Joint Photographic Experts Group format
.jpg	poor man's *.jpeg*
.js	JavaScript source code
.jsx	JSX (JavaScript XML-like extension) source code
.kmap	*lyx*(1) keymap
.l	equivalent to *.lex* or *.lisp*

.lex	*lex*(1) or*flex*(1) files
.lha	lharc archive
.lib	Common-Lisp library
.lisp	Lisp source
.ln	files for use with*lint*(1)
.log	log file, in particular produced by TeX
.lsm	Linux Software Map entry
.lsp	Common-Lisp source
.lzh	lharc archive
.m	Objective-C source code
.m4	*m4*(1) source
.mac	macro files for various programs
.man	manual page (usually source rather than formatted)
.map	map files for various programs
.me	Nroff source using the me macro package
.mf	Metafont (font generator for TeX) source
.mgp	MagicPoint file
.mm	sources for*groff*(1) in mm - format
.mo	Message catalog binary file
.mod	Modula-2 source for implementation modules
.mov	(quicktime) movie
.mp	Metapost source
.mp2	MPEG Layer 2 (audio) file
.mp3	MPEG Layer 3 (audio) file
.mpeg	movie file
.o	object file
.old	old or backup file
.orig	backup (original) version of a file, from*patch*(1)
.out	output file, often executable program (a.out)
.p	Pascal source
.pag	dbm data base data file
.patch	file differences for*patch*(1)
.pbm	portable bitmap format
.pcf	X11 font files
.pdf	Adobe Portable Data Format (use Acrobat/**acroread** or **xpdf**)
.perl	Perl source (see .ph, .pl, and .pm)
.pfa	PostScript font definition files, ASCII format
.pfb	PostScript font definition files, binary format
.pgm	portable greymap format
.pgp	PGP binary data
.ph	Perl header file
.php	PHP program file
.php3	PHP3 program file
.pid	File to store daemon PID (e.g., crond.pid)
.pl	TeX property list file or Perl library file
.pm	Perl module
.png	Portable Network Graphics file
.po	Message catalog source
.pod	*perldoc*(1) file
.ppm	portable pixmap format
.pr	bitmap source
.ps	PostScript file
.py	Python source
.pyc	compiled python
.qt	quicktime movie
.r	RATFOR source (obsolete)
.rej	patches that*patch*(1) couldn’t apply

.rpm	RPM software package
.rtf	Rich Text Format file
.rules	rules for something
.s	assembler source
.sa	stub libraries for a.out shared libraries
.sc	*sc*(1) spreadsheet commands
.scm	Scheme source code
.sed	sed source file
.sgml	SGML source file
.sh	*sh*(1) scripts
.shar	archive created by the*shar*(1) utility
.shtml	HTML using Server Side Includes
.so	Shared library or dynamically loadable object
.sql	SQL source
.sqml	SQML schema or query program
.sty	LaTeX style files
.sym	Modula-2 compiled definition modules
.tar	archive created by the*tar*(1) utility
.tar.Z	tar(1) archive compressed with*compress*(1)
.tar.bz2	tar(1) archive compressed with*bzip2*(1)
.tar.gz	tar(1) archive compressed with*gzip*(1)
.taz	tar(1) archive compressed with*compress*(1)
.tcl	tcl source code
.tex	TeX or LaTeX source
.texi	equivalent to *.texinfo*
.texinfo	Texinfo documentation source
.text	text file
.tfm	TeX font metric file
.tgz	tar archive compressed with*gzip*(1)
.tif	poor man's *.tiff*
.tiff	Tagged Image File Format
.tk	tcl/tk script
.tmp	temporary file
.tmpl	template files
.ts	TypeScript source code
.tsx	TypeScript with JSX source code (*.ts* + *.jsx*)
.txt	equivalent to *.text*
.uu	equivalent to *.uue*
.uue	binary file encoded with*uuencode*(1)
.vf	TeX virtual font file
.vpl	TeX virtual property list file
.w	Silvio Levi's CWEB
.wav	wave sound file
.web	Donald Knuth's WEB
.wml	Source file for Web Meta Language
.xbm	X11 bitmap source
.xcf	GIMP graphic
.xml	eXtended Markup Language file
.xpm	X11 pixmap source
.xs	Perl xsub file produced by h2xs
.xsl	XSL stylesheet
.y	*yacc*(1) or*bison*(1) (parser generator) files
.z	File compressed using*pack*(1) (or an old*gzip*(1))
.zip	*zip*(1) archive
.zoo	*zoo*(1) archive
~	Emacs or*patch*(1) backup file
rc	startup ('run control') file, e.g., *.newsrc*

STANDARDS

General UNIX conventions.

BUGS

This list is not exhaustive.

SEE ALSO

file(1), *make*(1)

NAME

symlink − symbolic link handling

DESCRIPTION

Symbolic links are files that act as pointers to other files. To understand their behavior, you must first understand how hard links work.

A hard link to a file is indistinguishable from the original file because it is a reference to the object underlying the original filename. (To be precise: each of the hard links to a file is a reference to the same *inode number*, where an inode number is an index into the inode table, which contains metadata about all files on a filesystem. See *stat*(2).) Changes to a file are independent of the name used to reference the file. Hard links may not refer to directories (to prevent the possibility of loops within the filesystem tree, which would confuse many programs) and may not refer to files on different filesystems (because inode numbers are not unique across filesystems).

A symbolic link is a special type of file whose contents are a string that is the pathname of another file, the file to which the link refers. (The contents of a symbolic link can be read using *readlink*(2).) In other words, a symbolic link is a pointer to another name, and not to an underlying object. For this reason, symbolic links may refer to directories and may cross filesystem boundaries.

There is no requirement that the pathname referred to by a symbolic link should exist. A symbolic link that refers to a pathname that does not exist is said to be a *dangling link*.

Because a symbolic link and its referenced object coexist in the filesystem name space, confusion can arise in distinguishing between the link itself and the referenced object. On historical systems, commands and system calls adopted their own link-following conventions in a somewhat ad-hoc fashion. Rules for a more uniform approach, as they are implemented on Linux and other systems, are outlined here. It is important that site-local applications also conform to these rules, so that the user interface can be as consistent as possible.

Magic links

There is a special class of symbolic-link-like objects known as "magic links", which can be found in certain pseudofilesystems such as *proc*(5) (examples include */proc/*pid*/exe* and */proc/*pid*/fd/**). Unlike normal symbolic links, magic links are not resolved through pathname-expansion, but instead act as direct references to the kernel's own representation of a file handle. As such, these magic links allow users to access files which cannot be referenced with normal paths (such as unlinked files still referenced by a running program).

Because they can bypass ordinary *mount_namespaces*(7)-based restrictions, magic links have been used as attack vectors in various exploits.

Symbolic link ownership, permissions, and timestamps

The owner and group of an existing symbolic link can be changed using *lchown*(2). The ownership of a symbolic link matters when the link is being removed or renamed in a directory that has the sticky bit set (see *inode*(7)), and when the *fs.protected_symlinks* sysctl is set (see *proc*(5)).

The last access and last modification timestamps of a symbolic link can be changed using *utimensat*(2) or *lutimes*(3).

On Linux, the permissions of an ordinary symbolic link are not used in any operations; the permissions are always 0777 (read, write, and execute for all user categories), and can't be changed.

However, magic links do not follow this rule. They can have a non-0777 mode, though this mode is not currently used in any permission checks.

Obtaining a file descriptor that refers to a symbolic link

Using the combination of the **O_PATH** and **O_NOFOLLOW** flags to *open*(2) yields a file descriptor that can be passed as the *dirfd* argument in system calls such as *fstatat*(2), *fchownat*(2), *fchmodat*(2), *linkat*(2), and *readlinkat*(2), in order to operate on the symbolic link itself (rather than the file to which it refers).

By default (i.e., if the **AT_SYMLINK_FOLLOW** flag is not specified), if *name_to_handle_at*(2) is applied to a symbolic link, it yields a handle for the symbolic link (rather than the file to which it refers). One can then obtain a file descriptor for the symbolic link (rather than the file to which it refers) by specifying the **O_PATH** flag in a subsequent call to *open_by_handle_at*(2). Again, that file descriptor can be used in the aforementioned system calls to operate on the symbolic link itself.

Handling of symbolic links by system calls and commands

Symbolic links are handled either by operating on the link itself, or by operating on the object referred to by the link. In the latter case, an application or system call is said to *follow* the link. Symbolic links may refer to other symbolic links, in which case the links are dereferenced until an object that is not a symbolic link is found, a symbolic link that refers to a file which does not exist is found, or a loop is detected. (Loop detection is done by placing an upper limit on the number of links that may be followed, and an error results if this limit is exceeded.)

There are three separate areas that need to be discussed. They are as follows:

- Symbolic links used as filename arguments for system calls.
- Symbolic links specified as command-line arguments to utilities that are not traversing a file tree.
- Symbolic links encountered by utilities that are traversing a file tree (either specified on the command line or encountered as part of the file hierarchy walk).

Before describing the treatment of symbolic links by system calls and commands, we require some terminology. Given a pathname of the form *a/b/c*, the part preceding the final slash (i.e., *a/b*) is called the *dirname* component, and the part following the final slash (i.e., *c*) is called the *basename* component.

Treatment of symbolic links in system calls

The first area is symbolic links used as filename arguments for system calls.

The treatment of symbolic links within a pathname passed to a system call is as follows:

(1) Within the dirname component of a pathname, symbolic links are always followed in nearly every system call. (This is also true for commands.) The one exception is *openat2*(2), which provides flags that can be used to explicitly prevent following of symbolic links in the dirname component.

(2) Except as noted below, all system calls follow symbolic links in the basename component of a pathname. For example, if there were a symbolic link *slink* which pointed to a file named *afile*, the system call *open("slink" ...)* would return a file descriptor referring to the file *afile*.

Various system calls do not follow links in the basename component of a pathname, and operate on the symbolic link itself. They are: *lchown*(2), *lgetxattr*(2), *llistxattr*(2), *lremovexattr*(2), *lsetxattr*(2), *lstat*(2), *readlink*(2), *rename*(2), *rmdir*(2), and *unlink*(2).

Certain other system calls optionally follow symbolic links in the basename component of a pathname. They are: *faccessat*(2), *fchownat*(2), *fstatat*(2), *linkat*(2), *name_to_handle_at*(2), *open*(2), *openat*(2), *open_by_handle_at*(2), and *utimensat*(2); see their manual pages for details. Because *remove*(3) is an alias for *unlink*(2), that library function also does not follow symbolic links. When *rmdir*(2) is applied to a symbolic link, it fails with the error **ENOTDIR**.

link(2) warrants special discussion. POSIX.1-2001 specifies that *link*(2) should dereference *oldpath* if it is a symbolic link. However, Linux does not do this. (By default, Solaris is the same, but the POSIX.1-2001 specified behavior can be obtained with suitable compiler options.) POSIX.1-2008 changed the specification to allow either behavior in an implementation.

Commands not traversing a file tree

The second area is symbolic links, specified as command-line filename arguments, to commands which are not traversing a file tree.

Except as noted below, commands follow symbolic links named as command-line arguments. For example, if there were a symbolic link *slink* which pointed to a file named *afile*, the command *cat slink* would display the contents of the file *afile*.

It is important to realize that this rule includes commands which may optionally traverse file trees; for example, the command *chown file* is included in this rule, while the command *chown −R file*, which performs a tree traversal, is not. (The latter is described in the third area, below.)

If it is explicitly intended that the command operate on the symbolic link instead of following the symbolic link—for example, it is desired that *chown slink* change the ownership of the file that *slink* is, whether it is a symbolic link or not—then the *−h* option should be used. In the above example, *chown root slink* would change the ownership of the file referred to by *slink*, while *chown −h root slink* would change the ownership of *slink* itself.

There are some exceptions to this rule:

- The *mv*(1) and *rm*(1) commands do not follow symbolic links named as arguments, but respectively attempt to rename and delete them. (Note, if the symbolic link references a file via a relative path, moving it to another directory may very well cause it to stop working, since the path may no longer be correct.)
- The *ls*(1) command is also an exception to this rule. For compatibility with historic systems (when *ls*(1) is not doing a tree walk—that is, *−R* option is not specified), the *ls*(1) command follows symbolic links named as arguments if the *−H* or *−L* option is specified, or if the *−F*, *−d*, or *−l* options are not specified. (The *ls*(1) command is the only command where the *−H* and *−L* options affect its behavior even though it is not doing a walk of a file tree.)
- The *file*(1) command is also an exception to this rule. The *file*(1) command does not follow symbolic links named as argument by default. The *file*(1) command does follow symbolic links named as argument if the *−L* option is specified.

Commands traversing a file tree

The following commands either optionally or always traverse file trees: *chgrp*(1), *chmod*(1), *chown*(1), *cp*(1), *du*(1), *find*(1), *ls*(1), *pax*(1), *rm*(1), and *tar*(1)

It is important to realize that the following rules apply equally to symbolic links encountered during the file tree traversal and symbolic links listed as command-line arguments.

The *first rule* applies to symbolic links that reference files other than directories. Operations that apply to symbolic links are performed on the links themselves, but otherwise the links are ignored.

The command *rm −r slink directory* will remove *slink*, as well as any symbolic links encountered in the tree traversal of *directory*, because symbolic links may be removed. In no case will *rm*(1) affect the file referred to by *slink*.

The *second rule* applies to symbolic links that refer to directories. Symbolic links that refer to directories are never followed by default. This is often referred to as a "physical" walk, as opposed to a "logical" walk (where symbolic links that refer to directories are followed).

Certain conventions are (should be) followed as consistently as possible by commands that perform file tree walks:

- A command can be made to follow any symbolic links named on the command line, regardless of the type of file they reference, by specifying the *−H* (for "half-logical") flag. This flag is intended to make the command-line name space look like the logical name space. (Note, for commands that do not always do file tree traversals, the *−H* flag will be ignored if the *−R* flag is not also specified.)

  For example, the command *chown −HR user slink* will traverse the file hierarchy rooted in the file pointed to by *slink*. Note, the *−H* is not the same as the previously discussed *−h* flag. The *−H* flag causes symbolic links specified on the command line to be dereferenced for the purposes of both the action to be performed and the tree walk, and it is as if the user had specified the name of the file to which the symbolic link pointed.
- A command can be made to follow any symbolic links named on the command line, as well as any symbolic links encountered during the traversal, regardless of the type of file they reference, by specifying the *−L* (for "logical") flag. This flag is intended to make the entire name space look like the logical name space. (Note, for commands that do not always do file tree traversals, the *−L* flag will be ignored if the *−R* flag is not also specified.)

  For example, the command *chown −LR user slink* will change the owner of the file referred to by *slink*. If *slink* refers to a directory, **chown** will traverse the file hierarchy rooted in the directory that it references. In addition, if any symbolic links are encountered in any file tree that **chown** traverses, they will be treated in the same fashion as *slink*.
- A command can be made to provide the default behavior by specifying the *−P* (for "physical") flag. This flag is intended to make the entire name space look like the physical name space.

For commands that do not by default do file tree traversals, the *−H*, *−L*, and *−P* flags are ignored if the *−R* flag is not also specified. In addition, you may specify the *−H*, *−L*, and *−P* options more than once; the last one specified determines the command's behavior. This is intended to permit you to alias commands to behave one way or the other, and then override that behavior on the command line.

The *ls*(1) and *rm*(1) commands have exceptions to these rules:

- The *rm*(1) command operates on the symbolic link, and not the file it references, and therefore never follows a symbolic link. The *rm*(1) command does not support the *−H*, *−L*, or *−P* options.
- To maintain compatibility with historic systems, the *ls*(1) command acts a little differently. If you do not specify the *−F*, *−d*, or *−l* options, *ls*(1) will follow symbolic links specified on the command line. If the *−L* flag is specified, *ls*(1) follows all symbolic links, regardless of their type, whether specified on the command line or encountered in the tree walk.

SEE ALSO

chgrp(1), *chmod*(1), *find*(1), *ln*(1), *ls*(1), *mv*(1), *namei*(1), *rm*(1), *lchown*(2), *link*(2), *lstat*(2), *readlink*(2), *rename*(2), *symlink*(2), *unlink*(2), *utimensat*(2), *lutimes*(3), *path_resolution*(7)

NAME

system_data_types − overview of system data types

DESCRIPTION

siginfo_t

Include: <*signal.h*>. Alternatively, <*sys/wait.h*>.

```
typedef struct {
    int      si_signo;  /* Signal number */
    int      si_code;   /* Signal code */
    pid_t    si_pid;    /* Sending process ID */
    uid_t    si_uid;    /* Real user ID of sending process */
    void    *si_addr;   /* Memory location which caused fault */
    int      si_status; /* Exit value or signal */
    union sigval si_value;  /* Signal value */
} siginfo_t;
```

Information associated with a signal. For further details on this structure (including additional, Linux-specific fields), see *sigaction*(2).

Conforming to: POSIX.1-2001 and later.

See also: *pidfd_send_signal*(2), *rt_sigqueueinfo*(2), *sigaction*(2), *sigwaitinfo*(2), *psiginfo*(3)

sigset_t

Include: <*signal.h*>. Alternatively, <*spawn.h*>, or <*sys/select.h*>.

This is a type that represents a set of signals. According to POSIX, this shall be an integer or structure type.

Conforming to: POSIX.1-2001 and later.

See also: *epoll_pwait*(2), *ppoll*(2), *pselect*(2), *sigaction*(2), *signalfd*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigwaitinfo*(2), *signal*(7)

NOTES

The structures described in this manual page shall contain, at least, the members shown in their definition, in no particular order.

Most of the integer types described in this page don't have a corresponding length modifier for the *printf*(3) and the *scanf*(3) families of functions. To print a value of an integer type that doesn't have a length modifier, it should be converted to *intmax_t* or *uintmax_t* by an explicit cast. To scan into a variable of an integer type that doesn't have a length modifier, an intermediate temporary variable of type *intmax_t* or *uintmax_t* should be used. When copying from the temporary variable to the destination variable, the value could overflow. If the type has upper and lower limits, the user should check that the value is within those limits, before actually copying the value. The example below shows how these conversions should be done.

Conventions used in this page

In "Conforming to" we only concern ourselves with C99 and later and POSIX.1-2001 and later. Some types may be specified in earlier versions of one of these standards, but in the interests of simplicity we omit details from earlier standards.

In "Include", we first note the "primary" header(s) that define the type according to either the C or POSIX.1 standards. Under "Alternatively", we note additional headers that the standards specify shall define the type.

EXAMPLES

The program shown below scans from a string and prints a value stored in a variable of an integer type that doesn't have a length modifier. The appropriate conversions from and to *intmax_t*, and the appropriate range checks, are used as explained in the notes section above.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int
```

```
main (void)
{
    static const char *const str = "500000 us in half a second";
    suseconds_t us;
    intmax_t    tmp;

    /* Scan the number from the string into the temporary variable. */

    sscanf(str, "%jd", &tmp);

    /* Check that the value is within the valid range of suseconds_t. */

    if (tmp < −1 || tmp > 1000000) {
        fprintf(stderr, "Scanned value outside valid range!\n");
        exit(EXIT_FAILURE);
    }

    /* Copy the value to the suseconds_t variable 'us'. */

    us = tmp;

    /* Even though suseconds_t can hold the value −1, this isn't
       a sensible number of microseconds. */

    if (us < 0) {
        fprintf(stderr, "Scanned value shouldn't be negative!\n");
        exit(EXIT_FAILURE);
    }

    /* Print the value. */

    printf("There are %jd microseconds in half a second.\n",
            (intmax_t) us);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

feature_test_macros(7), *standards*(7)

NAME

sysvipc − System V interprocess communication mechanisms

DESCRIPTION

System V IPC is the name given to three interprocess communication mechanisms that are widely available on UNIX systems: message queues, semaphore, and shared memory.

Message queues

System V message queues allow data to be exchanged in units called messages. Each message can have an associated priority. POSIX message queues provide an alternative API for achieving the same result; see *mq_overview*(7).

The System V message queue API consists of the following system calls:

msgget(2)
: Create a new message queue or obtain the ID of an existing message queue. This call returns an identifier that is used in the remaining APIs.

msgsnd(2)
: Add a message to a queue.

msgrcv(2)
: Remove a message from a queue.

msgctl(2)
: Perform various control operations on a queue, including deletion.

Semaphore sets

System V semaphores allow processes to synchronize their actions. System V semaphores are allocated in groups called sets; each semaphore in a set is a counting semaphore. POSIX semaphores provide an alternative API for achieving the same result; see *sem_overview*(7).

The System V semaphore API consists of the following system calls:

semget(2)
: Create a new set or obtain the ID of an existing set. This call returns an identifier that is used in the remaining APIs.

semop(2)
: Perform operations on the semaphores in a set.

semctl(2)
: Perform various control operations on a set, including deletion.

Shared memory segments

System V shared memory allows processes to share a region a memory (a "segment"). POSIX shared memory is an alternative API for achieving the same result; see *shm_overview*(7).

The System V shared memory API consists of the following system calls:

shmget(2)
: Create a new segment or obtain the ID of an existing segment. This call returns an identifier that is used in the remaining APIs.

shmat(2)
: Attach an existing shared memory object into the calling process’s address space.

shmdt(2)
: Detach a segment from the calling process’s address space.

shmctl(2)
: Perform various control operations on a segment, including deletion.

IPC namespaces

For a discussion of the interaction of System V IPC objects and IPC namespaces, see *ipc_namespaces*(7).

SEE ALSO

ipcmk(1), *ipcrm*(1), *ipcs*(1), *lsipc*(1), *ipc*(2), *msgctl*(2), *msgget*(2), *msgrcv*(2), *msgsnd*(2), *semctl*(2), *semget*(2), *semop*(2), *shmat*(2), *shmctl*(2), *shmdt*(2), *shmget*(2), *ftok*(3), *ipc_namespaces*(7)

NAME

tcp − TCP protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

tcp_socket **= socket(AF_INET, SOCK_STREAM, 0);**

DESCRIPTION

This is an implementation of the TCP protocol defined in RFC 793, RFC 1122 and RFC 2001 with the NewReno and SACK extensions. It provides a reliable, stream-oriented, full-duplex connection between two sockets on top of *ip*(7), for both v4 and v6 versions. TCP guarantees that the data arrives in order and retransmits lost packets. It generates and checks a per-packet checksum to catch transmission errors. TCP does not preserve record boundaries.

A newly created TCP socket has no remote or local address and is not fully specified. To create an outgoing TCP connection use *connect*(2) to establish a connection to another TCP socket. To receive new incoming connections, first *bind*(2) the socket to a local address and port and then call *listen*(2) to put the socket into the listening state. After that a new socket for each incoming connection can be accepted using *accept*(2). A socket which has had *accept*(2) or *connect*(2) successfully called on it is fully specified and may transmit data. Data cannot be transmitted on listening or not yet connected sockets.

Linux supports RFC 1323 TCP high performance extensions. These include Protection Against Wrapped Sequence Numbers (PAWS), Window Scaling and Timestamps. Window scaling allows the use of large (> 64 kB) TCP windows in order to support links with high latency or bandwidth. To make use of them, the send and receive buffer sizes must be increased. They can be set globally with the */proc/sys/net/ipv4/tcp_wmem* and */proc/sys/net/ipv4/tcp_rmem* files, or on individual sockets by using the **SO_SNDBUF** and **SO_RCVBUF** socket options with the *setsockopt*(2) call.

The maximum sizes for socket buffers declared via the **SO_SNDBUF** and **SO_RCVBUF** mechanisms are limited by the values in the */proc/sys/net/core/rmem_max* and */proc/sys/net/core/wmem_max* files. Note that TCP actually allocates twice the size of the buffer requested in the *setsockopt*(2) call, and so a succeeding *getsockopt*(2) call will not return the same size of buffer as requested in the *setsockopt*(2) call. TCP uses the extra space for administrative purposes and internal kernel structures, and the */proc* file values reflect the larger sizes compared to the actual TCP windows. On individual connections, the socket buffer size must be set prior to the *listen*(2) or *connect*(2) calls in order to have it take effect. See *socket*(7) for more information.

TCP supports urgent data. Urgent data is used to signal the receiver that some important message is part of the data stream and that it should be processed as soon as possible. To send urgent data specify the **MSG_OOB** option to *send*(2). When urgent data is received, the kernel sends a **SIGURG** signal to the process or process group that has been set as the socket "owner" using the **SIOCSPGRP** or **FIOSETOWN** ioctls (or the POSIX.1-specified *fcntl*(2) **F_SETOWN** operation). When the **SO_OOBINLINE** socket option is enabled, urgent data is put into the normal data stream (a program can test for its location using the **SIOCATMARK** ioctl described below), otherwise it can be received only when the **MSG_OOB** flag is set for *recv*(2) or *recvmsg*(2).

When out-of-band data is present, *select*(2) indicates the file descriptor as having an exceptional condition and *poll (2)* indicates a **POLLPRI** event.

Linux 2.4 introduced a number of changes for improved throughput and scaling, as well as enhanced functionality. Some of these features include support for zero-copy *sendfile*(2), Explicit Congestion Notification, new management of TIME_WAIT sockets, keep-alive socket options and support for Duplicate SACK extensions.

Address formats

TCP is built on top of IP (see *ip*(7)). The address formats defined by *ip*(7) apply to TCP. TCP supports point-to-point communication only; broadcasting and multicasting are not supported.

/proc interfaces

System-wide TCP parameter settings can be accessed by files in the directory */proc/sys/net/ipv4/*. In addition, most IP */proc* interfaces also apply to TCP; see *ip*(7). Variables described as *Boolean* take an

integer value, with a nonzero value ("true") meaning that the corresponding option is enabled, and a zero value ("false") meaning that the option is disabled.

tcp_abc (Integer; default: 0; Linux 2.6.15 to Linux 3.8)
Control the Appropriate Byte Count (ABC), defined in RFC 3465. ABC is a way of increasing the congestion window (*cwnd*) more slowly in response to partial acknowledgements. Possible values are:

0 increase *cwnd* once per acknowledgement (no ABC)

1 increase *cwnd* once per acknowledgement of full sized segment

2 allow increase *cwnd* by two if acknowledgement is of two segments to compensate for delayed acknowledgements.

tcp_abort_on_overflow (Boolean; default: disabled; since Linux 2.4)
Enable resetting connections if the listening service is too slow and unable to keep up and accept them. It means that if overflow occurred due to a burst, the connection will recover. Enable this option *only* if you are really sure that the listening daemon cannot be tuned to accept connections faster. Enabling this option can harm the clients of your server.

tcp_adv_win_scale (integer; default: 2; since Linux 2.4)
Count buffering overhead as *bytes/2^tcp_adv_win_scale*, if *tcp_adv_win_scale* is greater than 0; or *bytes−bytes/2^(−tcp_adv_win_scale)*, if *tcp_adv_win_scale* is less than or equal to zero.

The socket receive buffer space is shared between the application and kernel. TCP maintains part of the buffer as the TCP window, this is the size of the receive window advertised to the other end. The rest of the space is used as the "application" buffer, used to isolate the network from scheduling and application latencies. The *tcp_adv_win_scale* default value of 2 implies that the space used for the application buffer is one fourth that of the total.

tcp_allowed_congestion_control (String; default: see text; since Linux 2.4.20)
Show/set the congestion control algorithm choices available to unprivileged processes (see the description of the **TCP_CONGESTION** socket option). The items in the list are separated by white space and terminated by a newline character. The list is a subset of those listed in *tcp_available_congestion_control*. The default value for this list is "reno" plus the default setting of *tcp_congestion_control*.

tcp_autocorking (Boolean; default: enabled; since Linux 3.14)
If this option is enabled, the kernel tries to coalesce small writes (from consecutive *write*(2) and *sendmsg*(2) calls) as much as possible, in order to decrease the total number of sent packets. Coalescing is done if at least one prior packet for the flow is waiting in Qdisc queues or device transmit queue. Applications can still use the **TCP_CORK** socket option to obtain optimal behavior when they know how/when to uncork their sockets.

tcp_available_congestion_control (String; read-only; since Linux 2.4.20)
Show a list of the congestion-control algorithms that are registered. The items in the list are separated by white space and terminated by a newline character. This list is a limiting set for the list in *tcp_allowed_congestion_control*. More congestion-control algorithms may be available as modules, but not loaded.

tcp_app_win (integer; default: 31; since Linux 2.4)
This variable defines how many bytes of the TCP window are reserved for buffering overhead.

A maximum of (*window/2^tcp_app_win*, mss) bytes in the window are reserved for the application buffer. A value of 0 implies that no amount is reserved.

tcp_base_mss (Integer; default: 512; since Linux 2.6.17)
The initial value of *search_low* to be used by the packetization layer Path MTU discovery (MTU probing). If MTU probing is enabled, this is the initial MSS used by the connection.

tcp_bic (Boolean; default: disabled; Linux 2.4.27/2.6.6 to Linux 2.6.13)
Enable BIC TCP congestion control algorithm. BIC-TCP is a sender-side-only change that ensures a linear RTT fairness under large windows while offering both scalability and bounded TCP-friendliness. The protocol combines two schemes called additive increase and binary search increase. When the congestion window is large, additive increase with a large increment ensures linear RTT fairness as well as good scalability. Under small congestion

windows, binary search increase provides TCP friendliness.

tcp_bic_low_window (integer; default: 14; Linux 2.4.27/2.6.6 to Linux 2.6.13)
Set the threshold window (in packets) where BIC TCP starts to adjust the congestion window. Below this threshold BIC TCP behaves the same as the default TCP Reno.

tcp_bic_fast_convergence (Boolean; default: enabled; Linux 2.4.27/2.6.6 to Linux 2.6.13)
Force BIC TCP to more quickly respond to changes in congestion window. Allows two flows sharing the same connection to converge more rapidly.

tcp_congestion_control (String; default: see text; since Linux 2.4.13)
Set the default congestion-control algorithm to be used for new connections. The algorithm "reno" is always available, but additional choices may be available depending on kernel configuration. The default value for this file is set as part of kernel configuration.

tcp_dma_copybreak (integer; default: 4096; since Linux 2.6.24)
Lower limit, in bytes, of the size of socket reads that will be offloaded to a DMA copy engine, if one is present in the system and the kernel was configured with the **CONFIG_NET_DMA** option.

tcp_dsack (Boolean; default: enabled; since Linux 2.4)
Enable RFC 2883 TCP Duplicate SACK support.

tcp_fastopen (Bitmask; default: 0x1; since Linux 3.7)
Enables RFC 7413 Fast Open support. The flag is used as a bitmap with the following values:

0x1 Enables client side Fast Open support

0x2 Enables server side Fast Open support

0x4 Allows client side to transmit data in SYN without Fast Open option

0x200 Allows server side to accept SYN data without Fast Open option

0x400 Enables Fast Open on all listeners without **TCP_FASTOPEN** socket option

tcp_fastopen_key (since Linux 3.7)
Set server side RFC 7413 Fast Open key to generate Fast Open cookie when server side Fast Open support is enabled.

tcp_ecn (Integer; default: see below; since Linux 2.4)
Enable RFC 3168 Explicit Congestion Notification.

This file can have one of the following values:

0 Disable ECN. Neither initiate nor accept ECN. This was the default up to and including Linux 2.6.30.

1 Enable ECN when requested by incoming connections and also request ECN on outgoing connection attempts.

2 Enable ECN when requested by incoming connections, but do not request ECN on outgoing connections. This value is supported, and is the default, since Linux 2.6.31.

When enabled, connectivity to some destinations could be affected due to older, misbehaving middle boxes along the path, causing connections to be dropped. However, to facilitate and encourage deployment with option 1, and to work around such buggy equipment, the **tcp_ecn_fallback** option has been introduced.

tcp_ecn_fallback (Boolean; default: enabled; since Linux 4.1)
Enable RFC 3168, Section 6.1.1.1. fallback. When enabled, outgoing ECN-setup SYNs that time out within the normal SYN retransmission timeout will be resent with CWR and ECE cleared.

tcp_fack (Boolean; default: enabled; since Linux 2.2)
Enable TCP Forward Acknowledgement support.

tcp_fin_timeout (integer; default: 60; since Linux 2.2)
This specifies how many seconds to wait for a final FIN packet before the socket is forcibly closed. This is strictly a violation of the TCP specification, but required to prevent denial-of-service attacks. In Linux 2.2, the default value was 180.

tcp_frto (integer; default: see below; since Linux 2.4.21/2.6)
Enable F-RTO, an enhanced recovery algorithm for TCP retransmission timeouts (RTOs). It is particularly beneficial in wireless environments where packet loss is typically due to random radio interference rather than intermediate router congestion. See RFC 4138 for more details.

This file can have one of the following values:

0 Disabled. This was the default up to and including Linux 2.6.23.

1 The basic version F-RTO algorithm is enabled.

2 Enable SACK-enhanced F-RTO if flow uses SACK. The basic version can be used also when SACK is in use though in that case scenario(s) exists where F-RTO interacts badly with the packet counting of the SACK-enabled TCP flow. This value is the default since Linux 2.6.24.

Before Linux 2.6.22, this parameter was a Boolean value, supporting just values 0 and 1 above.

tcp_frto_response (integer; default: 0; since Linux 2.6.22)
When F-RTO has detected that a TCP retransmission timeout was spurious (i.e., the timeout would have been avoided had TCP set a longer retransmission timeout), TCP has several options concerning what to do next. Possible values are:

0 Rate halving based; a smooth and conservative response, results in halved congestion window (*cwnd*) and slow-start threshold (*ssthresh*) after one RTT.

1 Very conservative response; not recommended because even though being valid, it interacts poorly with the rest of Linux TCP; halves *cwnd* and *ssthresh* immediately.

2 Aggressive response; undoes congestion-control measures that are now known to be unnecessary (ignoring the possibility of a lost retransmission that would require TCP to be more cautious); *cwnd* and *ssthresh* are restored to the values prior to timeout.

tcp_keepalive_intvl (integer; default: 75; since Linux 2.4)
The number of seconds between TCP keep-alive probes.

tcp_keepalive_probes (integer; default: 9; since Linux 2.2)
The maximum number of TCP keep-alive probes to send before giving up and killing the connection if no response is obtained from the other end.

tcp_keepalive_time (integer; default: 7200; since Linux 2.2)
The number of seconds a connection needs to be idle before TCP begins sending out keep-alive probes. Keep-alives are sent only when the **SO_KEEPALIVE** socket option is enabled. The default value is 7200 seconds (2 hours). An idle connection is terminated after approximately an additional 11 minutes (9 probes an interval of 75 seconds apart) when keep-alive is enabled.

Note that underlying connection tracking mechanisms and application timeouts may be much shorter.

tcp_low_latency (Boolean; default: disabled; since Linux 2.4.21/2.6; obsolete since Linux 4.14)
If enabled, the TCP stack makes decisions that prefer lower latency as opposed to higher throughput. It this option is disabled, then higher throughput is preferred. An example of an application where this default should be changed would be a Beowulf compute cluster. Since Linux 4.14, this file still exists, but its value is ignored.

tcp_max_orphans (integer; default: see below; since Linux 2.4)
The maximum number of orphaned (not attached to any user file handle) TCP sockets allowed in the system. When this number is exceeded, the orphaned connection is reset and a warning is printed. This limit exists only to prevent simple denial-of-service attacks. Lowering this limit is not recommended. Network conditions might require you to increase the number of orphans allowed, but note that each orphan can eat up to ~64 kB of unswappable memory. The default initial value is set equal to the kernel parameter NR_FILE. This initial default is adjusted depending on the memory in the system.

tcp_max_syn_backlog (integer; default: see below; since Linux 2.2)
The maximum number of queued connection requests which have still not received an acknowledgement from the connecting client. If this number is exceeded, the kernel will begin dropping requests. The default value of 256 is increased to 1024 when the memory present in the system is adequate or greater (>= 128 MB), and reduced to 128 for those systems with very low memory (<= 32 MB).

Before Linux 2.6.20, it was recommended that if this needed to be increased above 1024, the size of the SYNACK hash table (**TCP_SYNQ_HSIZE**) in *include/net/tcp.h* should be modified to keep

```
TCP_SYNQ_HSIZE * 16 <= tcp_max_syn_backlog
```

and the kernel should be recompiled. In Linux 2.6.20, the fixed sized **TCP_SYNQ_HSIZE** was removed in favor of dynamic sizing.

tcp_max_tw_buckets (integer; default: see below; since Linux 2.4)
The maximum number of sockets in TIME_WAIT state allowed in the system. This limit exists only to prevent simple denial-of-service attacks. The default value of NR_FILE*2 is adjusted depending on the memory in the system. If this number is exceeded, the socket is closed and a warning is printed.

tcp_moderate_rcvbuf (Boolean; default: enabled; since Linux 2.4.17/2.6.7)
If enabled, TCP performs receive buffer auto-tuning, attempting to automatically size the buffer (no greater than *tcp_rmem[2]*) to match the size required by the path for full throughput.

tcp_mem (since Linux 2.4)
This is a vector of 3 integers: [low, pressure, high]. These bounds, measured in units of the system page size, are used by TCP to track its memory usage. The defaults are calculated at boot time from the amount of available memory. (TCP can only use *low memory* for this, which is limited to around 900 megabytes on 32-bit systems. 64-bit systems do not suffer this limitation.)

low TCP doesn't regulate its memory allocation when the number of pages it has allocated globally is below this number.

pressure
When the amount of memory allocated by TCP exceeds this number of pages, TCP moderates its memory consumption. This memory pressure state is exited once the number of pages allocated falls below the *low* mark.

high The maximum number of pages, globally, that TCP will allocate. This value overrides any other limits imposed by the kernel.

tcp_mtu_probing (integer; default: 0; since Linux 2.6.17)
This parameter controls TCP Packetization-Layer Path MTU Discovery. The following values may be assigned to the file:

0 Disabled

1 Disabled by default, enabled when an ICMP black hole detected

2 Always enabled, use initial MSS of *tcp_base_mss*.

tcp_no_metrics_save (Boolean; default: disabled; since Linux 2.6.6)
By default, TCP saves various connection metrics in the route cache when the connection closes, so that connections established in the near future can use these to set initial conditions. Usually, this increases overall performance, but it may sometimes cause performance degradation. If *tcp_no_metrics_save* is enabled, TCP will not cache metrics on closing connections.

tcp_orphan_retries (integer; default: 8; since Linux 2.4)
The maximum number of attempts made to probe the other end of a connection which has been closed by our end.

tcp_reordering (integer; default: 3; since Linux 2.4)
The maximum a packet can be reordered in a TCP packet stream without TCP assuming packet loss and going into slow start. It is not advisable to change this number. This is a

packet reordering detection metric designed to minimize unnecessary back off and retransmits provoked by reordering of packets on a connection.

tcp_retrans_collapse (Boolean; default: enabled; since Linux 2.2)
Try to send full-sized packets during retransmit.

tcp_retries1 (integer; default: 3; since Linux 2.2)
The number of times TCP will attempt to retransmit a packet on an established connection normally, without the extra effort of getting the network layers involved. Once we exceed this number of retransmits, we first have the network layer update the route if possible before each new retransmit. The default is the RFC specified minimum of 3.

tcp_retries2 (integer; default: 15; since Linux 2.2)
The maximum number of times a TCP packet is retransmitted in established state before giving up. The default value is 15, which corresponds to a duration of approximately between 13 to 30 minutes, depending on the retransmission timeout. The RFC 1122 specified minimum limit of 100 seconds is typically deemed too short.

tcp_rfc1337 (Boolean; default: disabled; since Linux 2.2)
Enable TCP behavior conformant with RFC 1337. When disabled, if a RST is received in TIME_WAIT state, we close the socket immediately without waiting for the end of the TIME_WAIT period.

tcp_rmem (since Linux 2.4)
This is a vector of 3 integers: [min, default, max]. These parameters are used by TCP to regulate receive buffer sizes. TCP dynamically adjusts the size of the receive buffer from the defaults listed below, in the range of these values, depending on memory available in the system.

min minimum size of the receive buffer used by each TCP socket. The default value is the system page size. (On Linux 2.4, the default value is 4 kB, lowered to **PAGE_SIZE** bytes in low-memory systems.) This value is used to ensure that in memory pressure mode, allocations below this size will still succeed. This is not used to bound the size of the receive buffer declared using **SO_RCVBUF** on a socket.

default the default size of the receive buffer for a TCP socket. This value overwrites the initial default buffer size from the generic global *net.core.rmem_default* defined for all protocols. The default value is 87380 bytes. (On Linux 2.4, this will be lowered to 43689 in low-memory systems.) If larger receive buffer sizes are desired, this value should be increased (to affect all sockets). To employ large TCP windows, the *net.ipv4.tcp_window_scaling* must be enabled (default).

max the maximum size of the receive buffer used by each TCP socket. This value does not override the global *net.core.rmem_max*. This is not used to limit the size of the receive buffer declared using **SO_RCVBUF** on a socket. The default value is calculated using the formula

```
max(87380, min(4 MB, tcp_mem[1]*PAGE_SIZE/128))
```

(On Linux 2.4, the default is 87380*2 bytes, lowered to 87380 in low-memory systems).

tcp_sack (Boolean; default: enabled; since Linux 2.2)
Enable RFC 2018 TCP Selective Acknowledgements.

tcp_slow_start_after_idle (Boolean; default: enabled; since Linux 2.6.18)
If enabled, provide RFC 2861 behavior and time out the congestion window after an idle period. An idle period is defined as the current RTO (retransmission timeout). If disabled, the congestion window will not be timed out after an idle period.

tcp_stdurg (Boolean; default: disabled; since Linux 2.2)
If this option is enabled, then use the RFC 1122 interpretation of the TCP urgent-pointer field. According to this interpretation, the urgent pointer points to the last byte of urgent data. If this option is disabled, then use the BSD-compatible interpretation of the urgent pointer: the urgent pointer points to the first byte after the urgent data. Enabling this option may lead to interoperability problems.

tcp_syn_retries (integer; default: 6; since Linux 2.2)
The maximum number of times initial SYNs for an active TCP connection attempt will be retransmitted. This value should not be higher than 255. The default value is 6, which corresponds to retrying for up to approximately 127 seconds. Before Linux 3.7, the default value was 5, which (in conjunction with calculation based on other kernel parameters) corresponded to approximately 180 seconds.

tcp_synack_retries (integer; default: 5; since Linux 2.2)
The maximum number of times a SYN/ACK segment for a passive TCP connection will be retransmitted. This number should not be higher than 255.

tcp_syncookies (integer; default: 1; since Linux 2.2)
Enable TCP syncookies. The kernel must be compiled with **CONFIG_SYN_COOKIES**. The syncookies feature attempts to protect a socket from a SYN flood attack. This should be used as a last resort, if at all. This is a violation of the TCP protocol, and conflicts with other areas of TCP such as TCP extensions. It can cause problems for clients and relays. It is not recommended as a tuning mechanism for heavily loaded servers to help with overloaded or misconfigured conditions. For recommended alternatives see *tcp_max_syn_backlog*, *tcp_synack_retries*, and *tcp_abort_on_overflow*. Set to one of the following values:

0 Disable TCP syncookies.

1 Send out syncookies when the syn backlog queue of a socket overflows.

2 (since Linux 3.12) Send out syncookies unconditionally. This can be useful for network testing.

tcp_timestamps (integer; default: 1; since Linux 2.2)
Set to one of the following values to enable or disable RFC 1323 TCP timestamps:

0 Disable timestamps.

1 Enable timestamps as defined in RFC1323 and use random offset for each connection rather than only using the current time.

2 As for the value 1, but without random offsets. Setting *tcp_timestamps* to this value is meaningful since Linux 4.10.

tcp_tso_win_divisor (integer; default: 3; since Linux 2.6.9)
This parameter controls what percentage of the congestion window can be consumed by a single TCP Segmentation Offload (TSO) frame. The setting of this parameter is a tradeoff between burstiness and building larger TSO frames.

tcp_tw_recycle (Boolean; default: disabled; Linux 2.4 to Linux 4.11)
Enable fast recycling of TIME_WAIT sockets. Enabling this option is not recommended as the remote IP may not use monotonically increasing timestamps (devices behind NAT, devices with per-connection timestamp offsets). See RFC 1323 (PAWS) and RFC 6191.

tcp_tw_reuse (Boolean; default: disabled; since Linux 2.4.19/2.6)
Allow to reuse TIME_WAIT sockets for new connections when it is safe from protocol viewpoint. It should not be changed without advice/request of technical experts.

tcp_vegas_cong_avoid (Boolean; default: disabled; Linux 2.2 to Linux 2.6.13)
Enable TCP Vegas congestion avoidance algorithm. TCP Vegas is a sender-side-only change to TCP that anticipates the onset of congestion by estimating the bandwidth. TCP Vegas adjusts the sending rate by modifying the congestion window. TCP Vegas should provide less packet loss, but it is not as aggressive as TCP Reno.

tcp_westwood (Boolean; default: disabled; Linux 2.4.26/2.6.3 to Linux 2.6.13)
Enable TCP Westwood+ congestion control algorithm. TCP Westwood+ is a sender-side-only modification of the TCP Reno protocol stack that optimizes the performance of TCP congestion control. It is based on end-to-end bandwidth estimation to set congestion window and slow start threshold after a congestion episode. Using this estimation, TCP Westwood+ adaptively sets a slow start threshold and a congestion window which takes into account the bandwidth used at the time congestion is experienced. TCP Westwood+ significantly increases fairness with respect to TCP Reno in wired networks and throughput over wireless links.

tcp_window_scaling (Boolean; default: enabled; since Linux 2.2)
: Enable RFC 1323 TCP window scaling. This feature allows the use of a large window (> 64 kB) on a TCP connection, should the other end support it. Normally, the 16 bit window length field in the TCP header limits the window size to less than 64 kB. If larger windows are desired, applications can increase the size of their socket buffers and the window scaling option will be employed. If *tcp_window_scaling* is disabled, TCP will not negotiate the use of window scaling with the other end during connection setup.

tcp_wmem (since Linux 2.4)
: This is a vector of 3 integers: [min, default, max]. These parameters are used by TCP to regulate send buffer sizes. TCP dynamically adjusts the size of the send buffer from the default values listed below, in the range of these values, depending on memory available.

  min
  : Minimum size of the send buffer used by each TCP socket. The default value is the system page size. (On Linux 2.4, the default value is 4 kB.) This value is used to ensure that in memory pressure mode, allocations below this size will still succeed. This is not used to bound the size of the send buffer declared using **SO_SNDBUF** on a socket.

  default
  : The default size of the send buffer for a TCP socket. This value overwrites the initial default buffer size from the generic global */proc/sys/net/core/wmem_default* defined for all protocols. The default value is 16 kB. If larger send buffer sizes are desired, this value should be increased (to affect all sockets). To employ large TCP windows, the */proc/sys/net/ipv4/tcp_window_scaling* must be set to a nonzero value (default).

  max
  : The maximum size of the send buffer used by each TCP socket. This value does not override the value in */proc/sys/net/core/wmem_max*. This is not used to limit the size of the send buffer declared using **SO_SNDBUF** on a socket. The default value is calculated using the formula

    ```
    max(65536, min(4 MB, tcp_mem[1]*PAGE_SIZE/128))
    ```

    (On Linux 2.4, the default value is 128 kB, lowered 64 kB depending on low-memory systems.)

tcp_workaround_signed_windows (Boolean; default: disabled; since Linux 2.6.26)
: If enabled, assume that no receipt of a window-scaling option means that the remote TCP is broken and treats the window as a signed quantity. If disabled, assume that the remote TCP is not broken even if we do not receive a window scaling option from it.

Socket options

To set or get a TCP socket option, call *getsockopt*(2) to read or *setsockopt*(2) to write the option with the option level argument set to **IPPROTO_TCP**. Unless otherwise noted, *optval* is a pointer to an *int*. In addition, most **IPPROTO_IP** socket options are valid on TCP sockets. For more information see *ip*(7).

Following is a list of TCP-specific socket options. For details of some other socket options that are also applicable for TCP sockets, see *socket*(7).

TCP_CONGESTION (since Linux 2.6.13)
: The argument for this option is a string. This option allows the caller to set the TCP congestion control algorithm to be used, on a per-socket basis. Unprivileged processes are restricted to choosing one of the algorithms in *tcp_allowed_congestion_control* (described above). Privileged processes (**CAP_NET_ADMIN**) can choose from any of the available congestion-control algorithms (see the description of *tcp_available_congestion_control* above).

TCP_CORK (since Linux 2.2)
: If set, don't send out partial frames. All queued partial frames are sent when the option is cleared again. This is useful for prepending headers before calling *sendfile*(2), or for throughput optimization. As currently implemented, there is a 200 millisecond ceiling on the time for which output is corked by **TCP_CORK**. If this ceiling is reached, then queued data is automatically transmitted. This option can be combined with **TCP_NODELAY** only since Linux 2.5.71. This option should not be used in code intended to be portable.

TCP_DEFER_ACCEPT (since Linux 2.4)
Allow a listener to be awakened only when data arrives on the socket. Takes an integer value (seconds), this can bound the maximum number of attempts TCP will make to complete the connection. This option should not be used in code intended to be portable.

TCP_INFO (since Linux 2.4)
Used to collect information about this socket. The kernel returns a *struct tcp_info* as defined in the file */usr/include/linux/tcp.h*. This option should not be used in code intended to be portable.

TCP_KEEPCNT (since Linux 2.4)
The maximum number of keepalive probes TCP should send before dropping the connection. This option should not be used in code intended to be portable.

TCP_KEEPIDLE (since Linux 2.4)
The time (in seconds) the connection needs to remain idle before TCP starts sending keepalive probes, if the socket option **SO_KEEPALIVE** has been set on this socket. This option should not be used in code intended to be portable.

TCP_KEEPINTVL (since Linux 2.4)
The time (in seconds) between individual keepalive probes. This option should not be used in code intended to be portable.

TCP_LINGER2 (since Linux 2.4)
The lifetime of orphaned FIN_WAIT2 state sockets. This option can be used to override the system-wide setting in the file */proc/sys/net/ipv4/tcp_fin_timeout* for this socket. This is not to be confused with the *socket*(7) level option **SO_LINGER**. This option should not be used in code intended to be portable.

TCP_MAXSEG
The maximum segment size for outgoing TCP packets. In Linux 2.2 and earlier, and in Linux 2.6.28 and later, if this option is set before connection establishment, it also changes the MSS value announced to the other end in the initial packet. Values greater than the (eventual) interface MTU have no effect. TCP will also impose its minimum and maximum bounds over the value provided.

TCP_NODELAY
If set, disable the Nagle algorithm. This means that segments are always sent as soon as possible, even if there is only a small amount of data. When not set, data is buffered until there is a sufficient amount to send out, thereby avoiding the frequent sending of small packets, which results in poor utilization of the network. This option is overridden by **TCP_CORK**; however, setting this option forces an explicit flush of pending output, even if **TCP_CORK** is currently set.

TCP_QUICKACK (since Linux 2.4.4)
Enable quickack mode if set or disable quickack mode if cleared. In quickack mode, acks are sent immediately, rather than delayed if needed in accordance to normal TCP operation. This flag is not permanent, it only enables a switch to or from quickack mode. Subsequent operation of the TCP protocol will once again enter/leave quickack mode depending on internal protocol processing and factors such as delayed ack timeouts occurring and data transfer. This option should not be used in code intended to be portable.

TCP_SYNCNT (since Linux 2.4)
Set the number of SYN retransmits that TCP should send before aborting the attempt to connect. It cannot exceed 255. This option should not be used in code intended to be portable.

TCP_USER_TIMEOUT (since Linux 2.6.37)
This option takes an *unsigned int* as an argument. When the value is greater than 0, it specifies the maximum amount of time in milliseconds that transmitted data may remain unacknowledged, or buffered data may remain untransmitted (due to zero window size) before TCP will forcibly close the corresponding connection and return **ETIMEDOUT** to the application. If the option value is specified as 0, TCP will use the system default.

Increasing user timeouts allows a TCP connection to survive extended periods without end-to-end connectivity. Decreasing user timeouts allows applications to "fail fast", if so desired.

Otherwise, failure may take up to 20 minutes with the current system defaults in a normal WAN environment.

This option can be set during any state of a TCP connection, but is effective only during the synchronized states of a connection (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, and LAST-ACK). Moreover, when used with the TCP keepalive (**SO_KEEPALIVE**) option, **TCP_USER_TIMEOUT** will override keepalive to determine when to close a connection due to keepalive failure.

The option has no effect on when TCP retransmits a packet, nor when a keepalive probe is sent.

This option, like many others, will be inherited by the socket returned by *accept*(2), if it was set on the listening socket.

Further details on the user timeout feature can be found in RFC 793 and RFC 5482 ("TCP User Timeout Option").

TCP_WINDOW_CLAMP (since Linux 2.4)
Bound the size of the advertised window to this value. The kernel imposes a minimum size of SOCK_MIN_RCVBUF/2. This option should not be used in code intended to be portable.

TCP_FASTOPEN (since Linux 3.6)
This option enables Fast Open (RFC 7413) on the listener socket. The value specifies the maximum length of pending SYNs (similar to the backlog argument in *listen*(2)). Once enabled, the listener socket grants the TCP Fast Open cookie on incoming SYN with TCP Fast Open option.

More importantly it accepts the data in SYN with a valid Fast Open cookie and responds SYN-ACK acknowledging both the data and the SYN sequence. *accept*(2) returns a socket that is available for read and write when the handshake has not completed yet. Thus the data exchange can commence before the handshake completes. This option requires enabling the server-side support on sysctl *net.ipv4.tcp_fastopen* (see above). For TCP Fast Open client-side support, see *send*(2) **MSG_FASTOPEN** or **TCP_FASTOPEN_CONNECT** below.

TCP_FASTOPEN_CONNECT (since Linux 4.11)
This option enables an alternative way to perform Fast Open on the active side (client). When this option is enabled, *connect*(2) would behave differently depending on if a Fast Open cookie is available for the destination.

If a cookie is not available (i.e. first contact to the destination), *connect*(2) behaves as usual by sending a SYN immediately, except the SYN would include an empty Fast Open cookie option to solicit a cookie.

If a cookie is available, *connect*(2) would return 0 immediately but the SYN transmission is deferred. A subsequent *write*(2) or *sendmsg*(2) would trigger a SYN with data plus cookie in the Fast Open option. In other words, the actual connect operation is deferred until data is supplied.

Note: While this option is designed for convenience, enabling it does change the behaviors and certain system calls might set different *errno* values. With cookie present, *write*(2) or *sendmsg*(2) must be called right after *connect*(2) in order to send out SYN+data to complete 3WHS and establish connection. Calling *read*(2) right after *connect*(2) without *write*(2) will cause the blocking socket to be blocked forever.

The application should either set **TCP_FASTOPEN_CONNECT** socket option before *write*(2) or *sendmsg*(2), or call *write*(2) or *sendmsg*(2) with **MSG_FASTOPEN** flag directly, instead of both on the same connection.

Here is the typical call flow with this new option:

```
s = socket();
setsockopt(s, IPPROTO_TCP, TCP_FASTOPEN_CONNECT, 1, ...);
connect(s);
write(s); /* write() should always follow connect()
           * in order to trigger SYN to go out. */
read(s)/write(s);
```

```
/* ... */
close(s);
```

Sockets API

TCP provides limited support for out-of-band data, in the form of (a single byte of) urgent data. In Linux this means if the other end sends newer out-of-band data the older urgent data is inserted as normal data into the stream (even when **SO_OOBINLINE** is not set). This differs from BSD-based stacks.

Linux uses the BSD compatible interpretation of the urgent pointer field by default. This violates RFC 1122, but is required for interoperability with other stacks. It can be changed via */proc/sys/net/ipv4/tcp_stdurg*.

It is possible to peek at out-of-band data using the *recv*(2) **MSG_PEEK** flag.

Since Linux 2.4, Linux supports the use of **MSG_TRUNC** in the *flags* argument of *recv*(2) (and *recvmsg*(2)). This flag causes the received bytes of data to be discarded, rather than passed back in a caller-supplied buffer. Since Linux 2.4.4, **MSG_TRUNC** also has this effect when used in conjunction with **MSG_OOB** to receive out-of-band data.

Ioctls

The following *ioctl*(2) calls return information in *value*. The correct syntax is:

int *value***;**
error **= ioctl(***tcp_socket***,** *ioctl_type***, &***value***);**

ioctl_type is one of the following:

SIOCINQ

Returns the amount of queued unread data in the receive buffer. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCINQ** is defined in <*linux/sockios.h*>. Alternatively, you can use the synonymous **FIONREAD**, defined in <*sys/ioctl.h*>.

SIOCATMARK

Returns true (i.e., *value* is nonzero) if the inbound data stream is at the urgent mark.

If the **SO_OOBINLINE** socket option is set, and **SIOCATMARK** returns true, then the next read from the socket will return the urgent data. If the **SO_OOBINLINE** socket option is not set, and **SIOCATMARK** returns true, then the next read from the socket will return the bytes following the urgent data (to actually read the urgent data requires the **recv(MSG_OOB)** flag).

Note that a read never reads across the urgent mark. If an application is informed of the presence of urgent data via *select*(2) (using the *exceptfds* argument) or through delivery of a **SIGURG** signal, then it can advance up to the mark using a loop which repeatedly tests **SIOCATMARK** and performs a read (requesting any number of bytes) as long as **SIOCATMARK** returns false.

SIOCOUTQ

Returns the amount of unsent data in the socket send queue. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCOUTQ** is defined in <*linux/sockios.h*>. Alternatively, you can use the synonymous **TIOCOUTQ**, defined in <*sys/ioctl.h*>.

Error handling

When a network error occurs, TCP tries to resend the packet. If it doesn't succeed after some time, either **ETIMEDOUT** or the last received error on this connection is reported.

Some applications require a quicker error notification. This can be enabled with the **IPPROTO_IP** level **IP_RECVERR** socket option. When this option is enabled, all incoming errors are immediately passed to the user program. Use this option with care — it makes TCP less tolerant to routing changes and other normal network conditions.

ERRORS

EAFNOTSUPPORT

Passed socket address type in *sin_family* was not **AF_INET**.

EPIPE The other end closed the socket unexpectedly or a read is executed on a shut down socket.

ETIMEDOUT
The other end didn't acknowledge retransmitted data after some time.

Any errors defined for *ip*(7) or the generic socket layer may also be returned for TCP.

VERSIONS

Support for Explicit Congestion Notification, zero-copy *sendfile*(2), reordering support and some SACK extensions (DSACK) were introduced in Linux 2.4. Support for forward acknowledgement (FACK), TIME_WAIT recycling, and per-connection keepalive socket options were introduced in Linux 2.3.

BUGS

Not all errors are documented.

IPv6 is not described.

SEE ALSO

accept(2), *bind*(2), *connect*(2), *getsockopt*(2), *listen*(2), *recvmsg*(2), *sendfile*(2), *sendmsg*(2), *socket*(2), *ip*(7), *socket*(7)

The kernel source file *Documentation/networking/ip−sysctl.txt*.

RFC 793 for the TCP specification.
RFC 1122 for the TCP requirements and a description of the Nagle algorithm.
RFC 1323 for TCP timestamp and window scaling options.
RFC 1337 for a description of TIME_WAIT assassination hazards.
RFC 3168 for a description of Explicit Congestion Notification.
RFC 2581 for TCP congestion control algorithms.
RFC 2018 and RFC 2883 for SACK and extensions to SACK.

NAME

termio − System V terminal driver interface

DESCRIPTION

termio is the name of the old System V terminal driver interface. This interface defined a *termio* structure used to store terminal settings, and a range of *ioctl*(2) operations to get and set terminal attributes.

The **termio** interface is now obsolete: POSIX.1-1990 standardized a modified version of this interface, under the name **termios**. The POSIX.1 data structure differs slightly from the System V version, and POSIX.1 defined a suite of functions to replace the various *ioctl*(2) operations that existed in System V. (This was done because *ioctl*(2) was unstandardized, and its variadic third argument does not allow argument type checking.)

If you're looking for a page called "termio", then you can probably find most of the information that you seek in either *termios*(3) or *ioctl_tty*(2).

SEE ALSO

reset(1), *setterm*(1), *stty*(1), *ioctl_tty*(2), *termios*(3), *tty*(4)

NAME

thread-keyring − per-thread keyring

DESCRIPTION

The thread keyring is a keyring used to anchor keys on behalf of a process. It is created only when a thread requests it. The thread keyring has the name (description) *_tid*.

A special serial number value, **KEY_SPEC_THREAD_KEYRING**, is defined that can be used in lieu of the actual serial number of the calling thread's thread keyring.

From the *keyctl*(1) utility, '**@t**' can be used instead of a numeric key ID in much the same way, but as *keyctl*(1) is a program run after forking, this is of no utility.

Thread keyrings are not inherited across *clone*(2) and *fork*(2) and are cleared by *execve*(2). A thread keyring is destroyed when the thread that refers to it terminates.

Initially, a thread does not have a thread keyring. If a thread doesn't have a thread keyring when it is accessed, then it will be created if it is to be modified; otherwise the operation fails with the error **ENOKEY**.

SEE ALSO

keyctl(1), *keyctl*(3), *keyrings*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *user-keyring*(7), *user-session-keyring*(7)

NAME

time − overview of time and timers

DESCRIPTION

Real time and process time

Real time is defined as time measured from some fixed point, either from a standard point in the past (see the description of the Epoch and calendar time below), or from some point (e.g., the start) in the life of a process (*elapsed time*).

Process time is defined as the amount of CPU time used by a process. This is sometimes divided into *user* and *system* components. User CPU time is the time spent executing code in user mode. System CPU time is the time spent by the kernel executing in system mode on behalf of the process (e.g., executing system calls). The *time*(1) command can be used to determine the amount of CPU time consumed during the execution of a program. A program can determine the amount of CPU time it has consumed using *times*(2), *getrusage*(2), or *clock*(3).

The hardware clock

Most computers have a (battery-powered) hardware clock which the kernel reads at boot time in order to initialize the software clock. For further details, see *rtc*(4) and *hwclock*(8)

The software clock, HZ, and jiffies

The accuracy of various system calls that set timeouts, (e.g., *select*(2), *sigtimedwait*(2)) and measure CPU time (e.g., *getrusage*(2)) is limited by the resolution of the *software clock*, a clock maintained by the kernel which measures time in *jiffies*. The size of a jiffy is determined by the value of the kernel constant *HZ*.

The value of *HZ* varies across kernel versions and hardware platforms. On i386 the situation is as follows: on kernels up to and including Linux 2.4.x, HZ was 100, giving a jiffy value of 0.01 seconds; starting with Linux 2.6.0, HZ was raised to 1000, giving a jiffy of 0.001 seconds. Since Linux 2.6.13, the HZ value is a kernel configuration parameter and can be 100, 250 (the default) or 1000, yielding a jiffies value of, respectively, 0.01, 0.004, or 0.001 seconds. Since Linux 2.6.20, a further frequency is available: 300, a number that divides evenly for the common video frame rates (PAL, 25 Hz; NTSC, 30 Hz).

The *times*(2) system call is a special case. It reports times with a granularity defined by the kernel constant *USER_HZ*. User-space applications can determine the value of this constant using *sysconf(_SC_CLK_TCK)*.

System and process clocks; time namespaces

The kernel supports a range of clocks that measure various kinds of elapsed and virtual (i.e., consumed CPU) time. These clocks are described in *clock_gettime*(2). A few of the clocks are settable using *clock_settime*(2). The values of certain clocks are virtualized by time namespaces; see *time_namespaces*(7).

High-resolution timers

Before Linux 2.6.21, the accuracy of timer and sleep system calls (see below) was also limited by the size of the jiffy.

Since Linux 2.6.21, Linux supports high-resolution timers (HRTs), optionally configurable via **CONFIG_HIGH_RES_TIMERS**. On a system that supports HRTs, the accuracy of sleep and timer system calls is no longer constrained by the jiffy, but instead can be as accurate as the hardware allows (microsecond accuracy is typical of modern hardware). You can determine whether high-resolution timers are supported by checking the resolution returned by a call to *clock_getres*(2) or looking at the "resolution" entries in */proc/timer_list*.

HRTs are not supported on all hardware architectures. (Support is provided on x86, ARM, and PowerPC, among others.)

The Epoch

UNIX systems represent time in seconds since the *Epoch*, 1970-01-01 00:00:00 +0000 (UTC).

A program can determine the *calendar time* via the *clock_gettime*(2) **CLOCK_REALTIME** clock, which returns time (in seconds and nanoseconds) that have elapsed since the Epoch; *time*(2) provides similar information, but only with accuracy to the nearest second. The system time can be changed using *clock_settime*(2).

Broken-down time

Certain library functions use a structure of type *tm* to represent *broken-down time*, which stores time value separated out into distinct components (year, month, day, hour, minute, second, etc.). This structure is described in *tm*(3type), which also describes functions that convert between calendar time and broken-down time. Functions for converting between broken-down time and printable string representations of the time are described in *ctime*(3), *strftime*(3), and *strptime*(3).

Sleeping and setting timers

Various system calls and functions allow a program to sleep (suspend execution) for a specified period of time; see *nanosleep*(2), *clock_nanosleep*(2), and *sleep*(3).

Various system calls allow a process to set a timer that expires at some point in the future, and optionally at repeated intervals; see *alarm*(2), *getitimer*(2), *timerfd_create*(2), and *timer_create*(2).

Timer slack

Since Linux 2.6.28, it is possible to control the "timer slack" value for a thread. The timer slack is the length of time by which the kernel may delay the wake-up of certain system calls that block with a timeout. Permitting this delay allows the kernel to coalesce wake-up events, thus possibly reducing the number of system wake-ups and saving power. For more details, see the description of **PR_SET_TIMERSLACK** in *prctl*(2).

SEE ALSO

date(1), *time*(1), *timeout*(1), *adjtimex*(2), *alarm*(2), *clock_gettime*(2), *clock_nanosleep*(2), *getitimer*(2), *getrlimit*(2), *getrusage*(2), *gettimeofday*(2), *nanosleep*(2), *stat*(2), *time*(2), *timer_create*(2), *timerfd_create*(2), *times*(2), *utime*(2), *adjtime*(3), *clock*(3), *clock_getcpuclockid*(3), *ctime*(3), *ntp_adjtime*(3), *ntp_gettime*(3), *pthread_getcpuclockid*(3), *sleep*(3), *strftime*(3), *strptime*(3), *timeradd*(3), *usleep*(3), *rtc*(4), *time_namespaces*(7), *hwclock*(8)

NAME

time_namespaces − overview of Linux time namespaces

DESCRIPTION

Time namespaces virtualize the values of two system clocks:

- **CLOCK_MONOTONIC** (and likewise **CLOCK_MONOTONIC_COARSE** and **CLOCK_MONOTONIC_RAW**), a nonsettable clock that represents monotonic time since—as described by POSIX—"some unspecified point in the past".
- **CLOCK_BOOTTIME** (and likewise **CLOCK_BOOTTIME_ALARM**), a nonsettable clock that is identical to **CLOCK_MONOTONIC**, except that it also includes any time that the system is suspended.

Thus, the processes in a time namespace share per-namespace values for these clocks. This affects various APIs that measure against these clocks, including: *clock_gettime*(2), *clock_nanosleep*(2), *nanosleep*(2), *timer_settime*(2), *timerfd_settime*(2), and */proc/uptime*.

Currently, the only way to create a time namespace is by calling *unshare*(2) with the **CLONE_NEWTIME** flag. This call creates a new time namespace but does *not* place the calling process in the new namespace. Instead, the calling process's subsequently created children are placed in the new namespace. This allows clock offsets (see below) for the new namespace to be set before the first process is placed in the namespace. The */proc/*pid*/ns/time_for_children* symbolic link shows the time namespace in which the children of a process will be created. (A process can use a file descriptor opened on this symbolic link in a call to *setns*(2) in order to move into the namespace.)

/proc/*pid*/timens_offsets

Associated with each time namespace are offsets, expressed with respect to the initial time namespace, that define the values of the monotonic and boot-time clocks in that namespace. These offsets are exposed via the file */proc/*pid*/timens_offsets*. Within this file, the offsets are expressed as lines consisting of three space-delimited fields:

```
<clock-id> <offset-secs> <offset-nanosecs>
```

The *clock-id* is a string that identifies the clock whose offsets are being shown. This field is either *monotonic*, for **CLOCK_MONOTONIC**, or *boottime*, for **CLOCK_BOOTTIME**. The remaining fields express the offset (seconds plus nanoseconds) for the clock in this time namespace. These offsets are expressed relative to the clock values in the initial time namespace. The *offset-secs* value can be negative, subject to restrictions noted below; *offset-nanosecs* is an unsigned value.

In the initial time namespace, the contents of the *timens_offsets* file are as follows:

```
$ cat /proc/self/timens_offsets
monotonic           0         0
boottime            0         0
```

In a new time namespace that has had no member processes, the clock offsets can be modified by writing newline-terminated records of the same form to the *timens_offsets* file. The file can be written to multiple times, but after the first process has been created in or has entered the namespace, *write*(2)s on this file fail with the error **EACCES**. In order to write to the *timens_offsets* file, a process must have the **CAP_SYS_TIME** capability in the user namespace that owns the time namespace.

Writes to the *timens_offsets* file can fail with the following errors:

EINVAL
An *offset-nanosecs* value is greater than 999,999,999.

EINVAL
A *clock-id* value is not valid.

EPERM
The caller does not have the **CAP_SYS_TIME** capability.

ERANGE
An *offset-secs* value is out of range. In particular;

- *offset-secs* can't be set to a value which would make the current time on the corresponding clock inside the namespace a negative value; and

- *offset-secs* can't be set to a value such that the time on the corresponding clock inside the namespace would exceed half of the value of the kernel constant **KTIME_SEC_MAX** (this limits the clock value to a maximum of approximately 146 years).

In a new time namespace created by *unshare*(2), the contents of the *timens_offsets* file are inherited from the time namespace of the creating process.

NOTES

Use of time namespaces requires a kernel that is configured with the **CONFIG_TIME_NS** option.

Note that time namespaces do not virtualize the **CLOCK_REALTIME** clock. Virtualization of this clock was avoided for reasons of complexity and overhead within the kernel.

For compatibility with the initial implementation, when writing a *clock-id* to the */proc/*pid*/timens_offsets* file, the numerical values of the IDs can be written instead of the symbolic names show above; i.e., 1 instead of *monotonic*, and 7 instead of *boottime*. For readability, the use of the symbolic names over the numbers is preferred.

The motivation for adding time namespaces was to allow the monotonic and boot-time clocks to maintain consistent values during container migration and checkpoint/restore.

EXAMPLES

The following shell session demonstrates the operation of time namespaces. We begin by displaying the inode number of the time namespace of a shell in the initial time namespace:

```
$ readlink /proc/$$/ns/time
time:[4026531834]
```

Continuing in the initial time namespace, we display the system uptime using *uptime*(1) and use the *clock_times* example program shown in *clock_getres*(2) to display the values of various clocks:

```
$ uptime --pretty
up 21 hours, 17 minutes
$ ./clock_times
CLOCK_REALTIME : 1585989401.971 (18356 days +  8h 36m 41s)
CLOCK_TAI      : 1585989438.972 (18356 days +  8h 37m 18s)
CLOCK_MONOTONIC:      56338.247 (15h 38m 58s)
CLOCK_BOOTTIME :      76633.544 (21h 17m 13s)
```

We then use *unshare*(1) to create a time namespace and execute a *bash*(1) shell. From the new shell, we use the built-in **echo** command to write records to the *timens_offsets* file adjusting the offset for the **CLOCK_MONOTONIC** clock forward 2 days and the offset for the **CLOCK_BOOTTIME** clock forward 7 days:

```
$ PS1="ns2# " sudo unshare -T -- bash --norc
ns2# echo "monotonic $((2*24*60*60)) 0" > /proc/$$/timens_offsets
ns2# echo "boottime  $((7*24*60*60)) 0" > /proc/$$/timens_offsets
```

Above, we started the *bash*(1) shell with the **--norc** option so that no start-up scripts were executed. This ensures that no child processes are created from the shell before we have a chance to update the *timens_offsets* file.

We then use *cat*(1) to display the contents of the *timens_offsets* file. The execution of *cat*(1) creates the first process in the new time namespace, after which further attempts to update the *timens_offsets* file produce an error.

```
ns2# cat /proc/$$/timens_offsets
monotonic      172800         0
boottime       604800         0
ns2# echo "boottime $((9*24*60*60)) 0" > /proc/$$/timens_offsets
bash: echo: write error: Permission denied
```

Continuing in the new namespace, we execute *uptime*(1) and the *clock_times* example program:

```
ns2# uptime --pretty
up 1 week, 21 hours, 18 minutes
ns2# ./clock_times
CLOCK_REALTIME : 1585989457.056 (18356 days +  8h 37m 37s)
```

```
CLOCK_TAI      : 1585989494.057 (18356 days +  8h 38m 14s)
CLOCK_MONOTONIC:     229193.332 (2 days + 15h 39m 53s)
CLOCK_BOOTTIME :     681488.629 (7 days + 21h 18m  8s)
```

From the above output, we can see that the monotonic and boot-time clocks have different values in the new time namespace.

Examining the */proc/*pid*/ns/time* and */proc/*pid*/ns/time_for_children* symbolic links, we see that the shell is a member of the initial time namespace, but its children are created in the new namespace.

```
ns2# readlink /proc/$$/ns/time
time:[4026531834]
ns2# readlink /proc/$$/ns/time_for_children
time:[4026532900]
ns2# readlink /proc/self/ns/time   # Creates a child process
time:[4026532900]
```

Returning to the shell in the initial time namespace, we see that the monotonic and boot-time clocks are unaffected by the *timens_offsets* changes that were made in the other time namespace:

```
$ uptime --pretty
up 21 hours, 19 minutes
$ ./clock_times
CLOCK_REALTIME : 1585989401.971 (18356 days +  8h 38m 51s)
CLOCK_TAI      : 1585989438.972 (18356 days +  8h 39m 28s)
CLOCK_MONOTONIC:      56338.247 (15h 41m  8s)
CLOCK_BOOTTIME :      76633.544 (21h 19m 23s)
```

SEE ALSO

nsenter(1), *unshare*(1), *clock_settime*(2), *setns*(2), *unshare*(2), *namespaces*(7), *time*(7)

NAME

udp − User Datagram Protocol for IPv4

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/udp.h>

udp_socket = **socket(AF_INET, SOCK_DGRAM, 0);**

DESCRIPTION

This is an implementation of the User Datagram Protocol described in RFC 768. It implements a connectionless, unreliable datagram packet service. Packets may be reordered or duplicated before they arrive. UDP generates and checks checksums to catch transmission errors.

When a UDP socket is created, its local and remote addresses are unspecified. Datagrams can be sent immediately using *sendto*(2) or *sendmsg*(2) with a valid destination address as an argument. When *connect*(2) is called on the socket, the default destination address is set and datagrams can now be sent using *send*(2) or *write*(2) without specifying a destination address. It is still possible to send to other destinations by passing an address to *sendto*(2) or *sendmsg*(2). In order to receive packets, the socket can be bound to a local address first by using *bind*(2). Otherwise, the socket layer will automatically assign a free local port out of the range defined by */proc/sys/net/ipv4/ip_local_port_range* and bind the socket to **INADDR_ANY**.

All receive operations return only one packet. When the packet is smaller than the passed buffer, only that much data is returned; when it is bigger, the packet is truncated and the **MSG_TRUNC** flag is set. **MSG_WAITALL** is not supported.

IP options may be sent or received using the socket options described in *ip*(7). They are processed by the kernel only when the appropriate */proc* parameter is enabled (but still passed to the user even when it is turned off). See *ip*(7).

When the **MSG_DONTROUTE** flag is set on sending, the destination address must refer to a local interface address and the packet is sent only to that interface.

By default, Linux UDP does path MTU (Maximum Transmission Unit) discovery. This means the kernel will keep track of the MTU to a specific target IP address and return **EMSGSIZE** when a UDP packet write exceeds it. When this happens, the application should decrease the packet size. Path MTU discovery can be also turned off using the **IP_MTU_DISCOVER** socket option or the */proc/sys/net/ipv4/ip_no_pmtu_disc* file; see *ip*(7) for details. When turned off, UDP will fragment outgoing UDP packets that exceed the interface MTU. However, disabling it is not recommended for performance and reliability reasons.

Address format

UDP uses the IPv4 *sockaddr_in* address format described in *ip*(7).

Error handling

All fatal errors will be passed to the user as an error return even when the socket is not connected. This includes asynchronous errors received from the network. You may get an error for an earlier packet that was sent on the same socket. This behavior differs from many other BSD socket implementations which don't pass any errors unless the socket is connected. Linux's behavior is mandated by **RFC 1122**.

For compatibility with legacy code, in Linux 2.0 and 2.2 it was possible to set the **SO_BSDCOMPAT SOL_SOCKET** option to receive remote errors only when the socket has been connected (except for **EPROTO** and **EMSGSIZE**). Locally generated errors are always passed. Support for this socket option was removed in later kernels; see *socket*(7) for further information.

When the **IP_RECVERR** option is enabled, all errors are stored in the socket error queue, and can be received by *recvmsg*(2) with the **MSG_ERRQUEUE** flag set.

/proc interfaces

System-wide UDP parameter settings can be accessed by files in the directory */proc/sys/net/ipv4/* .

udp_mem (since Linux 2.6.25)
: This is a vector of three integers governing the number of pages allowed for queueing by all UDP sockets.

min Below this number of pages, UDP is not bothered about its memory appetite. When the amount of memory allocated by UDP exceeds this number, UDP starts to moderate memory usage.

pressure
This value was introduced to follow the format of *tcp_mem* (see *tcp*(7)).

max Number of pages allowed for queueing by all UDP sockets.

Defaults values for these three items are calculated at boot time from the amount of available memory.

udp_rmem_min (integer; default value: PAGE_SIZE; since Linux 2.6.25)
Minimal size, in bytes, of receive buffers used by UDP sockets in moderation. Each UDP socket is able to use the size for receiving data, even if total pages of UDP sockets exceed *udp_mem* pressure.

udp_wmem_min (integer; default value: PAGE_SIZE; since Linux 2.6.25)
Minimal size, in bytes, of send buffer used by UDP sockets in moderation. Each UDP socket is able to use the size for sending data, even if total pages of UDP sockets exceed *udp_mem* pressure.

Socket options

To set or get a UDP socket option, call *getsockopt*(2) to read or *setsockopt*(2) to write the option with the option level argument set to **IPPROTO_UDP**. Unless otherwise noted, *optval* is a pointer to an *int*.

Following is a list of UDP-specific socket options. For details of some other socket options that are also applicable for UDP sockets, see *socket*(7).

UDP_CORK (since Linux 2.5.44)
If this option is enabled, then all data output on this socket is accumulated into a single datagram that is transmitted when the option is disabled. This option should not be used in code intended to be portable.

UDP_SEGMENT (since Linux 4.18)
Enables UDP segmentation offload. Segmentation offload reduces *send*(2) cost by transferring multiple datagrams worth of data as a single large packet through the kernel transmit path, even when that exceeds MTU. As late as possible, the large packet is split by segment size into a series of datagrams. This segmentation offload step is deferred to hardware if supported, else performed in software. This option takes a value in the range [**0**, **USHRT_MAX**] that sets the segment size: the size of datagram payload, excluding the UDP header. The segment size must be chosen such that at most 64 datagrams are sent in a single call and that the datagrams after segmentation meet the same MTU rules that apply to datagrams sent without this option. Segmentation offload depends on checksum offload, as datagram checksums are computed after segmentation. The option may also be set for individual *sendmsg*(2) calls by passing it as a *cmsg*(3). A value of zero disables the feature. This option should not be used in code intended to be portable.

UDP_GRO (since Linux 5.0)
Enables UDP receive offload. If enabled, the socket may receive multiple datagrams worth of data as a single large buffer, together with a *cmsg*(3) that holds the segment size. This option is the inverse of segmentation offload. It reduces receive cost by handling multiple datagrams worth of data as a single large packet in the kernel receive path, even when that exceeds MTU. This option should not be used in code intended to be portable.

Ioctls

These ioctls can be accessed using *ioctl*(2). The correct syntax is:

int *value***;**
error **= ioctl(***udp_socket***,** *ioctl_type***, &***value***);**

FIONREAD (**SIOCINQ**)
Gets a pointer to an integer as argument. Returns the size of the next pending datagram in the integer in bytes, or 0 when no datagram is pending. **Warning:** Using **FIONREAD**, it is impossible to distinguish the case where no datagram is pending from the case where the next

pending datagram contains zero bytes of data. It is safer to use *select*(2), *poll*(2), or *epoll*(7) to distinguish these cases.

TIOCOUTQ (**SIOCOUTQ**)
Returns the number of data bytes in the local send queue. Supported only with Linux 2.4 and above.

In addition, all ioctls documented in *ip*(7) and *socket*(7) are supported.

ERRORS

All errors documented for *socket*(7) or *ip*(7) may be returned by a send or receive on a UDP socket.

ECONNREFUSED
No receiver was associated with the destination address. This might be caused by a previous packet sent over the socket.

VERSIONS

IP_RECVERR is a new feature in Linux 2.2.

SEE ALSO

ip(7), *raw*(7), *socket*(7), *udplite*(7)

The kernel source file *Documentation/networking/ip−sysctl.txt*.

RFC 768 for the User Datagram Protocol.
RFC 1122 for the host requirements.
RFC 1191 for a description of path MTU discovery.

NAME

udplite − Lightweight User Datagram Protocol

SYNOPSIS

#include <sys/socket.h>

sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE);

DESCRIPTION

This is an implementation of the Lightweight User Datagram Protocol (UDP-Lite), as described in RFC 3828.

UDP-Lite is an extension of UDP (RFC 768) to support variable-length checksums. This has advantages for some types of multimedia transport that may be able to make use of slightly damaged datagrams, rather than having them discarded by lower-layer protocols.

The variable-length checksum coverage is set via a *setsockopt*(2) option. If this option is not set, the only difference from UDP is in using a different IP protocol identifier (IANA number 136).

The UDP-Lite implementation is a full extension of *udp*(7)—that is, it shares the same API and API behavior, and in addition offers two socket options to control the checksum coverage.

Address format

UDP-Litev4 uses the *sockaddr_in* address format described in *ip*(7). UDP-Litev6 uses the *sockaddr_in6* address format described in *ipv6*(7).

Socket options

To set or get a UDP-Lite socket option, call *getsockopt*(2) to read or *setsockopt*(2) to write the option with the option level argument set to **IPPROTO_UDPLITE**. In addition, all **IPPROTO_UDP** socket options are valid on a UDP-Lite socket. See *udp*(7) for more information.

The following two options are specific to UDP-Lite.

UDPLITE_SEND_CSCOV

This option sets the sender checksum coverage and takes an *int* as argument, with a checksum coverage value in the range 0..2^16-1.

A value of 0 means that the entire datagram is always covered. Values from 1−7 are illegal (RFC 3828, 3.1) and are rounded up to the minimum coverage of 8.

With regard to IPv6 jumbograms (RFC 2675), the UDP-Litev6 checksum coverage is limited to the first 2^16-1 octets, as per RFC 3828, 3.5. Higher values are therefore silently truncated to 2^16-1. If in doubt, the current coverage value can always be queried using *getsockopt*(2).

UDPLITE_RECV_CSCOV

This is the receiver-side analogue and uses the same argument format and value range as **UDPLITE_SEND_CSCOV**. This option is not required to enable traffic with partial checksum coverage. Its function is that of a traffic filter: when enabled, it instructs the kernel to drop all packets which have a coverage *less* than the specified coverage value.

When the value of **UDPLITE_RECV_CSCOV** exceeds the actual packet coverage, incoming packets are silently dropped, but may generate a warning message in the system log.

ERRORS

All errors documented for *udp*(7) may be returned. UDP-Lite does not add further errors.

FILES

/proc/net/snmp

Basic UDP-Litev4 statistics counters.

/proc/net/snmp6

Basic UDP-Litev6 statistics counters.

VERSIONS

UDP-Litev4/v6 first appeared in Linux 2.6.20.

BUGS

Where glibc support is missing, the following definitions are needed:

```
#define IPPROTO_UDPLITE     136
#define UDPLITE_SEND_CSCOV  10
```

```
#define UDPLITE_RECV_CSCOV  11
```

SEE ALSO

ip(7), *ipv6*(7), *socket*(7), *udp*(7)

RFC 3828 for the Lightweight User Datagram Protocol (UDP-Lite).

Documentation/networking/udplite.txt in the Linux kernel source tree

NAME

unicode − universal character set

DESCRIPTION

The international standard ISO/IEC 10646 defines the Universal Character Set (UCS). UCS contains all characters of all other character set standards. It also guarantees "round-trip compatibility"; in other words, conversion tables can be built such that no information is lost when a string is converted from any other encoding to UCS and back.

UCS contains the characters required to represent practically all known languages. This includes not only the Latin, Greek, Cyrillic, Hebrew, Arabic, Armenian, and Georgian scripts, but also Chinese, Japanese and Korean Han ideographs as well as scripts such as Hiragana, Katakana, Hangul, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Khmer, Bopomofo, Tibetan, Runic, Ethiopic, Canadian Syllabics, Cherokee, Mongolian, Ogham, Myanmar, Sinhala, Thaana, Yi, and others. For scripts not yet covered, research on how to best encode them for computer usage is still going on and they will be added eventually. This might eventually include not only Hieroglyphs and various historic Indo-European languages, but even some selected artistic scripts such as Tengwar, Cirth, and Klingon. UCS also covers a large number of graphical, typographical, mathematical, and scientific symbols, including those provided by TeX, Postscript, APL, MS-DOS, MS-Windows, Macintosh, OCR fonts, as well as many word processing and publishing systems, and more are being added.

The UCS standard (ISO/IEC 10646) describes a 31-bit character set architecture consisting of 128 24-bit *groups*, each divided into 256 16-bit *planes* made up of 256 8-bit *rows* with 256 *column* positions, one for each character. Part 1 of the standard (ISO/IEC 10646-1) defines the first 65534 code positions (0x0000 to 0xfffd), which form the *Basic Multilingual Plane* (BMP), that is plane 0 in group 0. Part 2 of the standard (ISO/IEC 10646-2) adds characters to group 0 outside the BMP in several *supplementary planes* in the range 0x10000 to 0x10ffff. There are no plans to add characters beyond 0x10ffff to the standard, therefore of the entire code space, only a small fraction of group 0 will ever be actually used in the foreseeable future. The BMP contains all characters found in the commonly used other character sets. The supplemental planes added by ISO/IEC 10646-2 cover only more exotic characters for special scientific, dictionary printing, publishing industry, higher-level protocol and enthusiast needs.

The representation of each UCS character as a 2-byte word is referred to as the UCS-2 form (only for BMP characters), whereas UCS-4 is the representation of each character by a 4-byte word. In addition, there exist two encoding forms UTF-8 for backward compatibility with ASCII processing software and UTF-16 for the backward-compatible handling of non-BMP characters up to 0x10ffff by UCS-2 software.

The UCS characters 0x0000 to 0x007f are identical to those of the classic US-ASCII character set and the characters in the range 0x0000 to 0x00ff are identical to those in ISO/IEC 8859-1 (Latin-1).

Combining characters

Some code points in UCS have been assigned to *combining characters*. These are similar to the nonspacing accent keys on a typewriter. A combining character just adds an accent to the previous character. The most important accented characters have codes of their own in UCS, however, the combining character mechanism allows us to add accents and other diacritical marks to any character. The combining characters always follow the character which they modify. For example, the German character Umlaut-A ("Latin capital letter A with diaeresis") can either be represented by the precomposed UCS code 0x00c4, or alternatively as the combination of a normal "Latin capital letter A" followed by a "combining diaeresis": 0x0041 0x0308.

Combining characters are essential for instance for encoding the Thai script or for mathematical typesetting and users of the International Phonetic Alphabet.

Implementation levels

As not all systems are expected to support advanced mechanisms like combining characters, ISO/IEC 10646-1 specifies the following three *implementation levels* of UCS:

Level 1 Combining characters and Hangul Jamo (a variant encoding of the Korean script, where a Hangul syllable glyph is coded as a triplet or pair of vowel/consonant codes) are not supported.

Level 2 In addition to level 1, combining characters are now allowed for some languages where they are essential (e.g., Thai, Lao, Hebrew, Arabic, Devanagari, Malayalam).

Level 3 All UCS characters are supported.

The Unicode 3.0 Standard published by the Unicode Consortium contains exactly the UCS Basic Multilingual Plane at implementation level 3, as described in ISO/IEC 10646-1:2000. Unicode 3.1 added the supplemental planes of ISO/IEC 10646-2. The Unicode standard and technical reports published by the Unicode Consortium provide much additional information on the semantics and recommended usages of various characters. They provide guidelines and algorithms for editing, sorting, comparing, normalizing, converting, and displaying Unicode strings.

Unicode under Linux

Under GNU/Linux, the C type *wchar_t* is a signed 32-bit integer type. Its values are always interpreted by the C library as UCS code values (in all locales), a convention that is signaled by the GNU C library to applications by defining the constant **__STDC_ISO_10646__** as specified in the ISO C99 standard.

UCS/Unicode can be used just like ASCII in input/output streams, terminal communication, plaintext files, filenames, and environment variables in the ASCII compatible UTF-8 multibyte encoding. To signal the use of UTF-8 as the character encoding to all applications, a suitable *locale* has to be selected via environment variables (e.g., "LANG=en_GB.UTF-8").

The **nl_langinfo(CODESET)** function returns the name of the selected encoding. Library functions such as *wctomb*(3) and *mbsrtowcs*(3) can be used to transform the internal *wchar_t* characters and strings into the system character encoding and back and *wcwidth*(3) tells how many positions (0–2) the cursor is advanced by the output of a character.

Private Use Areas (PUA)

In the Basic Multilingual Plane, the range 0xe000 to 0xf8ff will never be assigned to any characters by the standard and is reserved for private usage. For the Linux community, this private area has been subdivided further into the range 0xe000 to 0xefff which can be used individually by any end-user and the Linux zone in the range 0xf000 to 0xf8ff where extensions are coordinated among all Linux users. The registry of the characters assigned to the Linux zone is maintained by LANANA and the registry itself is *Documentation/admin−guide/unicode.rst* in the Linux kernel sources (or *Documentation/unicode.txt* before Linux 4.10).

Two other planes are reserved for private usage, plane 15 (Supplementary Private Use Area-A, range 0xf0000 to 0xffffd) and plane 16 (Supplementary Private Use Area-B, range 0x100000 to 0x10fffd).

Literature

- Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane. International Standard ISO/IEC 10646-1, International Organization for Standardization, Geneva, 2000.

  This is the official specification of UCS. Available from .

- The Unicode Standard, Version 3.0. The Unicode Consortium, Addison-Wesley, Reading, MA, 2000, ISBN 0-201-61633-5.

- S. Harbison, G. Steele. C: A Reference Manual. Fourth edition, Prentice Hall, Englewood Cliffs, 1995, ISBN 0-13-326224-3.

  A good reference book about the C programming language. The fourth edition covers the 1994 Amendment 1 to the ISO C90 standard, which adds a large number of new C library functions for handling wide and multibyte character encodings, but it does not yet cover ISO C99, which improved wide and multibyte character support even further.

- Unicode Technical Reports.

- Markus Kuhn: UTF-8 and Unicode FAQ for UNIX/Linux.

- Bruno Haible: Unicode HOWTO.

SEE ALSO

locale(1), *setlocale*(3), *charsets*(7), *utf-8*(7)

NAME

units − decimal and binary prefixes

DESCRIPTION

Decimal prefixes

The SI system of units uses prefixes that indicate powers of ten. A kilometer is 1000 meter, and a megawatt is 1000000 watt. Below the standard prefixes.

Prefix	Name	Value
q	quecto	10^−30 = 0.000000000000000000000000000001
r	ronto	10^−27 = 0.000000000000000000000000001
y	yocto	10^−24 = 0.000000000000000000000001
z	zepto	10^−21 = 0.000000000000000000001
a	atto	10^−18 = 0.000000000000000001
f	femto	10^−15 = 0.000000000000001
p	pico	10^−12 = 0.000000000001
n	nano	10^−9 = 0.000000001
µ	micro	10^−6 = 0.000001
m	milli	10^−3 = 0.001
c	centi	10^−2 = 0.01
d	deci	10^−1 = 0.1
da	deka	10^ 1 = 10
h	hecto	10^ 2 = 100
k	kilo	10^ 3 = 1000
M	mega	10^ 6 = 1000000
G	giga	10^ 9 = 1000000000
T	tera	10^12 = 1000000000000
P	peta	10^15 = 1000000000000000
E	exa	10^18 = 1000000000000000000
Z	zetta	10^21 = 1000000000000000000000
Y	yotta	10^24 = 1000000000000000000000000
R	ronna	10^27 = 1000000000000000000000000000
Q	quetta	10^30 = 1000000000000000000000000000000

The symbol for micro is the Greek letter mu, often written u in an ASCII context where this Greek letter is not available.

Binary prefixes

The binary prefixes resemble the decimal ones, but have an additional 'i' (and "Ki" starts with a capital 'K'). The names are formed by taking the first syllable of the names of the decimal prefix with roughly the same size, followed by "bi" for "binary".

Prefix	Name	Value
Ki	kibi	2^10 = 1024
Mi	mebi	2^20 = 1048576
Gi	gibi	2^30 = 1073741824
Ti	tebi	2^40 = 1099511627776
Pi	pebi	2^50 = 1125899906842624
Ei	exbi	2^60 = 1152921504606846976
Zi	zebi	2^70 = 1180591620717411303424
Yi	yobi	2^80 = 1208925819614629174706176

Discussion

Before these binary prefixes were introduced, it was fairly common to use k=1000 and K=1024, just like b=bit, B=byte. Unfortunately, the M is capital already, and cannot be capitalized to indicate binary-ness.

At first that didn’t matter too much, since memory modules and disks came in sizes that were powers of two, so everyone knew that in such contexts "kilobyte" and "megabyte" meant 1024 and 1048576 bytes, respectively. What originally was a sloppy use of the prefixes "kilo" and "mega" started to become regarded as the "real true meaning" when computers were involved. But then disk technology changed, and disk sizes became arbitrary numbers. After a period of uncertainty all disk manufacturers settled on the standard, namely k=1000, M=1000 k, G=1000 M.

The situation was messy: in the 14k4 modems, k=1000; in the 1.44 MB diskettes, M=1024000; and so on. In 1998 the IEC approved the standard that defines the binary prefixes given above, enabling people to be precise and unambiguous.

Thus, today, MB = 1000000 B and MiB = 1048576 B.

In the free software world programs are slowly being changed to conform. When the Linux kernel boots and says

```
hda: 120064896 sectors (61473 MB) w/2048KiB Cache
```

the MB are megabytes and the KiB are kibibytes.

SEE ALSO

The International System of Units.

NAME

unix − sockets for local interprocess communication

SYNOPSIS

#include <sys/socket.h>
#include <sys/un.h>

unix_socket = **socket(AF_UNIX, type, 0);**
error = **socketpair(AF_UNIX, type, 0, int ****sv***);**

DESCRIPTION

The **AF_UNIX** (also known as **AF_LOCAL**) socket family is used to communicate between processes on the same machine efficiently. Traditionally, UNIX domain sockets can be either unnamed, or bound to a filesystem pathname (marked as being of type socket). Linux also supports an abstract namespace which is independent of the filesystem.

Valid socket types in the UNIX domain are: **SOCK_STREAM**, for a stream-oriented socket; **SOCK_DGRAM**, for a datagram-oriented socket that preserves message boundaries (as on most UNIX implementations, UNIX domain datagram sockets are always reliable and don't reorder datagrams); and (since Linux 2.6.4) **SOCK_SEQPACKET**, for a sequenced-packet socket that is connection-oriented, preserves message boundaries, and delivers messages in the order that they were sent.

UNIX domain sockets support passing file descriptors or process credentials to other processes using ancillary data.

Address format

A UNIX domain socket address is represented in the following structure:

```
struct sockaddr_un {
    sa_family_t sun_family;               /* AF_UNIX */
    char        sun_path[108];            /* Pathname */
};
```

The *sun_family* field always contains **AF_UNIX**. On Linux, *sun_path* is 108 bytes in size; see also BUGS, below.

Various system calls (for example, *bind*(2), *connect*(2), and *sendto*(2)) take a *sockaddr_un* argument as input. Some other system calls (for example, *getsockname*(2), *getpeername*(2), *recvfrom*(2), and *accept*(2)) return an argument of this type.

Three types of address are distinguished in the *sockaddr_un* structure:

pathname
: a UNIX domain socket can be bound to a null-terminated filesystem pathname using *bind*(2). When the address of a pathname socket is returned (by one of the system calls noted above), its length is

  ```
  offsetof(struct sockaddr_un, sun_path) + strlen(sun_path) + 1
  ```

  and *sun_path* contains the null-terminated pathname. (On Linux, the above **offsetof**() expression equates to the same value as *sizeof(sa_family_t)*, but some other implementations include other fields before *sun_path*, so the **offsetof**() expression more portably describes the size of the address structure.)

  For further details of pathname sockets, see below.

unnamed
: A stream socket that has not been bound to a pathname using *bind*(2) has no name. Likewise, the two sockets created by *socketpair*(2) are unnamed. When the address of an unnamed socket is returned, its length is *sizeof(sa_family_t)*, and *sun_path* should not be inspected.

abstract
: an abstract socket address is distinguished (from a pathname socket) by the fact that *sun_path[0]* is a null byte ('\0'). The socket's address in this namespace is given by the additional bytes in *sun_path* that are covered by the specified length of the address structure. (Null bytes in the name have no special significance.) The name has no connection with filesystem pathnames. When the address of an abstract socket is returned, the returned *addrlen* is greater than *sizeof(sa_family_t)* (i.e., greater than 2), and the name of the socket is

contained in the first *(addrlen − sizeof(sa_family_t))* bytes of *sun_path*.

Pathname sockets

When binding a socket to a pathname, a few rules should be observed for maximum portability and ease of coding:

- The pathname in *sun_path* should be null-terminated.
- The length of the pathname, including the terminating null byte, should not exceed the size of *sun_path*.
- The *addrlen* argument that describes the enclosing *sockaddr_un* structure should have a value of at least:

```
offsetof(struct sockaddr_un, sun_path)+strlen(addr.sun_path)+1
```

  or, more simply, *addrlen* can be specified as *sizeof(struct sockaddr_un)*.

There is some variation in how implementations handle UNIX domain socket addresses that do not follow the above rules. For example, some (but not all) implementations append a null terminator if none is present in the supplied *sun_path*.

When coding portable applications, keep in mind that some implementations have *sun_path* as short as 92 bytes.

Various system calls (**accept**(2), *recvfrom*(2), *getsockname*(2), *getpeername*(2)) return socket address structures. When applied to UNIX domain sockets, the value-result *addrlen* argument supplied to the call should be initialized as above. Upon return, the argument is set to indicate the *actual* size of the address structure. The caller should check the value returned in this argument: if the output value exceeds the input value, then there is no guarantee that a null terminator is present in *sun_path*. (See BUGS.)

Pathname socket ownership and permissions

In the Linux implementation, pathname sockets honor the permissions of the directory they are in. Creation of a new socket fails if the process does not have write and search (execute) permission on the directory in which the socket is created.

On Linux, connecting to a stream socket object requires write permission on that socket; sending a datagram to a datagram socket likewise requires write permission on that socket. POSIX does not make any statement about the effect of the permissions on a socket file, and on some systems (e.g., older BSDs), the socket permissions are ignored. Portable programs should not rely on this feature for security.

When creating a new socket, the owner and group of the socket file are set according to the usual rules. The socket file has all permissions enabled, other than those that are turned off by the process *umask*(2).

The owner, group, and permissions of a pathname socket can be changed (using *chown*(2) and *chmod*(2)).

Abstract sockets

Socket permissions have no meaning for abstract sockets: the process *umask*(2) has no effect when binding an abstract socket, and changing the ownership and permissions of the object (via *fchown*(2) and *fchmod*(2)) has no effect on the accessibility of the socket.

Abstract sockets automatically disappear when all open references to the socket are closed.

The abstract socket namespace is a nonportable Linux extension.

Socket options

For historical reasons, these socket options are specified with a **SOL_SOCKET** type even though they are **AF_UNIX** specific. They can be set with *setsockopt*(2) and read with *getsockopt*(2) by specifying **SOL_SOCKET** as the socket family.

SO_PASSCRED

Enabling this socket option causes receipt of the credentials of the sending process in an **SCM_CREDENTIALS ancillary** message in each subsequently received message. The returned credentials are those specified by the sender using **SCM_CREDENTIALS**, or a default that includes the sender's PID, real user ID, and real group ID, if the sender did not

specify **SCM_CREDENTIALS** ancillary data.

When this option is set and the socket is not yet connected, a unique name in the abstract namespace will be generated automatically.

The value given as an argument to *setsockopt*(2) and returned as the result of *getsockopt*(2) is an integer boolean flag.

SO_PASSSEC
Enables receiving of the SELinux security label of the peer socket in an ancillary message of type **SCM_SECURITY** (see below).

The value given as an argument to *setsockopt*(2) and returned as the result of *getsockopt*(2) is an integer boolean flag.

The **SO_PASSSEC** option is supported for UNIX domain datagram sockets since Linux 2.6.18; support for UNIX domain stream sockets was added in Linux 4.2.

SO_PEEK_OFF
See *socket*(7).

SO_PEERCRED
This read-only socket option returns the credentials of the peer process connected to this socket. The returned credentials are those that were in effect at the time of the call to *connect*(2), *listen*(2), or *socketpair*(2).

The argument to *getsockopt*(2) is a pointer to a *ucred* structure; define the **_GNU_SOURCE** feature test macro to obtain the definition of that structure from <*sys/socket.h*>.

The use of this option is possible only for connected **AF_UNIX** stream sockets and for **AF_UNIX** stream and datagram socket pairs created using *socketpair*(2).

SO_PEERSEC
This read-only socket option returns the security context of the peer socket connected to this socket. By default, this will be the same as the security context of the process that created the peer socket unless overridden by the policy or by a process with the required permissions.

The argument to *getsockopt*(2) is a pointer to a buffer of the specified length in bytes into which the security context string will be copied. If the buffer length is less than the length of the security context string, then *getsockopt*(2) returns −1, sets *errno* to **ERANGE**, and returns the required length via *optlen*. The caller should allocate at least **NAME_MAX** bytes for the buffer initially, although this is not guaranteed to be sufficient. Resizing the buffer to the returned length and retrying may be necessary.

The security context string may include a terminating null character in the returned length, but is not guaranteed to do so: a security context "foo" might be represented as either {'f','o','o'} of length 3 or {'f','o','o','\0'} of length 4, which are considered to be interchangeable. The string is printable, does not contain non-terminating null characters, and is in an unspecified encoding (in particular, it is not guaranteed to be ASCII or UTF-8).

The use of this option for sockets in the **AF_UNIX** address family is supported since Linux 2.6.2 for connected stream sockets, and since Linux 4.18 also for stream and datagram socket pairs created using *socketpair*(2).

Autobind feature
If a *bind*(2) call specifies *addrlen* as *sizeof(sa_family_t)*, or the **SO_PASSCRED** socket option was specified for a socket that was not explicitly bound to an address, then the socket is autobound to an abstract address. The address consists of a null byte followed by 5 bytes in the character set *[0−9a−f]*. Thus, there is a limit of 2^20 autobind addresses. (From Linux 2.1.15, when the autobind feature was added, 8 bytes were used, and the limit was thus 2^32 autobind addresses. The change to 5 bytes came in Linux 2.3.15.)

Sockets API
The following paragraphs describe domain-specific details and unsupported features of the sockets API for UNIX domain sockets on Linux.

UNIX domain sockets do not support the transmission of out-of-band data (the **MSG_OOB** flag for *send*(2) and *recv*(2)).

The *send*(2) **MSG_MORE** flag is not supported by UNIX domain sockets.

Before Linux 3.4, the use of **MSG_TRUNC** in the *flags* argument of *recv*(2) was not supported by UNIX domain sockets.

The **SO_SNDBUF** socket option does have an effect for UNIX domain sockets, but the **SO_RCVBUF** option does not. For datagram sockets, the **SO_SNDBUF** value imposes an upper limit on the size of outgoing datagrams. This limit is calculated as the doubled (see *socket*(7)) option value less 32 bytes used for overhead.

Ancillary messages

Ancillary data is sent and received using *sendmsg*(2) and *recvmsg*(2). For historical reasons, the ancillary message types listed below are specified with a **SOL_SOCKET** type even though they are **AF_UNIX** specific. To send them, set the *cmsg_level* field of the struct *cmsghdr* to **SOL_SOCKET** and the *cmsg_type* field to the type. For more information, see *cmsg*(3).

SCM_RIGHTS

Send or receive a set of open file descriptors from another process. The data portion contains an integer array of the file descriptors.

Commonly, this operation is referred to as "passing a file descriptor" to another process. However, more accurately, what is being passed is a reference to an open file description (see *open*(2)), and in the receiving process it is likely that a different file descriptor number will be used. Semantically, this operation is equivalent to duplicating (**dup**(2)) a file descriptor into the file descriptor table of another process.

If the buffer used to receive the ancillary data containing file descriptors is too small (or is absent), then the ancillary data is truncated (or discarded) and the excess file descriptors are automatically closed in the receiving process.

If the number of file descriptors received in the ancillary data would cause the process to exceed its **RLIMIT_NOFILE** resource limit (see *getrlimit*(2)), the excess file descriptors are automatically closed in the receiving process.

The kernel constant **SCM_MAX_FD** defines a limit on the number of file descriptors in the array. Attempting to send an array larger than this limit causes *sendmsg*(2) to fail with the error **EINVAL**. **SCM_MAX_FD** has the value 253 (or 255 before Linux 2.6.38).

SCM_CREDENTIALS

Send or receive UNIX credentials. This can be used for authentication. The credentials are passed as a *struct ucred* ancillary message. This structure is defined in *<sys/socket.h>* as follows:

```
struct ucred {
    pid_t pid;    /* Process ID of the sending process */
    uid_t uid;    /* User ID of the sending process */
    gid_t gid;    /* Group ID of the sending process */
};
```

Since glibc 2.8, the **_GNU_SOURCE** feature test macro must be defined (before including *any* header files) in order to obtain the definition of this structure.

The credentials which the sender specifies are checked by the kernel. A privileged process is allowed to specify values that do not match its own. The sender must specify its own process ID (unless it has the capability **CAP_SYS_ADMIN**, in which case the PID of any existing process may be specified), its real user ID, effective user ID, or saved set-user-ID (unless it has **CAP_SETUID**), and its real group ID, effective group ID, or saved set-group-ID (unless it has **CAP_SETGID**).

To receive a *struct ucred* message, the **SO_PASSCRED** option must be enabled on the socket.

SCM_SECURITY

Receive the SELinux security context (the security label) of the peer socket. The received ancillary data is a null-terminated string containing the security context. The receiver should allocate at least **NAME_MAX** bytes in the data portion of the ancillary message for this data.

To receive the security context, the **SO_PASSSEC** option must be enabled on the socket (see above).

When sending ancillary data with *sendmsg*(2), only one item of each of the above types may be included in the sent message.

At least one byte of real data should be sent when sending ancillary data. On Linux, this is required to successfully send ancillary data over a UNIX domain stream socket. When sending ancillary data over a UNIX domain datagram socket, it is not necessary on Linux to send any accompanying real data. However, portable applications should also include at least one byte of real data when sending ancillary data over a datagram socket.

When receiving from a stream socket, ancillary data forms a kind of barrier for the received data. For example, suppose that the sender transmits as follows:

(1) *sendmsg*(2) of four bytes, with no ancillary data.
(2) *sendmsg*(2) of one byte, with ancillary data.
(3) *sendmsg*(2) of four bytes, with no ancillary data.

Suppose that the receiver now performs *recvmsg*(2) calls each with a buffer size of 20 bytes. The first call will receive five bytes of data, along with the ancillary data sent by the second *sendmsg*(2) call. The next call will receive the remaining four bytes of data.

If the space allocated for receiving incoming ancillary data is too small then the ancillary data is truncated to the number of headers that will fit in the supplied buffer (or, in the case of an **SCM_RIGHTS** file descriptor list, the list of file descriptors may be truncated). If no buffer is provided for incoming ancillary data (i.e., the *msg_control* field of the *msghdr* structure supplied to *recvmsg*(2) is NULL), then the incoming ancillary data is discarded. In both of these cases, the **MSG_CTRUNC** flag will be set in the *msg.msg_flags* value returned by *recvmsg*(2).

Ioctls

The following *ioctl*(2) calls return information in *value*. The correct syntax is:

int *value***;**
error **= ioctl(***unix_socket***,** *ioctl_type***, &***value***);**

ioctl_type can be:

SIOCINQ
For **SOCK_STREAM** sockets, this call returns the number of unread bytes in the receive buffer. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCINQ** is defined in <*linux/sockios.h*>. Alternatively, you can use the synonymous **FIONREAD**, defined in <*sys/ioctl.h*>. For **SOCK_DGRAM** sockets, the returned value is the same as for Internet domain datagram sockets; see *udp*(7).

ERRORS

EADDRINUSE
The specified local address is already in use or the filesystem socket object already exists.

EBADF
This error can occur for *sendmsg*(2) when sending a file descriptor as ancillary data over a UNIX domain socket (see the description of **SCM_RIGHTS**, above), and indicates that the file descriptor number that is being sent is not valid (e.g., it is not an open file descriptor).

ECONNREFUSED
The remote address specified by *connect*(2) was not a listening socket. This error can also occur if the target pathname is not a socket.

ECONNRESET
Remote socket was unexpectedly closed.

EFAULT
User memory address was not valid.

EINVAL
Invalid argument passed. A common cause is that the value **AF_UNIX** was not specified in the *sun_type* field of passed addresses, or the socket was in an invalid state for the applied operation.

EISCONN
connect(2) called on an already connected socket or a target address was specified on a connected socket.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOENT
The pathname in the remote address specified to *connect*(2) did not exist.

ENOMEM
Out of memory.

ENOTCONN
Socket operation needs a target address, but the socket is not connected.

EOPNOTSUPP
Stream operation called on non-stream oriented socket or tried to use the out-of-band data option.

EPERM
The sender passed invalid credentials in the *struct ucred*.

EPIPE Remote socket was closed on a stream socket. If enabled, a **SIGPIPE** is sent as well. This can be avoided by passing the **MSG_NOSIGNAL** flag to *send*(2) or *sendmsg*(2).

EPROTONOSUPPORT
Passed protocol is not **AF_UNIX**.

EPROTOTYPE
Remote socket does not match the local socket type (**SOCK_DGRAM** versus **SOCK_STREAM**).

ESOCKTNOSUPPORT
Unknown socket type.

ESRCH
While sending an ancillary message containing credentials (**SCM_CREDENTIALS**), the caller specified a PID that does not match any existing process.

ETOOMANYREFS
This error can occur for *sendmsg*(2) when sending a file descriptor as ancillary data over a UNIX domain socket (see the description of **SCM_RIGHTS**, above). It occurs if the number of "in-flight" file descriptors exceeds the **RLIMIT_NOFILE** resource limit and the caller does not have the **CAP_SYS_RESOURCE** capability. An in-flight file descriptor is one that has been sent using *sendmsg*(2) but has not yet been accepted in the recipient process using *recvmsg*(2).

This error is diagnosed since mainline Linux 4.5 (and in some earlier kernel versions where the fix has been backported). In earlier kernel versions, it was possible to place an unlimited number of file descriptors in flight, by sending each file descriptor with *sendmsg*(2) and then closing the file descriptor so that it was not accounted against the **RLIMIT_NOFILE** resource limit.

Other errors can be generated by the generic socket layer or by the filesystem while generating a filesystem socket object. See the appropriate manual pages for more information.

VERSIONS

SCM_CREDENTIALS and the abstract namespace were introduced with Linux 2.2 and should not be used in portable programs. (Some BSD-derived systems also support credential passing, but the implementation details differ.)

NOTES

Binding to a socket with a filename creates a socket in the filesystem that must be deleted by the caller when it is no longer needed (using *unlink*(2)). The usual UNIX close-behind semantics apply; the socket can be unlinked at any time and will be finally removed from the filesystem when the last reference to it is closed.

To pass file descriptors or credentials over a **SOCK_STREAM** socket, you must send or receive at

least one byte of nonancillary data in the same *sendmsg*(2) or *recvmsg*(2) call.

UNIX domain stream sockets do not support the notion of out-of-band data.

BUGS

When binding a socket to an address, Linux is one of the implementations that append a null terminator if none is supplied in *sun_path*. In most cases this is unproblematic: when the socket address is retrieved, it will be one byte longer than that supplied when the socket was bound. However, there is one case where confusing behavior can result: if 108 non-null bytes are supplied when a socket is bound, then the addition of the null terminator takes the length of the pathname beyond *sizeof(sun_path)*. Consequently, when retrieving the socket address (for example, via *accept*(2)), if the input *addrlen* argument for the retrieving call is specified as *sizeof(struct sockaddr_un)*, then the returned address structure *won't* have a null terminator in *sun_path*.

In addition, some implementations don't require a null terminator when binding a socket (the *addrlen* argument is used to determine the length of *sun_path*) and when the socket address is retrieved on these implementations, there is no null terminator in *sun_path*.

Applications that retrieve socket addresses can (portably) code to handle the possibility that there is no null terminator in *sun_path* by respecting the fact that the number of valid bytes in the pathname is:

```
strnlen(addr.sun_path, addrlen - offsetof(sockaddr_un, sun_path))
```

Alternatively, an application can retrieve the socket address by allocating a buffer of size *sizeof(struct sockaddr_un)+1* that is zeroed out before the retrieval. The retrieving call can specify *addrlen* as *sizeof(struct sockaddr_un)*, and the extra zero byte ensures that there will be a null terminator for the string returned in *sun_path*:

```
void *addrp;

addrlen = sizeof(struct sockaddr_un);
addrp = malloc(addrlen + 1);
if (addrp == NULL)
    /* Handle error */ ;
memset(addrp, 0, addrlen + 1);

if (getsockname(sfd, (struct sockaddr *) addrp, &addrlen)) == -1)
    /* handle error */ ;

printf("sun_path = %s\n", ((struct sockaddr_un *) addrp)->sun_path);
```

This sort of messiness can be avoided if it is guaranteed that the applications that *create* pathname sockets follow the rules outlined above under *Pathname sockets*.

EXAMPLES

The following code demonstrates the use of sequenced-packet sockets for local interprocess communication. It consists of two programs. The server program waits for a connection from the client program. The client sends each of its command-line arguments in separate messages. The server treats the incoming messages as integers and adds them up. The client sends the command string "END". The server sends back a message containing the sum of the client's integers. The client prints the sum and exits. The server waits for the next client to connect. To stop the server, the client is called with the command-line argument "DOWN".

The following output was recorded while running the server in the background and repeatedly executing the client. Execution of the server program ends when it receives the "DOWN" command.

Example output

```
$ ./server &
[1] 25887
$ ./client 3 4
Result = 7
$ ./client 11 -5
Result = 6
$ ./client DOWN
Result = 0
```

```
[1]+  Done                    ./server
$
```

Program source

```
/*
 * File connection.h
 */

#define SOCKET_NAME "/tmp/9Lq7BNBnBycd6nxy.socket"
#define BUFFER_SIZE 12
```

```
/*
 * File server.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#include "connection.h"

int
main(void)
{
    int                 down_flag = 0;
    int                 ret;
    int                 connection_socket;
    int                 data_socket;
    int                 result;
    ssize_t             r, w;
    struct sockaddr_un  name;
    char                buffer[BUFFER_SIZE];

    /* Create local socket. */

    connection_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (connection_socket == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    /*
     * For portability clear the whole structure, since some
     * implementations have additional (nonstandard) fields in
     * the structure.
     */

    memset(&name, 0, sizeof(name));

    /* Bind socket to socket name. */

    name.sun_family = AF_UNIX;
    strncpy(name.sun_path, SOCKET_NAME, sizeof(name.sun_path) - 1);

    ret = bind(connection_socket, (const struct sockaddr *) &name,
               sizeof(name));
```

```
if (ret == -1) {
    perror("bind");
    exit(EXIT_FAILURE);
}

/*
 * Prepare for accepting connections. The backlog size is set
 * to 20. So while one request is being processed other requests
 * can be waiting.
 */

ret = listen(connection_socket, 20);
if (ret == -1) {
    perror("listen");
    exit(EXIT_FAILURE);
}

/* This is the main loop for handling connections. */

for (;;) {

    /* Wait for incoming connection. */

    data_socket = accept(connection_socket, NULL, NULL);
    if (data_socket == -1) {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    result = 0;
    for (;;) {

        /* Wait for next data packet. */

        r = read(data_socket, buffer, sizeof(buffer));
        if (r == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }

        /* Ensure buffer is 0-terminated. */

        buffer[sizeof(buffer) - 1] = 0;

        /* Handle commands. */

        if (!strncmp(buffer, "DOWN", sizeof(buffer))) {
            down_flag = 1;
            continue;
        }

        if (!strncmp(buffer, "END", sizeof(buffer))) {
            break;
        }

        if (down_flag) {
            continue;
        }
```

```
            /* Add received summand. */

            result += atoi(buffer);
        }

        /* Send result. */

        sprintf(buffer, "%d", result);
        w = write(data_socket, buffer, sizeof(buffer));
        if (w == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }

        /* Close socket. */

        close(data_socket);

        /* Quit on DOWN command. */

        if (down_flag) {
            break;
        }
    }

    close(connection_socket);

    /* Unlink the socket. */

    unlink(SOCKET_NAME);

    exit(EXIT_SUCCESS);
}
```

```
/*
 * File client.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#include "connection.h"

int
main(int argc, char *argv[])
{
    int                 ret;
    int                 data_socket;
    ssize_t             r, w;
    struct sockaddr_un  addr;
    char                buffer[BUFFER_SIZE];

    /* Create local socket. */

    data_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
```

```
if (data_socket == -1) {
    perror("socket");
    exit(EXIT_FAILURE);
}

/*
 * For portability clear the whole structure, since some
 * implementations have additional (nonstandard) fields in
 * the structure.
 */

memset(&addr, 0, sizeof(addr));

/* Connect socket to socket address. */

addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, SOCKET_NAME, sizeof(addr.sun_path) - 1);

ret = connect(data_socket, (const struct sockaddr *) &addr,
              sizeof(addr));
if (ret == -1) {
    fprintf(stderr, "The server is down.\n");
    exit(EXIT_FAILURE);
}

/* Send arguments. */

for (int i = 1; i < argc; ++i) {
    w = write(data_socket, argv[i], strlen(argv[i]) + 1);
    if (w == -1) {
        perror("write");
        break;
    }
}

/* Request result. */

strcpy(buffer, "END");
w = write(data_socket, buffer, strlen(buffer) + 1);
if (w == -1) {
    perror("write");
    exit(EXIT_FAILURE);
}

/* Receive result. */

r = read(data_socket, buffer, sizeof(buffer));
if (r == -1) {
    perror("read");
    exit(EXIT_FAILURE);
}

/* Ensure buffer is 0-terminated. */

buffer[sizeof(buffer) - 1] = 0;

printf("Result = %s\n", buffer);

/* Close socket. */
```

```
    close(data_socket);

    exit(EXIT_SUCCESS);
}
```

For examples of the use of **SCM_RIGHTS**, see *cmsg*(3) and *seccomp_unotify*(2).

SEE ALSO

recvmsg(2), *sendmsg*(2), *socket*(2), *socketpair*(2), *cmsg*(3), *capabilities*(7), *credentials*(7), *socket*(7), *udp*(7)

NAME

uri, url, urn − uniform resource identifier (URI), including a URL or URN

SYNOPSIS

URI = [*absoluteURI* | *relativeURI*] ["**#**" *fragment*]

absoluteURI = *scheme* "**:**" (*hierarchical_part* | *opaque_part*)

relativeURI = (*net_path* | *absolute_path* | *relative_path*) ["**?**" *query*]

scheme = "**http**" | "**ftp**" | "**gopher**" | "**mailto**" | "**news**" | "**telnet**" | "**file**" | "**ftp**" | "**man**" | "**info**" | "**whatis**" | "**ldap**" | "**wais**" | ...

hierarchical_part = (*net_path* | *absolute_path*) ["**?**" *query*]

net_path = "//" *authority* [*absolute_path*]

absolute_path = "/" *path_segments*

relative_path = *relative_segment* [*absolute_path*]

DESCRIPTION

A Uniform Resource Identifier (URI) is a short string of characters identifying an abstract or physical resource (for example, a web page). A Uniform Resource Locator (URL) is a URI that identifies a resource through its primary access mechanism (e.g., its network "location"), rather than by name or some other attribute of that resource. A Uniform Resource Name (URN) is a URI that must remain globally unique and persistent even when the resource ceases to exist or becomes unavailable.

URIs are the standard way to name hypertext link destinations for tools such as web browsers. The string "http://www.kernel.org" is a URL (and thus it is also a URI). Many people use the term URL loosely as a synonym for URI (though technically URLs are a subset of URIs).

URIs can be absolute or relative. An absolute identifier refers to a resource independent of context, while a relative identifier refers to a resource by describing the difference from the current context. Within a relative path reference, the complete path segments "." and ".." have special meanings: "the current hierarchy level" and "the level above this hierarchy level", respectively, just like they do in UNIX-like systems. A path segment which contains a colon character can't be used as the first segment of a relative URI path (e.g., "this:that"), because it would be mistaken for a scheme name; precede such segments with ./ (e.g., "./this:that"). Note that descendants of MS-DOS (e.g., Microsoft Windows) replace devicename colons with the vertical bar ("|") in URIs, so "C:" becomes "C|".

A fragment identifier, if included, refers to a particular named portion (fragment) of a resource; text after a '#' identifies the fragment. A URI beginning with '#' refers to that fragment in the current resource.

Usage

There are many different URI schemes, each with specific additional rules and meanings, but they are intentionally made to be as similar as possible. For example, many URL schemes permit the authority to be the following format, called here an *ip_server* (square brackets show what's optional):

ip_server = [*user* [: *password*] @] *host* [: *port*]

This format allows you to optionally insert a username, a user plus password, and/or a port number. The *host* is the name of the host computer, either its name as determined by DNS or an IP address (numbers separated by periods). Thus the URI <http://fred:fredpassword@example.com:8080/> logs into a web server on host example.com as fred (using fredpassword) using port 8080. Avoid including a password in a URI if possible because of the many security risks of having a password written down. If the URL supplies a username but no password, and the remote server requests a password, the program interpreting the URL should request one from the user.

Here are some of the most common schemes in use on UNIX-like systems that are understood by many tools. Note that many tools using URIs also have internal schemes or specialized schemes; see those tools' documentation for information on those schemes.

http − Web (HTTP) server

http://*ip_server*/*path*
http://*ip_server*/*path*?*query*

This is a URL accessing a web (HTTP) server. The default port is 80. If the path refers to a directory,

the web server will choose what to return; usually if there is a file named "index.html" or "index.htm" its content is returned, otherwise, a list of the files in the current directory (with appropriate links) is generated and returned. An example is <http://lwn.net>.

A query can be given in the archaic "isindex" format, consisting of a word or phrase and not including an equal sign (=). A query can also be in the longer "GET" format, which has one or more query entries of the form *key=value* separated by the ampersand character (&). Note that *key* can be repeated more than once, though it's up to the web server and its application programs to determine if there's any meaning to that. There is an unfortunate interaction with HTML/XML/SGML and the GET query format; when such URIs with more than one key are embedded in SGML/XML documents (including HTML), the ampersand (&) has to be rewritten as &. Note that not all queries use this format; larger forms may be too long to store as a URI, so they use a different interaction mechanism (called POST) which does not include the data in the URI. See the Common Gateway Interface specification at for more information.

ftp − File Transfer Protocol (FTP)

ftp://*ip_server*/*path*

This is a URL accessing a file through the file transfer protocol (FTP). The default port (for control) is 21. If no username is included, the username "anonymous" is supplied, and in that case many clients provide as the password the requestor's Internet email address. An example is <ftp://ftp.is.co.za/rfc/rfc1808.txt>.

gopher − Gopher server

gopher://*ip_server*/*gophertype selector*
gopher://*ip_server*/*gophertype selector*%09*search*
gopher://*ip_server*/*gophertype selector*%09*search*%09*gopher+_string*

The default gopher port is 70. *gophertype* is a single-character field to denote the Gopher type of the resource to which the URL refers. The entire path may also be empty, in which case the delimiting "/" is also optional and the gophertype defaults to "1".

selector is the Gopher selector string. In the Gopher protocol, Gopher selector strings are a sequence of octets which may contain any octets except 09 hexadecimal (US-ASCII HT or tab), 0A hexadecimal (US-ASCII character LF), and 0D (US-ASCII character CR).

mailto − Email address

mailto:*email-address*

This is an email address, usually of the form *name@hostname*. See *mailaddr*(7) for more information on the correct format of an email address. Note that any % character must be rewritten as %25. An example is <mailto:dwheeler@dwheeler.com>.

news − Newsgroup or News message

news:*newsgroup-name*
news:*message-id*

A *newsgroup-name* is a period-delimited hierarchical name, such as "comp.infosystems.www.misc". If <newsgroup-name> is "*" (as in <news:*>), it is used to refer to "all available news groups". An example is <news:comp.lang.ada>.

A *message-id* corresponds to the Message-ID of IETF RFC 1036, without the enclosing "<" and ">"; it takes the form *unique@ full_domain_name*. A message identifier may be distinguished from a news group name by the presence of the "@" character.

telnet − Telnet login

telnet://*ip_server*/

The Telnet URL scheme is used to designate interactive text services that may be accessed by the Telnet protocol. The final "/" character may be omitted. The default port is 23. An example is <telnet://melvyl.ucop.edu/>.

file − Normal file

file://*ip_server*/*path_segments*
file:*path_segments*

This represents a file or directory accessible locally. As a special case, *ip_server* can be the string "localhost" or the empty string; this is interpreted as "the machine from which the URL is being interpreted". If the path is to a directory, the viewer should display the directory's contents with links to each containee; not all viewers currently do this. KDE supports generated files through the URL <file:/cgi-bin>. If the given file isn't found, browser writers may want to try to expand the filename via filename globbing (see *glob*(7) and *glob*(3)).

The second format (e.g., <file:/etc/passwd>) is a correct format for referring to a local file. However, older standards did not permit this format, and some programs don't recognize this as a URI. A more portable syntax is to use an empty string as the server name, for example, <file:///etc/passwd>; this form does the same thing and is easily recognized by pattern matchers and older programs as a URI. Note that if you really mean to say "start from the current location", don't specify the scheme at all; use a relative address like <../test.txt>, which has the side-effect of being scheme-independent. An example of this scheme is <file:///etc/passwd>.

man − Man page documentation

man:*command-name*
man:*command-name*(*section*)

This refers to local online manual (man) reference pages. The command name can optionally be followed by a parenthesis and section number; see *man*(7) for more information on the meaning of the section numbers. This URI scheme is unique to UNIX-like systems (such as Linux) and is not currently registered by the IETF. An example is <man:ls(1)>.

info − Info page documentation

info:*virtual-filename*
info:*virtual-filename#nodename*
info:(*virtual-filename*)
info:(*virtual-filename*)*nodename*

This scheme refers to online info reference pages (generated from texinfo files), a documentation format used by programs such as the GNU tools. This URI scheme is unique to UNIX-like systems (such as Linux) and is not currently registered by the IETF. As of this writing, GNOME and KDE differ in their URI syntax and do not accept the other's syntax. The first two formats are the GNOME format; in nodenames all spaces are written as underscores. The second two formats are the KDE format; spaces in nodenames must be written as spaces, even though this is forbidden by the URI standards. It's hoped that in the future most tools will understand all of these formats and will always accept underscores for spaces in nodenames. In both GNOME and KDE, if the form without the nodename is used the nodename is assumed to be "Top". Examples of the GNOME format are <info:gcc> and <info:gcc#G++_and_GCC>. Examples of the KDE format are <info:(gcc)> and <info:(gcc)G++ and GCC>.

whatis − Documentation search

whatis:*string*

This scheme searches the database of short (one-line) descriptions of commands and returns a list of descriptions containing that string. Only complete word matches are returned. See *whatis*(1)This URI scheme is unique to UNIX-like systems (such as Linux) and is not currently registered by the IETF.

ghelp − GNOME help documentation

ghelp:*name-of-application*

This loads GNOME help for the given application. Note that not much documentation currently exists in this format.

ldap − Lightweight Directory Access Protocol

ldap://*hostport*
ldap://*hostport*/
ldap://*hostport*/*dn*
ldap://*hostport*/*dn*?*attributes*
ldap://*hostport*/*dn*?*attributes*?*scope*
ldap://*hostport*/*dn*?*attributes*?*scope*?*filter*
ldap://*hostport*/*dn*?*attributes*?*scope*?*filter*?*extensions*

This scheme supports queries to the Lightweight Directory Access Protocol (LDAP), a protocol for querying a set of servers for hierarchically organized information (such as people and computing resources). See RFC 2255 for more information on the LDAP URL scheme. The components of this URL are:

hostport
: the LDAP server to query, written as a hostname optionally followed by a colon and the port number. The default LDAP port is TCP port 389. If empty, the client determines which the LDAP server to use.

dn
: the LDAP Distinguished Name, which identifies the base object of the LDAP search (see RFC 2253 section 3).

attributes
: a comma-separated list of attributes to be returned; see RFC 2251 section 4.1.5. If omitted, all attributes should be returned.

scope
: specifies the scope of the search, which can be one of "base" (for a base object search), "one" (for a one-level search), or "sub" (for a subtree search). If scope is omitted, "base" is assumed.

filter
: specifies the search filter (subset of entries to return). If omitted, all entries should be returned. See RFC 2254 section 4.

extensions
: a comma-separated list of type=value pairs, where the =value portion may be omitted for options not requiring it. An extension prefixed with a '!' is critical (must be supported to be valid), otherwise it is noncritical (optional).

LDAP queries are easiest to explain by example. Here's a query that asks ldap.itd.umich.edu for information about the University of Michigan in the U.S.:

ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US

To just get its postal address attribute, request:

ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAddress

To ask a host.com at port 6666 for information about the person with common name (cn) "Babs Jensen" at University of Michigan, request:

ldap://host.com:6666/o=University%20of%20Michigan,c=US??sub?(cn=Babs%20Jensen)

wais − Wide Area Information Servers

wais://*hostport*/*database*
wais://*hostport*/*database*?*search*
wais://*hostport*/*database*/*wtype*/*wpath*

This scheme designates a WAIS database, search, or document (see IETF RFC 1625 for more information on WAIS). Hostport is the hostname, optionally followed by a colon and port number (the default port number is 210).

The first form designates a WAIS database for searching. The second form designates a particular search of the WAIS database *database*. The third form designates a particular document within a WAIS database to be retrieved. *wtype* is the WAIS designation of the type of the object and *wpath* is the WAIS document-id.

other schemes

There are many other URI schemes. Most tools that accept URIs support a set of internal URIs (e.g., Mozilla has the about: scheme for internal information, and the GNOME help browser has the toc: scheme for various starting locations). There are many schemes that have been defined but are not as widely used at the current time (e.g., prospero). The nntp: scheme is deprecated in favor of the news: scheme. URNs are to be supported by the urn: scheme, with a hierarchical name space (e.g., urn:ietf:... would identify IETF documents); at this time URNs are not widely implemented. Not all tools support all schemes.

Character encoding

URIs use a limited number of characters so that they can be typed in and used in a variety of situations.

The following characters are reserved, that is, they may appear in a URI but their use is limited to their

reserved purpose (conflicting data must be escaped before forming the URI):

```
; / ? : @ & = + $ ,
```

Unreserved characters may be included in a URI. Unreserved characters include uppercase and lowercase Latin letters, decimal digits, and the following limited set of punctuation marks and symbols:

```
- _ . ! ~ * ' ( )
```

All other characters must be escaped. An escaped octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing the octet code (you can use uppercase or lowercase letters for the hexadecimal digits). For example, a blank space must be escaped as "%20", a tab character as "%09", and the "&" as "%26". Because the percent "%" character always has the reserved purpose of being the escape indicator, it must be escaped as "%25". It is common practice to escape space characters as the plus symbol (+) in query text; this practice isn't uniformly defined in the relevant RFCs (which recommend %20 instead) but any tool accepting URIs with query text should be prepared for them. A URI is always shown in its "escaped" form.

Unreserved characters can be escaped without changing the semantics of the URI, but this should not be done unless the URI is being used in a context that does not allow the unescaped character to appear. For example, "%7e" is sometimes used instead of "~" in an HTTP URL path, but the two are equivalent for an HTTP URL.

For URIs which must handle characters outside the US ASCII character set, the HTML 4.01 specification (section B.2) and IETF RFC 3986 (last paragraph of section 2.5) recommend the following approach:

(1) translate the character sequences into UTF-8 (IETF RFC 3629)—see *utf-8*(7)—and then

(2) use the URI escaping mechanism, that is, use the %HH encoding for unsafe octets.

Writing a URI

When written, URIs should be placed inside double quotes (e.g., "http://www.kernel.org"), enclosed in angle brackets (e.g., <http://lwn.net>), or placed on a line by themselves. A warning for those who use double-quotes: **never** move extraneous punctuation (such as the period ending a sentence or the comma in a list) inside a URI, since this will change the value of the URI. Instead, use angle brackets instead, or switch to a quoting system that never includes extraneous characters inside quotation marks. This latter system, called the 'new' or 'logical' quoting system by "Hart's Rules" and the "Oxford Dictionary for Writers and Editors", is preferred practice in Great Britain and in various European languages. Older documents suggested inserting the prefix "URL:" just before the URI, but this form has never caught on.

The URI syntax was designed to be unambiguous. However, as URIs have become commonplace, traditional media (television, radio, newspapers, billboards, etc.) have increasingly used abbreviated URI references consisting of only the authority and path portions of the identified resource (e.g., <www.w3.org/Addressing>). Such references are primarily intended for human interpretation rather than machine, with the assumption that context-based heuristics are sufficient to complete the URI (e.g., hostnames beginning with "www" are likely to have a URI prefix of "http://" and hostnames beginning with "ftp" likely to have a prefix of "ftp://"). Many client implementations heuristically resolve these references. Such heuristics may change over time, particularly when new schemes are introduced. Since an abbreviated URI has the same syntax as a relative URL path, abbreviated URI references cannot be used where relative URIs are permitted, and can be used only when there is no defined base (such as in dialog boxes). Don't use abbreviated URIs as hypertext links inside a document; use the standard format as described here.

STANDARDS

(IETF RFC 2396), (HTML 4.0).

NOTES

Any tool accepting URIs (e.g., a web browser) on a Linux system should be able to handle (directly or indirectly) all of the schemes described here, including the man: and info: schemes. Handling them by invoking some other program is fine and in fact encouraged.

Technically the fragment isn't part of the URI.

For information on how to embed URIs (including URLs) in a data format, see documentation on that format. HTML uses the format <A HREF="*uri*"> *text* </A>. Texinfo files use the format @uref{*uri*}.

Man and mdoc have the recently added UR macro, or just include the URI in the text (viewers should be able to detect :// as part of a URI).

The GNOME and KDE desktop environments currently vary in the URIs they accept, in particular in their respective help browsers. To list man pages, GNOME uses <toc:man> while KDE uses <man:(index)>, and to list info pages, GNOME uses <toc:info> while KDE uses <info:(dir)> (the author of this man page prefers the KDE approach here, though a more regular format would be even better). In general, KDE uses <file:/cgi-bin/> as a prefix to a set of generated files. KDE prefers documentation in HTML, accessed via the <file:/cgi-bin/helpindex>. GNOME prefers the ghelp scheme to store and find documentation. Neither browser handles file: references to directories at the time of this writing, making it difficult to refer to an entire directory with a browsable URI. As noted above, these environments differ in how they handle the info: scheme, probably the most important variation. It is expected that GNOME and KDE will converge to common URI formats, and a future version of this man page will describe the converged result. Efforts to aid this convergence are encouraged.

Security

A URI does not in itself pose a security threat. There is no general guarantee that a URL, which at one time located a given resource, will continue to do so. Nor is there any guarantee that a URL will not locate a different resource at some later point in time; such a guarantee can be obtained only from the person(s) controlling that namespace and the resource in question.

It is sometimes possible to construct a URL such that an attempt to perform a seemingly harmless operation, such as the retrieval of an entity associated with the resource, will in fact cause a possibly damaging remote operation to occur. The unsafe URL is typically constructed by specifying a port number other than that reserved for the network protocol in question. The client unwittingly contacts a site that is in fact running a different protocol. The content of the URL contains instructions that, when interpreted according to this other protocol, cause an unexpected operation. An example has been the use of a gopher URL to cause an unintended or impersonating message to be sent via a SMTP server.

Caution should be used when using any URL that specifies a port number other than the default for the protocol, especially when it is a number within the reserved space.

Care should be taken when a URI contains escaped delimiters for a given protocol (for example, CR and LF characters for telnet protocols) that these are not unescaped before transmission. This might violate the protocol, but avoids the potential for such characters to be used to simulate an extra operation or parameter in that protocol, which might lead to an unexpected and possibly harmful remote operation to be performed.

It is clearly unwise to use a URI that contains a password which is intended to be secret. In particular, the use of a password within the "userinfo" component of a URI is strongly recommended against except in those rare cases where the "password" parameter is intended to be public.

BUGS

Documentation may be placed in a variety of locations, so there currently isn't a good URI scheme for general online documentation in arbitrary formats. References of the form <file:///usr/doc/ZZZ> don't work because different distributions and local installation requirements may place the files in different directories (it may be in /usr/doc, or /usr/local/doc, or /usr/share, or somewhere else). Also, the directory ZZZ usually changes when a version changes (though filename globbing could partially overcome this). Finally, using the file: scheme doesn't easily support people who dynamically load documentation from the Internet (instead of loading the files onto a local filesystem). A future URI scheme may be added (e.g., "userdoc:") to permit programs to include cross-references to more detailed documentation without having to know the exact location of that documentation. Alternatively, a future version of the filesystem specification may specify file locations sufficiently so that the file: scheme will be able to locate documentation.

Many programs and file formats don't include a way to incorporate or implement links using URIs.

Many programs can't handle all of these different URI formats; there should be a standard mechanism to load an arbitrary URI that automatically detects the users' environment (e.g., text or graphics, desktop environment, local user preferences, and currently executing tools) and invokes the right tool for any URI.

SEE ALSO

lynx(1), *man2html*(1), *mailaddr*(7), *utf-8*(7)

IETF RFC 2255

NAME

user-keyring − per-user keyring

DESCRIPTION

The user keyring is a keyring used to anchor keys on behalf of a user. Each UID the kernel deals with has its own user keyring that is shared by all processes with that UID. The user keyring has a name (description) of the form *_uid.*<*UID*> where <*UID*> is the user ID of the corresponding user.

The user keyring is associated with the record that the kernel maintains for the UID. It comes into existence upon the first attempt to access either the user keyring, the *user-session-keyring*(7), or the *session-keyring*(7). The keyring remains pinned in existence so long as there are processes running with that real UID or files opened by those processes remain open. (The keyring can also be pinned indefinitely by linking it into another keyring.)

Typically, the user keyring is created by *pam_keyinit*(8) when a user logs in.

The user keyring is not searched by default by *request_key*(2). When *pam_keyinit*(8) creates a session keyring, it adds to it a link to the user keyring so that the user keyring will be searched when the session keyring is.

A special serial number value, **KEY_SPEC_USER_KEYRING**, is defined that can be used in lieu of the actual serial number of the calling process's user keyring.

From the *keyctl*(1) utility, '**@u**' can be used instead of a numeric key ID in much the same way.

User keyrings are independent of *clone*(2), *fork*(2), *vfork*(2), *execve*(2), and *_exit*(2) excepting that the keyring is destroyed when the UID record is destroyed when the last process pinning it exits.

If it is necessary for a key associated with a user to exist beyond the UID record being garbage collected—for example, for use by a *cron*(8) script—then the *persistent-keyring*(7) should be used instead.

If a user keyring does not exist when it is accessed, it will be created.

SEE ALSO

keyctl(1), *keyctl*(3), *keyrings*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-session-keyring*(7), *pam_keyinit*(8)

NAME

user-session-keyring − per-user default session keyring

DESCRIPTION

The user session keyring is a keyring used to anchor keys on behalf of a user. Each UID the kernel deals with has its own user session keyring that is shared by all processes with that UID. The user session keyring has a name (description) of the form *_uid_ses.<UID>* where *<UID>* is the user ID of the corresponding user.

The user session keyring is associated with the record that the kernel maintains for the UID. It comes into existence upon the first attempt to access either the user session keyring, the *user-keyring*(7), or the *session-keyring*(7). The keyring remains pinned in existence so long as there are processes running with that real UID or files opened by those processes remain open. (The keyring can also be pinned indefinitely by linking it into another keyring.)

The user session keyring is created on demand when a thread requests it or when a thread asks for its *session-keyring*(7) and that keyring doesn't exist. In the latter case, a user session keyring will be created and, if the session keyring wasn't to be created, the user session keyring will be set as the process's actual session keyring.

The user session keyring is searched by *request_key*(2) if the actual session keyring does not exist and is ignored otherwise.

A special serial number value, **KEY_SPEC_USER_SESSION_KEYRING**, is defined that can be used in lieu of the actual serial number of the calling process's user session keyring.

From the *keyctl*(1) utility, '**@us**' can be used instead of a numeric key ID in much the same way.

User session keyrings are independent of *clone*(2), *fork*(2), *vfork*(2), *execve*(2), and *_exit*(2) excepting that the keyring is destroyed when the UID record is destroyed when the last process pinning it exits.

If a user session keyring does not exist when it is accessed, it will be created.

Rather than relying on the user session keyring, it is strongly recommended—especially if the process is running as root—that a *session-keyring*(7) be set explicitly, for example by *pam_keyinit*(8)

NOTES

The user session keyring was added to support situations where a process doesn't have a session keyring, perhaps because it was created via a pathway that didn't involve PAM (e.g., perhaps it was a daemon started by *inetd*(8)). In such a scenario, the user session keyring acts as a substitute for the *session-keyring*(7).

SEE ALSO

keyctl(1), *keyctl*(3), *keyrings*(7), *persistent-keyring*(7), *process-keyring*(7), *session-keyring*(7), *thread-keyring*(7), *user-keyring*(7)

NAME

user_namespaces − overview of Linux user namespaces

DESCRIPTION

For an overview of namespaces, see *namespaces*(7).

User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs (see *credentials*(7)), the root directory, keys (see *keyrings*(7)), and capabilities (see *capabilities*(7)). A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

Nested namespaces, namespace membership

User namespaces can be nested; that is, each user namespace—except the initial ("root") namespace—has a parent user namespace, and can have zero or more child user namespaces. The parent user namespace is the user namespace of the process that creates the user namespace via a call to *unshare*(2) or *clone*(2) with the **CLONE_NEWUSER** flag.

The kernel imposes (since Linux 3.11) a limit of 32 nested levels of user namespaces. Calls to *unshare*(2) or *clone*(2) that would cause this limit to be exceeded fail with the error **EUSERS**.

Each process is a member of exactly one user namespace. A process created via *fork*(2) or *clone*(2) without the **CLONE_NEWUSER** flag is a member of the same user namespace as its parent. A single-threaded process can join another user namespace with *setns*(2) if it has the **CAP_SYS_ADMIN** in that namespace; upon doing so, it gains a full set of capabilities in that namespace.

A call to *clone*(2) or *unshare*(2) with the **CLONE_NEWUSER** flag makes the new child process (for *clone*(2)) or the caller (for *unshare*(2)) a member of the new user namespace created by the call.

The **NS_GET_PARENT** *ioctl*(2) operation can be used to discover the parental relationship between user namespaces; see *ioctl_ns*(2).

A task that changes one of its effective IDs will have its dumpability reset to the value in */proc/sys/fs/suid_dumpable*. This may affect the ownership of proc files of child processes and may thus cause the parent to lack the permissions to write to mapping files of child processes running in a new user namespace. In such cases making the parent process dumpable, using **PR_SET_DUMPABLE** in a call to *prctl*(2), before creating a child process in a new user namespace may rectify this problem. See *prctl*(2) and *proc*(5) for details on how ownership is affected.

Capabilities

The child process created by *clone*(2) with the **CLONE_NEWUSER** flag starts out with a complete set of capabilities in the new user namespace. Likewise, a process that creates a new user namespace using *unshare*(2) or joins an existing user namespace using *setns*(2) gains a full set of capabilities in that namespace. On the other hand, that process has no capabilities in the parent (in the case of *clone*(2)) or previous (in the case of *unshare*(2) and *setns*(2)) user namespace, even if the new namespace is created or joined by the root user (i.e., a process with user ID 0 in the root namespace).

Note that a call to *execve*(2) will cause a process's capabilities to be recalculated in the usual way (see *capabilities*(7)). Consequently, unless the process has a user ID of 0 within the namespace, or the executable file has a nonempty inheritable capabilities mask, the process will lose all capabilities. See the discussion of user and group ID mappings, below.

A call to *clone*(2) or *unshare*(2) using the **CLONE_NEWUSER** flag or a call to *setns*(2) that moves the caller into another user namespace sets the "securebits" flags (see *capabilities*(7)) to their default values (all flags disabled) in the child (for *clone*(2)) or caller (for *unshare*(2) or *setns*(2)). Note that because the caller no longer has capabilities in its original user namespace after a call to *setns*(2), it is not possible for a process to reset its "securebits" flags while retaining its user namespace membership by using a pair of *setns*(2) calls to move to another user namespace and then return to its original user namespace.

The rules for determining whether or not a process has a capability in a particular user namespace are as follows:

- A process has a capability inside a user namespace if it is a member of that namespace and it has the capability in its effective capability set. A process can gain capabilities in its effective capability set in various ways. For example, it may execute a set-user-ID program or an executable with associated file capabilities. In addition, a process may gain capabilities via the effect of *clone*(2), *unshare*(2), or *setns*(2), as already described.
- If a process has a capability in a user namespace, then it has that capability in all child (and further removed descendant) namespaces as well.
- When a user namespace is created, the kernel records the effective user ID of the creating process as being the "owner" of the namespace. A process that resides in the parent of the user namespace and whose effective user ID matches the owner of the namespace has all capabilities in the namespace. By virtue of the previous rule, this means that the process has all capabilities in all further removed descendant user namespaces as well. The **NS_GET_OWNER_UID** *ioctl*(2) operation can be used to discover the user ID of the owner of the namespace; see *ioctl_ns*(2).

Effect of capabilities within a user namespace

Having a capability inside a user namespace permits a process to perform operations (that require privilege) only on resources governed by that namespace. In other words, having a capability in a user namespace permits a process to perform privileged operations on resources that are governed by (nonuser) namespaces owned by (associated with) the user namespace (see the next subsection).

On the other hand, there are many privileged operations that affect resources that are not associated with any namespace type, for example, changing the system (i.e., calendar) time (governed by **CAP_SYS_TIME**), loading a kernel module (governed by **CAP_SYS_MODULE**), and creating a device (governed by **CAP_MKNOD**). Only a process with privileges in the *initial* user namespace can perform such operations.

Holding **CAP_SYS_ADMIN** within the user namespace that owns a process's mount namespace allows that process to create bind mounts and mount the following types of filesystems:

- */proc* (since Linux 3.8)
- */sys* (since Linux 3.8)
- *devpts* (since Linux 3.9)
- *tmpfs*(5) (since Linux 3.9)
- *ramfs* (since Linux 3.9)
- *mqueue* (since Linux 3.9)
- *bpf* (since Linux 4.4)
- *overlayfs* (since Linux 5.11)

Holding **CAP_SYS_ADMIN** within the user namespace that owns a process's cgroup namespace allows (since Linux 4.6) that process to the mount the cgroup version 2 filesystem and cgroup version 1 named hierarchies (i.e., cgroup filesystems mounted with the *"none,name="* option).

Holding **CAP_SYS_ADMIN** within the user namespace that owns a process's PID namespace allows (since Linux 3.8) that process to mount */proc* filesystems.

Note, however, that mounting block-based filesystems can be done only by a process that holds **CAP_SYS_ADMIN** in the initial user namespace.

Interaction of user namespaces and other types of namespaces

Starting in Linux 3.8, unprivileged processes can create user namespaces, and the other types of namespaces can be created with just the **CAP_SYS_ADMIN** capability in the caller's user namespace.

When a nonuser namespace is created, it is owned by the user namespace in which the creating process was a member at the time of the creation of the namespace. Privileged operations on resources governed by the nonuser namespace require that the process has the necessary capabilities in the user namespace that owns the nonuser namespace.

If **CLONE_NEWUSER** is specified along with other **CLONE_NEW*** flags in a single *clone*(2) or *unshare*(2) call, the user namespace is guaranteed to be created first, giving the child (**clone**(2)) or caller (**unshare**(2)) privileges over the remaining namespaces created by the call. Thus, it is possible for an unprivileged caller to specify this combination of flags.

When a new namespace (other than a user namespace) is created via *clone*(2) or *unshare*(2), the kernel records the user namespace of the creating process as the owner of the new namespace. (This

association can't be changed.) When a process in the new namespace subsequently performs privileged operations that operate on global resources isolated by the namespace, the permission checks are performed according to the process's capabilities in the user namespace that the kernel associated with the new namespace. For example, suppose that a process attempts to change the hostname (**sethostname**(2)), a resource governed by the UTS namespace. In this case, the kernel will determine which user namespace owns the process's UTS namespace, and check whether the process has the required capability (**CAP_SYS_ADMIN**) in that user namespace.

The **NS_GET_USERNS** *ioctl*(2) operation can be used to discover the user namespace that owns a nonuser namespace; see *ioctl_ns*(2).

User and group ID mappings: uid_map and gid_map

When a user namespace is created, it starts out without a mapping of user IDs (group IDs) to the parent user namespace. The */proc/*pid*/uid_map* and */proc/*pid*/gid_map* files (available since Linux 3.5) expose the mappings for user and group IDs inside the user namespace for the process *pid*. These files can be read to view the mappings in a user namespace and written to (once) to define the mappings.

The description in the following paragraphs explains the details for *uid_map*; *gid_map* is exactly the same, but each instance of "user ID" is replaced by "group ID".

The *uid_map* file exposes the mapping of user IDs from the user namespace of the process *pid* to the user namespace of the process that opened *uid_map* (but see a qualification to this point below). In other words, processes that are in different user namespaces will potentially see different values when reading from a particular *uid_map* file, depending on the user ID mappings for the user namespaces of the reading processes.

Each line in the *uid_map* file specifies a 1-to-1 mapping of a range of contiguous user IDs between two user namespaces. (When a user namespace is first created, this file is empty.) The specification in each line takes the form of three numbers delimited by white space. The first two numbers specify the starting user ID in each of the two user namespaces. The third number specifies the length of the mapped range. In detail, the fields are interpreted as follows:

(1) The start of the range of user IDs in the user namespace of the process *pid*.

(2) The start of the range of user IDs to which the user IDs specified by field one map. How field two is interpreted depends on whether the process that opened *uid_map* and the process *pid* are in the same user namespace, as follows:

  (a) If the two processes are in different user namespaces: field two is the start of a range of user IDs in the user namespace of the process that opened *uid_map*.

  (b) If the two processes are in the same user namespace: field two is the start of the range of user IDs in the parent user namespace of the process *pid*. This case enables the opener of *uid_map* (the common case here is opening */proc/self/uid_map*) to see the mapping of user IDs into the user namespace of the process that created this user namespace.

(3) The length of the range of user IDs that is mapped between the two user namespaces.

System calls that return user IDs (group IDs)—for example, *getuid*(2), *getgid*(2), and the credential fields in the structure returned by *stat*(2)—return the user ID (group ID) mapped into the caller's user namespace.

When a process accesses a file, its user and group IDs are mapped into the initial user namespace for the purpose of permission checking and assigning IDs when creating a file. When a process retrieves file user and group IDs via *stat*(2), the IDs are mapped in the opposite direction, to produce values relative to the process user and group ID mappings.

The initial user namespace has no parent namespace, but, for consistency, the kernel provides dummy user and group ID mapping files for this namespace. Looking at the *uid_map* file (*gid_map* is the same) from a shell in the initial namespace shows:

```
$ cat /proc/$$/uid_map
         0          0 4294967295
```

This mapping tells us that the range starting at user ID 0 in this namespace maps to a range starting at 0 in the (nonexistent) parent namespace, and the length of the range is the largest 32-bit unsigned integer. This leaves 4294967295 (the 32-bit signed −1 value) unmapped. This is deliberate: *(uid_t) −1* is used in several interfaces (e.g., *setreuid*(2)) as a way to specify "no user ID". Leaving *(uid_t) −1* unmapped

and unusable guarantees that there will be no confusion when using these interfaces.

Defining user and group ID mappings: writing to uid_map and gid_map

After the creation of a new user namespace, the *uid_map* file of *one* of the processes in the namespace may be written to *once* to define the mapping of user IDs in the new user namespace. An attempt to write more than once to a *uid_map* file in a user namespace fails with the error **EPERM**. Similar rules apply for *gid_map* files.

The lines written to *uid_map* (*gid_map*) must conform to the following validity rules:

- The three fields must be valid numbers, and the last field must be greater than 0.
- Lines are terminated by newline characters.
- There is a limit on the number of lines in the file. In Linux 4.14 and earlier, this limit was (arbitrarily) set at 5 lines. Since Linux 4.15, the limit is 340 lines. In addition, the number of bytes written to the file must be less than the system page size, and the write must be performed at the start of the file (i.e., *lseek*(2) and *pwrite*(2) can't be used to write to nonzero offsets in the file).
- The range of user IDs (group IDs) specified in each line cannot overlap with the ranges in any other lines. In the initial implementation (Linux 3.8), this requirement was satisfied by a simplistic implementation that imposed the further requirement that the values in both field 1 and field 2 of successive lines must be in ascending numerical order, which prevented some otherwise valid maps from being created. Linux 3.9 and later fix this limitation, allowing any valid set of nonoverlapping maps.
- At least one line must be written to the file.

Writes that violate the above rules fail with the error **EINVAL**.

In order for a process to write to the */proc/*pid*/uid_map* (*/proc/*pid*/gid_map*) file, all of the following permission requirements must be met:

- The writing process must have the **CAP_SETUID** (**CAP_SETGID**) capability in the user namespace of the process *pid*.
- The writing process must either be in the user namespace of the process *pid* or be in the parent user namespace of the process *pid*.
- The mapped user IDs (group IDs) must in turn have a mapping in the parent user namespace.
- If updating */proc/*pid*/uid_map* to create a mapping that maps UID 0 in the parent namespace, then one of the following must be true:

  (a) if writing process is in the parent user namespace, then it must have the **CAP_SETFCAP** capability in that user namespace; or

  (b) if the writing process is in the child user namespace, then the process that created the user namespace must have had the **CAP_SETFCAP** capability when the namespace was created.

  This rule has been in place since Linux 5.12. It eliminates an earlier security bug whereby a UID 0 process that lacks the **CAP_SETFCAP** capability, which is needed to create a binary with namespaced file capabilities (as described in *capabilities*(7)), could nevertheless create such a binary, by the following steps:

  (1) Create a new user namespace with the identity mapping (i.e., UID 0 in the new user namespace maps to UID 0 in the parent namespace), so that UID 0 in both namespaces is equivalent to the same root user ID.

  (2) Since the child process has the **CAP_SETFCAP** capability, it could create a binary with namespaced file capabilities that would then be effective in the parent user namespace (because the root user IDs are the same in the two namespaces).
- One of the following two cases applies:

  (a) *Either* the writing process has the **CAP_SETUID** (**CAP_SETGID**) capability in the *parent* user namespace.

  - No further restrictions apply: the process can make mappings to arbitrary user IDs (group IDs) in the parent user namespace.

(b) *Or* otherwise all of the following restrictions apply:

- The data written to *uid_map* (*gid_map*) must consist of a single line that maps the writing process's effective user ID (group ID) in the parent user namespace to a user ID (group ID) in the user namespace.
- The writing process must have the same effective user ID as the process that created the user namespace.
- In the case of *gid_map*, use of the *setgroups*(2) system call must first be denied by writing "*deny*" to the */proc/*pid*/setgroups* file (see below) before writing to *gid_map*.

Writes that violate the above rules fail with the error **EPERM**.

Project ID mappings: projid_map

Similarly to user and group ID mappings, it is possible to create project ID mappings for a user namespace. (Project IDs are used for disk quotas; see *setquota*(8) and *quotactl*(2).)

Project ID mappings are defined by writing to the */proc/*pid*/projid_map* file (present since Linux 3.7).

The validity rules for writing to the */proc/*pid*/projid_map* file are as for writing to the *uid_map* file; violation of these rules causes *write*(2) to fail with the error **EINVAL**.

The permission rules for writing to the */proc/*pid*/projid_map* file are as follows:

- The writing process must either be in the user namespace of the process *pid* or be in the parent user namespace of the process *pid*.
- The mapped project IDs must in turn have a mapping in the parent user namespace.

Violation of these rules causes *write*(2) to fail with the error **EPERM**.

Interaction with system calls that change process UIDs or GIDs

In a user namespace where the *uid_map* file has not been written, the system calls that change user IDs will fail. Similarly, if the *gid_map* file has not been written, the system calls that change group IDs will fail. After the *uid_map* and *gid_map* files have been written, only the mapped values may be used in system calls that change user and group IDs.

For user IDs, the relevant system calls include *setuid*(2), *setfsuid*(2), *setreuid*(2), and *setresuid*(2). For group IDs, the relevant system calls include *setgid*(2), *setfsgid*(2), *setregid*(2), *setresgid*(2), and *setgroups*(2).

Writing "*deny*" to the */proc/*pid*/setgroups* file before writing to */proc/*pid*/gid_map* will permanently disable *setgroups*(2) in a user namespace and allow writing to */proc/*pid*/gid_map* without having the **CAP_SETGID** capability in the parent user namespace.

The */proc/*pid*/setgroups* file

The */proc/*pid*/setgroups* file displays the string "*allow*" if processes in the user namespace that contains the process *pid* are permitted to employ the *setgroups*(2) system call; it displays "*deny*" if *setgroups*(2) is not permitted in that user namespace. Note that regardless of the value in the */proc/*pid*/setgroups* file (and regardless of the process's capabilities), calls to *setgroups*(2) are also not permitted if */proc/*pid*/gid_map* has not yet been set.

A privileged process (one with the **CAP_SYS_ADMIN** capability in the namespace) may write either of the strings "*allow*" or "*deny*" to this file *before* writing a group ID mapping for this user namespace to the file */proc/*pid*/gid_map*. Writing the string "*deny*" prevents any process in the user namespace from employing *setgroups*(2).

The essence of the restrictions described in the preceding paragraph is that it is permitted to write to */proc/*pid*/setgroups* only so long as calling *setgroups*(2) is disallowed because */proc/*pid*/gid_map* has not been set. This ensures that a process cannot transition from a state where *setgroups*(2) is allowed to a state where *setgroups*(2) is denied; a process can transition only from *setgroups*(2) being disallowed to *setgroups*(2) being allowed.

The default value of this file in the initial user namespace is "*allow*".

Once */proc/*pid*/gid_map* has been written to (which has the effect of enabling *setgroups*(2) in the user namespace), it is no longer possible to disallow *setgroups*(2) by writing "*deny*" to */proc/*pid*/setgroups* (the write fails with the error **EPERM**).

A child user namespace inherits the */proc/*pid*/setgroups* setting from its parent.

If the *setgroups* file has the value "*deny*", then the *setgroups*(2) system call can't subsequently be reenabled (by writing "*allow*" to the file) in this user namespace. (Attempts to do so fail with the error **EPERM**.) This restriction also propagates down to all child user namespaces of this user namespace.

The */proc/*pid*/setgroups* file was added in Linux 3.19, but was backported to many earlier stable kernel series, because it addresses a security issue. The issue concerned files with permissions such as "rwx−−−rwx". Such files give fewer permissions to "group" than they do to "other". This means that dropping groups using *setgroups*(2) might allow a process file access that it did not formerly have. Before the existence of user namespaces this was not a concern, since only a privileged process (one with the **CAP_SETGID** capability) could call *setgroups*(2). However, with the introduction of user namespaces, it became possible for an unprivileged process to create a new namespace in which the user had all privileges. This then allowed formerly unprivileged users to drop groups and thus gain file access that they did not previously have. The */proc/*pid*/setgroups* file was added to address this security issue, by denying any pathway for an unprivileged process to drop groups with *setgroups*(2).

Unmapped user and group IDs

There are various places where an unmapped user ID (group ID) may be exposed to user space. For example, the first process in a new user namespace may call *getuid*(2) before a user ID mapping has been defined for the namespace. In most such cases, an unmapped user ID is converted to the overflow user ID (group ID); the default value for the overflow user ID (group ID) is 65534. See the descriptions of */proc/sys/kernel/overflowuid* and */proc/sys/kernel/overflowgid* in *proc*(5).

The cases where unmapped IDs are mapped in this fashion include system calls that return user IDs (**getuid**(2), *getgid*(2), and similar), credentials passed over a UNIX domain socket, credentials returned by *stat*(2), *waitid*(2), and the System V IPC "ctl" **IPC_STAT** operations, credentials exposed by */proc/*pid*/status* and the files in */proc/sysvipc/**, credentials returned via the *si_uid* field in the *siginfo_t* received with a signal (see *sigaction*(2)), credentials written to the process accounting file (see *acct*(5)), and credentials returned with POSIX message queue notifications (see *mq_notify*(3)).

There is one notable case where unmapped user and group IDs are *not* converted to the corresponding overflow ID value. When viewing a *uid_map* or *gid_map* file in which there is no mapping for the second field, that field is displayed as 4294967295 (−1 as an unsigned integer).

Accessing files

In order to determine permissions when an unprivileged process accesses a file, the process credentials (UID, GID) and the file credentials are in effect mapped back to what they would be in the initial user namespace and then compared to determine the permissions that the process has on the file. The same is also true of other objects that employ the credentials plus permissions mask accessibility model, such as System V IPC objects.

Operation of file-related capabilities

Certain capabilities allow a process to bypass various kernel-enforced restrictions when performing operations on files owned by other users or groups. These capabilities are: **CAP_CHOWN**, **CAP_DAC_OVERRIDE**, **CAP_DAC_READ_SEARCH**, **CAP_FOWNER**, and **CAP_FSETID**.

Within a user namespace, these capabilities allow a process to bypass the rules if the process has the relevant capability over the file, meaning that:

- the process has the relevant effective capability in its user namespace; and
- the file's user ID and group ID both have valid mappings in the user namespace.

The **CAP_FOWNER** capability is treated somewhat exceptionally: it allows a process to bypass the corresponding rules so long as at least the file's user ID has a mapping in the user namespace (i.e., the file's group ID does not need to have a valid mapping).

Set-user-ID and set-group-ID programs

When a process inside a user namespace executes a set-user-ID (set-group-ID) program, the process's effective user (group) ID inside the namespace is changed to whatever value is mapped for the user (group) ID of the file. However, if either the user *or* the group ID of the file has no mapping inside the namespace, the set-user-ID (set-group-ID) bit is silently ignored: the new program is executed, but the process's effective user (group) ID is left unchanged. (This mirrors the semantics of executing a set-user-ID or set-group-ID program that resides on a filesystem that was mounted with the **MS_NOSUID** flag, as described in *mount*(2).)

Miscellaneous

When a process's user and group IDs are passed over a UNIX domain socket to a process in a different user namespace (see the description of **SCM_CREDENTIALS** in *unix*(7)), they are translated into the corresponding values as per the receiving process's user and group ID mappings.

STANDARDS

Linux.

NOTES

Over the years, there have been a lot of features that have been added to the Linux kernel that have been made available only to privileged users because of their potential to confuse set-user-ID-root applications. In general, it becomes safe to allow the root user in a user namespace to use those features because it is impossible, while in a user namespace, to gain more privilege than the root user of a user namespace has.

Global root

The term "global root" is sometimes used as a shorthand for user ID 0 in the initial user namespace.

Availability

Use of user namespaces requires a kernel that is configured with the **CONFIG_USER_NS** option. User namespaces require support in a range of subsystems across the kernel. When an unsupported subsystem is configured into the kernel, it is not possible to configure user namespaces support.

As at Linux 3.8, most relevant subsystems supported user namespaces, but a number of filesystems did not have the infrastructure needed to map user and group IDs between user namespaces. Linux 3.9 added the required infrastructure support for many of the remaining unsupported filesystems (Plan 9 (9P), Andrew File System (AFS), Ceph, CIFS, CODA, NFS, and OCFS2). Linux 3.12 added support for the last of the unsupported major filesystems, XFS.

EXAMPLES

The program below is designed to allow experimenting with user namespaces, as well as other types of namespaces. It creates namespaces as specified by command-line options and then executes a command inside those namespaces. The comments and *usage*() function inside the program provide a full explanation of the program. The following shell session demonstrates its use.

First, we look at the run-time environment:

```
$ uname -rs     # Need Linux 3.8 or later
Linux 3.8.0
$ id -u         # Running as unprivileged user
1000
$ id -g
1000
```

Now start a new shell in new user (*-U*), mount (*-m*), and PID (*-p*) namespaces, with user ID (*-M*) and group ID (*-G*) 1000 mapped to 0 inside the user namespace:

```
$ ./userns_child_exec -p -m -U -M '0 1000 1' -G '0 1000 1' bash
```

The shell has PID 1, because it is the first process in the new PID namespace:

```
bash$ echo $$
1
```

Mounting a new */proc* filesystem and listing all of the processes visible in the new PID namespace shows that the shell can't see any processes outside the PID namespace:

```
bash$ mount -t proc proc /proc
bash$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 pts/3    S      0:00 bash
   22 pts/3    R+     0:00 ps ax
```

Inside the user namespace, the shell has user and group ID 0, and a full set of permitted and effective capabilities:

```
bash$ cat /proc/$$/status | egrep '^[UG]id'
Uid:  0     0     0     0
```

```
    Gid:  0     0     0     0
    bash$ cat /proc/$$/status | egrep '^Cap(Prm|Inh|Eff)'
    CapInh:     0000000000000000
    CapPrm:     0000001fffffffff
    CapEff:     0000001fffffffff
```

Program source

```
/* userns_child_exec.c

   Licensed under GNU General Public License v2 or later

   Create a child process that executes a shell command in new
   namespace(s); allow UID and GID mappings to be specified when
   creating a user namespace.
*/
#define _GNU_SOURCE
#include <err.h>
#include <sched.h>
#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

struct child_args {
    char **argv;        /* Command to be executed by child, with args */
    int    pipe_fd[2];  /* Pipe used to synchronize parent and child */
};

static int verbose;

static void
usage(char *pname)
{
    fprintf(stderr, "Usage: %s [options] cmd [arg...]\n\n", pname);
    fprintf(stderr, "Create a child process that executes a shell "
            "command in a new user namespace,\n"
            "and possibly also other new namespace(s).\n\n");
    fprintf(stderr, "Options can be:\n\n");
#define fpe(str) fprintf(stderr, "    %s", str);
    fpe("-i          New IPC namespace\n");
    fpe("-m          New mount namespace\n");
    fpe("-n          New network namespace\n");
    fpe("-p          New PID namespace\n");
    fpe("-u          New UTS namespace\n");
    fpe("-U          New user namespace\n");
    fpe("-M uid_map  Specify UID map for user namespace\n");
    fpe("-G gid_map  Specify GID map for user namespace\n");
    fpe("-z          Map user's UID and GID to 0 in user namespace\n");
    fpe("            (equivalent to: -M '0 <uid> 1' -G '0 <gid> 1')\n");
    fpe("-v          Display verbose messages\n");
    fpe("\n");
    fpe("If -z, -M, or -G is specified, -U is required.\n");
```

```
        fpe("It is not permitted to specify both −z and either −M or −G.\n");
        fpe("\n");
        fpe("Map strings for −M and −G consist of records of the form:\n");
        fpe("\n");
        fpe("    ID−inside−ns   ID−outside−ns   len\n");
        fpe("\n");
        fpe("A map string can contain multiple records, separated"
            " by commas;\n");
        fpe("the commas are replaced by newlines before writing"
            " to map files.\n");

        exit(EXIT_FAILURE);
    }

    /* Update the mapping file 'map_file', with the value provided in
       'mapping', a string that defines a UID or GID mapping. A UID or
       GID mapping consists of one or more newline−delimited records
       of the form:

           ID_inside−ns    ID−outside−ns   length

       Requiring the user to supply a string that contains newlines is
       of course inconvenient for command−line use. Thus, we permit the
       use of commas to delimit records in this string, and replace them
       with newlines before writing the string to the file. */

    static void
    update_map(char *mapping, char *map_file)
    {
        int fd;
        size_t map_len;     /* Length of 'mapping' */

        /* Replace commas in mapping string with newlines. */

        map_len = strlen(mapping);
        for (size_t j = 0; j < map_len; j++)
            if (mapping[j] == ',')
                mapping[j] = '\n';

        fd = open(map_file, O_RDWR);
        if (fd == −1) {
            fprintf(stderr, "ERROR: open %s: %s\n", map_file,
                    strerror(errno));
            exit(EXIT_FAILURE);
        }

        if (write(fd, mapping, map_len) != map_len) {
            fprintf(stderr, "ERROR: write %s: %s\n", map_file,
                    strerror(errno));
            exit(EXIT_FAILURE);
        }

        close(fd);
    }

    /* Linux 3.19 made a change in the handling of setgroups(2) and
       the 'gid_map' file to address a security issue.  The issue
       allowed *unprivileged* users to employ user namespaces in
       order to drop groups.  The upshot of the 3.19 changes is that
```

```
       in order to update the 'gid_maps' file, use of the setgroups()
       system call in this user namespace must first be disabled by
       writing "deny" to one of the /proc/PID/setgroups files for
       this namespace.  That is the purpose of the following function.  */

   static void
   proc_setgroups_write(pid_t child_pid, char *str)
   {
       char setgroups_path[PATH_MAX];
       int fd;

       snprintf(setgroups_path, PATH_MAX, "/proc/%jd/setgroups",
               (intmax_t) child_pid);

       fd = open(setgroups_path, O_RDWR);
       if (fd == -1) {

           /* We may be on a system that doesn't support
              /proc/PID/setgroups. In that case, the file won't exist,
              and the system won't impose the restrictions that Linux 3.19
              added. That's fine: we don't need to do anything in order
              to permit 'gid_map' to be updated.

              However, if the error from open() was something other than
              the ENOENT error that is expected for that case,  let the
              user know. */

           if (errno != ENOENT)
               fprintf(stderr, "ERROR: open %s: %s\n", setgroups_path,
                   strerror(errno));
           return;
       }

       if (write(fd, str, strlen(str)) == -1)
           fprintf(stderr, "ERROR: write %s: %s\n", setgroups_path,
               strerror(errno));

       close(fd);
   }

   static int              /* Start function for cloned child */
   childFunc(void *arg)
   {
       struct child_args *args = arg;
       char ch;

       /* Wait until the parent has updated the UID and GID mappings.
          See the comment in main(). We wait for end of file on a
          pipe that will be closed by the parent process once it has
          updated the mappings. */

       close(args->pipe_fd[1]);    /* Close our descriptor for the write
                                      end of the pipe so that we see EOF
                                      when parent closes its descriptor. */
       if (read(args->pipe_fd[0], &ch, 1) != 0) {
           fprintf(stderr,
                   "Failure in child: read from pipe returned != 0\n");
           exit(EXIT_FAILURE);
       }
```

```
    close(args->pipe_fd[0]);

    /* Execute a shell command. */

    printf("About to exec %s\n", args->argv[0]);
    execvp(args->argv[0], args->argv);
    err(EXIT_FAILURE, "execvp");
}

#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];    /* Space for child's stack */

int
main(int argc, char *argv[])
{
    int flags, opt, map_zero;
    pid_t child_pid;
    struct child_args args;
    char *uid_map, *gid_map;
    const int MAP_BUF_SIZE = 100;
    char map_buf[MAP_BUF_SIZE];
    char map_path[PATH_MAX];

    /* Parse command-line options. The initial '+' character in
       the final getopt() argument prevents GNU-style permutation
       of command-line options. That's useful, since sometimes
       the 'command' to be executed by this program itself
       has command-line options. We don't want getopt() to treat
       those as options to this program. */

    flags = 0;
    verbose = 0;
    gid_map = NULL;
    uid_map = NULL;
    map_zero = 0;
    while ((opt = getopt(argc, argv, "+imnpuUM:G:zv")) != -1) {
        switch (opt) {
        case 'i': flags |= CLONE_NEWIPC;        break;
        case 'm': flags |= CLONE_NEWNS;         break;
        case 'n': flags |= CLONE_NEWNET;        break;
        case 'p': flags |= CLONE_NEWPID;        break;
        case 'u': flags |= CLONE_NEWUTS;        break;
        case 'v': verbose = 1;                  break;
        case 'z': map_zero = 1;                 break;
        case 'M': uid_map = optarg;             break;
        case 'G': gid_map = optarg;             break;
        case 'U': flags |= CLONE_NEWUSER;       break;
        default:  usage(argv[0]);
        }
    }

    /* -M or -G without -U is nonsensical */

    if (((uid_map != NULL || gid_map != NULL || map_zero) &&
                !(flags & CLONE_NEWUSER)) ||
            (map_zero && (uid_map != NULL || gid_map != NULL)))
        usage(argv[0]);
```

```
        args.argv = &argv[optind];

        /* We use a pipe to synchronize the parent and child, in order to
           ensure that the parent sets the UID and GID maps before the child
           calls execve(). This ensures that the child maintains its
           capabilities during the execve() in the common case where we
           want to map the child's effective user ID to 0 in the new user
           namespace. Without this synchronization, the child would lose
           its capabilities if it performed an execve() with nonzero
           user IDs (see the capabilities(7) man page for details of the
           transformation of a process's capabilities during execve()). */

        if (pipe(args.pipe_fd) == -1)
            err(EXIT_FAILURE, "pipe");

        /* Create the child in new namespace(s). */

        child_pid = clone(childFunc, child_stack + STACK_SIZE,
                          flags | SIGCHLD, &args);
        if (child_pid == -1)
            err(EXIT_FAILURE, "clone");

        /* Parent falls through to here. */

        if (verbose)
            printf("%s: PID of child created by clone() is %jd\n",
                    argv[0], (intmax_t) child_pid);

        /* Update the UID and GID maps in the child. */

        if (uid_map != NULL || map_zero) {
            snprintf(map_path, PATH_MAX, "/proc/%jd/uid_map",
                    (intmax_t) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %jd 1",
                        (intmax_t) getuid());
                uid_map = map_buf;
            }
            update_map(uid_map, map_path);
        }

        if (gid_map != NULL || map_zero) {
            proc_setgroups_write(child_pid, "deny");

            snprintf(map_path, PATH_MAX, "/proc/%jd/gid_map",
                    (intmax_t) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %ld 1",
                        (intmax_t) getgid());
                gid_map = map_buf;
            }
            update_map(gid_map, map_path);
        }

        /* Close the write end of the pipe, to signal to the child that we
           have updated the UID and GID maps. */

        close(args.pipe_fd[1]);
```

```
        if (waitpid(child_pid, NULL, 0) == -1)      /* Wait for child */
            err(EXIT_FAILURE, "waitpid");

        if (verbose)
            printf("%s: terminating\n", argv[0]);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

newgidmap(1), *newuidmap*(1), *clone*(2), *ptrace*(2), *setns*(2), *unshare*(2), *proc*(5), *subgid*(5), *subuid*(5), *capabilities*(7), *cgroup_namespaces*(7), *credentials*(7), *namespaces*(7), *pid_namespaces*(7)

The kernel source file *Documentation/admin-guide/namespaces/resource-control.rst*.

NAME

UTF-8 − an ASCII compatible multibyte Unicode encoding

DESCRIPTION

The Unicode 3.0 character set occupies a 16-bit code space. The most obvious Unicode encoding (known as UCS-2) consists of a sequence of 16-bit words. Such strings can contain—as part of many 16-bit characters—bytes such as '\0' or '/', which have a special meaning in filenames and other C library function arguments. In addition, the majority of UNIX tools expect ASCII files and can't read 16-bit words as characters without major modifications. For these reasons, UCS-2 is not a suitable external encoding of Unicode in filenames, text files, environment variables, and so on. The ISO/IEC 10646 Universal Character Set (UCS), a superset of Unicode, occupies an even larger code space—31 bits—and the obvious UCS-4 encoding for it (a sequence of 32-bit words) has the same problems.

The UTF-8 encoding of Unicode and UCS does not have these problems and is the common way in which Unicode is used on UNIX-style operating systems.

Properties

The UTF-8 encoding has the following nice properties:

- UCS characters 0x00000000 to 0x0000007f (the classic US-ASCII characters) are encoded simply as bytes 0x00 to 0x7f (ASCII compatibility). This means that files and strings which contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.
- All UCS characters greater than 0x7f are encoded as a multibyte sequence consisting only of bytes in the range 0x80 to 0xfd, so no ASCII byte can appear as part of another character and there are no problems with, for example, '\0' or '/'.
- The lexicographic sorting order of UCS-4 strings is preserved.
- All possible 2^31 UCS codes can be encoded using UTF-8.
- The bytes 0xc0, 0xc1, 0xfe, and 0xff are never used in the UTF-8 encoding.
- The first byte of a multibyte sequence which represents a single non-ASCII UCS character is always in the range 0xc2 to 0xfd and indicates how long this multibyte sequence is. All further bytes in a multibyte sequence are in the range 0x80 to 0xbf. This allows easy resynchronization and makes the encoding stateless and robust against missing bytes.
- UTF-8 encoded UCS characters may be up to six bytes long, however the Unicode standard specifies no characters above 0x10ffff, so Unicode characters can be only up to four bytes long in UTF-8.

Encoding

The following byte sequences are used to represent a character. The sequence to be used depends on the UCS code number of the character:

0x00000000 − 0x0000007F:
0*xxxxxxx*

0x00000080 − 0x000007FF:
110*xxxxx* 10*xxxxxx*

0x00000800 − 0x0000FFFF:
1110*xxxx* 10*xxxxxx* 10*xxxxxx*

0x00010000 − 0x001FFFFF:
11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*

0x00200000 − 0x03FFFFFF:
111110*xx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*

0x04000000 − 0x7FFFFFFF:
1111110*x* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*

The *xxx* bit positions are filled with the bits of the character code number in binary representation, most significant bit first (big-endian). Only the shortest possible multibyte sequence which can represent the code number of the character can be used.

The UCS code values 0xd800–0xdfff (UTF-16 surrogates) as well as 0xfffe and 0xffff (UCS

noncharacters) should not appear in conforming UTF-8 streams. According to RFC 3629 no point above U+10FFFF should be used, which limits characters to four bytes.

Example

The Unicode character 0xa9 = 1010 1001 (the copyright sign) is encoded in UTF-8 as

11000010 10101001 = 0xc2 0xa9

and character 0x2260 = 0010 0010 0110 0000 (the "not equal" symbol) is encoded as:

11100010 10001001 10100000 = 0xe2 0x89 0xa0

Application notes

Users have to select a UTF-8 locale, for example with

export LANG=en_GB.UTF-8

in order to activate the UTF-8 support in applications.

Application software that has to be aware of the used character encoding should always set the locale with for example

setlocale(LC_CTYPE, "")

and programmers can then test the expression

strcmp(nl_langinfo(CODESET), "UTF-8") == 0

to determine whether a UTF-8 locale has been selected and whether therefore all plaintext standard input and output, terminal communication, plaintext file content, filenames, and environment variables are encoded in UTF-8.

Programmers accustomed to single-byte encodings such as US-ASCII or ISO/IEC 8859 have to be aware that two assumptions made so far are no longer valid in UTF-8 locales. Firstly, a single byte does not necessarily correspond any more to a single character. Secondly, since modern terminal emulators in UTF-8 mode also support Chinese, Japanese, and Korean double-width characters as well as nonspacing combining characters, outputting a single character does not necessarily advance the cursor by one position as it did in ASCII. Library functions such as *mbsrtowcs*(3) and *wcswidth*(3) should be used today to count characters and cursor positions.

The official ESC sequence to switch from an ISO/IEC 2022 encoding scheme (as used for instance by VT100 terminals) to UTF-8 is ESC % G ("\x1b%G"). The corresponding return sequence from UTF-8 to ISO/IEC 2022 is ESC % @ ("\x1b%@"). Other ISO/IEC 2022 sequences (such as for switching the G0 and G1 sets) are not applicable in UTF-8 mode.

Security

The Unicode and UCS standards require that producers of UTF-8 shall use the shortest form possible, for example, producing a two-byte sequence with first byte 0xc0 is nonconforming. Unicode 3.1 has added the requirement that conforming programs must not accept non-shortest forms in their input. This is for security reasons: if user input is checked for possible security violations, a program might check only for the ASCII version of "/../" or ";" or NUL and overlook that there are many non-ASCII ways to represent these things in a non-shortest UTF-8 encoding.

Standards

ISO/IEC 10646-1:2000, Unicode 3.1, RFC 3629, Plan 9.

SEE ALSO

locale(1), *nl_langinfo*(3), *setlocale*(3), *charsets*(7), *unicode*(7)

NAME

uts_namespaces − overview of Linux UTS namespaces

DESCRIPTION

UTS namespaces provide isolation of two system identifiers: the hostname and the NIS domain name. These identifiers are set using *sethostname*(2) and *setdomainname*(2), and can be retrieved using *uname*(2), *gethostname*(2), and *getdomainname*(2). Changes made to these identifiers are visible to all other processes in the same UTS namespace, but are not visible to processes in other UTS namespaces.

When a process creates a new UTS namespace using *clone*(2) or *unshare*(2) with the **CLONE_NEWUTS** flag, the hostname and domain name of the new UTS namespace are copied from the corresponding values in the caller's UTS namespace.

Use of UTS namespaces requires a kernel that is configured with the **CONFIG_UTS_NS** option.

SEE ALSO

nsenter(1), *unshare*(1), *clone*(2), *getdomainname*(2), *gethostname*(2), *setns*(2), *uname*(2), *unshare*(2), *namespaces*(7)

NAME

vdso − overview of the virtual ELF dynamic shared object

SYNOPSIS

#include <sys/auxv.h>

void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);

DESCRIPTION

The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.

Why does the vDSO exist at all? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. This is due both to the frequency of the call as well as the context-switch overhead that results from exiting user space and entering the kernel.

The rest of this documentation is geared toward the curious and/or C library writers rather than general developers. If you're trying to call the vDSO in your own application rather than using the C library, you're most likely doing it wrong.

Example background

Making system calls can be slow. In x86 32-bit systems, you can trigger a software interrupt (*int $0x80*) to tell the kernel you wish to make a system call. However, this instruction is expensive: it goes through the full interrupt-handling paths in the processor's microcode as well as in the kernel. Newer processors have faster (but backward incompatible) instructions to initiate system calls. Rather than require the C library to figure out if this functionality is available at run time, the C library can use functions provided by the kernel in the vDSO.

Note that the terminology can be confusing. On x86 systems, the vDSO function used to determine the preferred method of making a system call is named "__kernel_vsyscall", but on x86-64, the term "vsyscall" also refers to an obsolete way to ask the kernel what time it is or what CPU the caller is on.

One frequently used system call is *gettimeofday*(2). This system call is called both directly by user-space applications as well as indirectly by the C library. Think timestamps or timing loops or polling—all of these frequently need to know what time it is right now. This information is also not secret—any application in any privilege mode (root or any unprivileged user) will get the same answer. Thus the kernel arranges for the information required to answer this question to be placed in memory the process can access. Now a call to *gettimeofday*(2) changes from a system call to a normal function call and a few memory accesses.

Finding the vDSO

The base address of the vDSO (if one exists) is passed by the kernel to each program in the initial auxiliary vector (see *getauxval*(3)), via the **AT_SYSINFO_EHDR** tag.

You must not assume the vDSO is mapped at any particular location in the user's memory map. The base address will usually be randomized at run time every time a new process image is created (at *execve*(2) time). This is done for security reasons, to prevent "return-to-libc" attacks.

For some architectures, there is also an **AT_SYSINFO** tag. This is used only for locating the vsyscall entry point and is frequently omitted or set to 0 (meaning it's not available). This tag is a throwback to the initial vDSO work (see *History* below) and its use should be avoided.

File format

Since the vDSO is a fully formed ELF image, you can do symbol lookups on it. This allows new symbols to be added with newer kernel releases, and allows the C library to detect available functionality at run time when running under different kernel versions. Oftentimes the C library will do detection with the first call and then cache the result for subsequent calls.

All symbols are also versioned (using the GNU version format). This allows the kernel to update the function signature without breaking backward compatibility. This means changing the arguments that the function accepts as well as the return value. Thus, when looking up a symbol in the vDSO, you must always include the version to match the ABI you expect.

Typically the vDSO follows the naming convention of prefixing all symbols with "__vdso_" or "__kernel_" so as to distinguish them from other standard symbols. For example, the "gettimeofday" function is named "__vdso_gettimeofday".

You use the standard C calling conventions when calling any of these functions. No need to worry about weird register or stack behavior.

NOTES

Source

When you compile the kernel, it will automatically compile and link the vDSO code for you. You will frequently find it under the architecture-specific directory:

```
find arch/$ARCH/ -name '*vdso*.so*' -o -name '*gate*.so*'
```

vDSO names

The name of the vDSO varies across architectures. It will often show up in things like glibc's *ldd*(1) output. The exact name should not matter to any code, so do not hardcode it.

user ABI	vDSO name
aarch64	linux-vdso.so.1
arm	linux-vdso.so.1
ia64	linux-gate.so.1
mips	linux-vdso.so.1
ppc/32	linux-vdso32.so.1
ppc/64	linux-vdso64.so.1
riscv	linux-vdso.so.1
s390	linux-vdso32.so.1
s390x	linux-vdso64.so.1
sh	linux-gate.so.1
i386	linux-gate.so.1
x86-64	linux-vdso.so.1
x86/x32	linux-vdso.so.1

strace(1), seccomp(2), and the vDSO

When tracing system calls with *strace*(1), symbols (system calls) that are exported by the vDSO will *not* appear in the trace output. Those system calls will likewise not be visible to *seccomp*(2) filters.

ARCHITECTURE-SPECIFIC NOTES

The subsections below provide architecture-specific notes on the vDSO.

Note that the vDSO that is used is based on the ABI of your user-space code and not the ABI of the kernel. Thus, for example, when you run an i386 32-bit ELF binary, you'll get the same vDSO regardless of whether you run it under an i386 32-bit kernel or under an x86-64 64-bit kernel. Therefore, the name of the user-space ABI should be used to determine which of the sections below is relevant.

ARM functions

The table below lists the symbols exported by the vDSO.

symbol	version
__vdso_gettimeofday	LINUX_2.6 (exported since Linux 4.1)
__vdso_clock_gettime	LINUX_2.6 (exported since Linux 4.1)

Additionally, the ARM port has a code page full of utility functions. Since it's just a raw page of code, there is no ELF information for doing symbol lookups or versioning. It does provide support for different versions though.

For information on this code page, it's best to refer to the kernel documentation as it's extremely detailed and covers everything you need to know: *Documentation/arm/kernel_user_helpers.rst*.

aarch64 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_rt_sigreturn	LINUX_2.6.39
__kernel_gettimeofday	LINUX_2.6.39

__kernel_clock_gettime	LINUX_2.6.39
__kernel_clock_getres	LINUX_2.6.39

bfin (Blackfin) functions (port removed in Linux 4.17)

As this CPU lacks a memory management unit (MMU), it doesn't set up a vDSO in the normal sense. Instead, it maps at boot time a few raw functions into a fixed location in memory. User-space applications then call directly into that region. There is no provision for backward compatibility beyond sniffing raw opcodes, but as this is an embedded CPU, it can get away with things—some of the object formats it runs aren't even ELF based (they're bFLT/FLAT).

For information on this code page, it's best to refer to the public documentation:
http://docs.blackfin.uclinux.org/doku.php?id=linux−kernel:fixed−code

mips functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_gettimeofday	LINUX_2.6 (exported since Linux 4.4)
__kernel_clock_gettime	LINUX_2.6 (exported since Linux 4.4)

ia64 (Itanium) functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_sigtramp	LINUX_2.5
__kernel_syscall_via_break	LINUX_2.5
__kernel_syscall_via_epc	LINUX_2.5

The Itanium port is somewhat tricky. In addition to the vDSO above, it also has "light-weight system calls" (also known as "fast syscalls" or "fsys"). You can invoke these via the *__kernel_syscall_via_epc* vDSO helper. The system calls listed here have the same semantics as if you called them directly via *syscall*(2), so refer to the relevant documentation for each. The table below lists the functions available via this mechanism.

function
clock_gettime
getcpu
getpid
getppid
gettimeofday
set_tid_address

parisc (hppa) functions

The parisc port has a code page with utility functions called a gateway page. Rather than use the normal ELF auxiliary vector approach, it passes the address of the page to the process via the SR2 register. The permissions on the page are such that merely executing those addresses automatically executes with kernel privileges and not in user space. This is done to match the way HP-UX works.

Since it's just a raw page of code, there is no ELF information for doing symbol lookups or versioning. Simply call into the appropriate offset via the branch instruction, for example:

```
ble <offset>(%sr2, %r0)
```

offset	function
00b0	lws_entry (CAS operations)
00e0	set_thread_pointer (used by glibc)
0100	linux_gateway_entry (syscall)

ppc/32 functions

The table below lists the symbols exported by the vDSO. The functions marked with a * are available only when the kernel is a PowerPC64 (64-bit) kernel.

symbol	version
__kernel_clock_getres	LINUX_2.6.15
__kernel_clock_gettime	LINUX_2.6.15
__kernel_clock_gettime64	LINUX_5.11
__kernel_datapage_offset	LINUX_2.6.15
__kernel_get_syscall_map	LINUX_2.6.15
__kernel_get_tbfreq	LINUX_2.6.15
__kernel_getcpu *	LINUX_2.6.15
__kernel_gettimeofday	LINUX_2.6.15
__kernel_sigtramp_rt32	LINUX_2.6.15
__kernel_sigtramp32	LINUX_2.6.15
__kernel_sync_dicache	LINUX_2.6.15
__kernel_sync_dicache_p5	LINUX_2.6.15

Before Linux 5.6, the **CLOCK_REALTIME_COARSE** and **CLOCK_MONOTONIC_COARSE** clocks are *not* supported by the *__kernel_clock_getres* and *__kernel_clock_gettime* interfaces; the kernel falls back to the real system call.

ppc/64 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_clock_getres	LINUX_2.6.15
__kernel_clock_gettime	LINUX_2.6.15
__kernel_datapage_offset	LINUX_2.6.15
__kernel_get_syscall_map	LINUX_2.6.15
__kernel_get_tbfreq	LINUX_2.6.15
__kernel_getcpu	LINUX_2.6.15
__kernel_gettimeofday	LINUX_2.6.15
__kernel_sigtramp_rt64	LINUX_2.6.15
__kernel_sync_dicache	LINUX_2.6.15
__kernel_sync_dicache_p5	LINUX_2.6.15

Before Linux 4.16, the **CLOCK_REALTIME_COARSE** and **CLOCK_MONOTONIC_COARSE** clocks are *not* supported by the *__kernel_clock_getres* and *__kernel_clock_gettime* interfaces; the kernel falls back to the real system call.

riscv functions

The table below lists the symbols exported by the vDSO.

symbol	version
__vdso_rt_sigreturn	LINUX_4.15
__vdso_gettimeofday	LINUX_4.15
__vdso_clock_gettime	LINUX_4.15
__vdso_clock_getres	LINUX_4.15
__vdso_getcpu	LINUX_4.15
__vdso_flush_icache	LINUX_4.15

s390 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_clock_getres	LINUX_2.6.29
__kernel_clock_gettime	LINUX_2.6.29
__kernel_gettimeofday	LINUX_2.6.29

s390x functions

The table below lists the symbols exported by the vDSO.

symbol	version

__kernel_clock_getres	LINUX_2.6.29
__kernel_clock_gettime	LINUX_2.6.29
__kernel_gettimeofday	LINUX_2.6.29

sh (SuperH) functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_rt_sigreturn	LINUX_2.6
__kernel_sigreturn	LINUX_2.6
__kernel_vsyscall	LINUX_2.6

i386 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_sigreturn	LINUX_2.5
__kernel_rt_sigreturn	LINUX_2.5
__kernel_vsyscall	LINUX_2.5
__vdso_clock_gettime	LINUX_2.6 (exported since Linux 3.15)
__vdso_gettimeofday	LINUX_2.6 (exported since Linux 3.15)
__vdso_time	LINUX_2.6 (exported since Linux 3.15)

x86-64 functions

The table below lists the symbols exported by the vDSO. All of these symbols are also available without the "__vdso_" prefix, but you should ignore those and stick to the names below.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6
__vdso_time	LINUX_2.6

x86/x32 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6
__vdso_time	LINUX_2.6

History

The vDSO was originally just a single function—the vsyscall. In older kernels, you might see that name in a process's memory map rather than "vdso". Over time, people realized that this mechanism was a great way to pass more functionality to user space, so it was reconceived as a vDSO in the current format.

SEE ALSO

syscalls(2), *getauxval*(3), *proc*(5)

The documents, examples, and source code in the Linux source code tree:

```
Documentation/ABI/stable/vdso
Documentation/ia64/fsys.rst
Documentation/vDSO/* (includes examples of using the vDSO)

find arch/ -iname '*vdso*' -o -iname '*gate*'
```

NAME

vsock − Linux VSOCK address family

SYNOPSIS

#include <sys/socket.h>
#include <linux/vm_sockets.h>

stream_socket = **socket(AF_VSOCK, SOCK_STREAM, 0);**
datagram_socket = **socket(AF_VSOCK, SOCK_DGRAM, 0);**

DESCRIPTION

The VSOCK address family facilitates communication between virtual machines and the host they are running on. This address family is used by guest agents and hypervisor services that need a communications channel that is independent of virtual machine network configuration.

Valid socket types are **SOCK_STREAM** and **SOCK_DGRAM**. **SOCK_STREAM** provides connection-oriented byte streams with guaranteed, in-order delivery. **SOCK_DGRAM** provides a connectionless datagram packet service with best-effort delivery and best-effort ordering. Availability of these socket types is dependent on the underlying hypervisor.

A new socket is created with

```
socket(AF_VSOCK, socket_type, 0);
```

When a process wants to establish a connection, it calls *connect*(2) with a given destination socket address. The socket is automatically bound to a free port if unbound.

A process can listen for incoming connections by first binding to a socket address using *bind*(2) and then calling *listen*(2).

Data is transmitted using the *send*(2) or *write*(2) families of system calls and data is received using the *recv*(2) or *read*(2) families of system calls.

Address format

A socket address is defined as a combination of a 32-bit Context Identifier (CID) and a 32-bit port number. The CID identifies the source or destination, which is either a virtual machine or the host. The port number differentiates between multiple services running on a single machine.

```
struct sockaddr_vm {
    sa_family_t    svm_family;    /* Address family: AF_VSOCK */
    unsigned short svm_reserved1;
    unsigned int   svm_port;      /* Port # in host byte order */
    unsigned int   svm_cid;       /* Address in host byte order */
    unsigned char  svm_zero[sizeof(struct sockaddr) −
                            sizeof(sa_family_t) −
                            sizeof(unsigned short) −
                            sizeof(unsigned int) −
                            sizeof(unsigned int)];
};
```

svm_family is always set to **AF_VSOCK**. *svm_reserved1* is always set to 0. *svm_port* contains the port number in host byte order. The port numbers below 1024 are called *privileged ports*. Only a process with the **CAP_NET_BIND_SERVICE** capability may *bind*(2) to these port numbers. *svm_zero* must be zero-filled.

There are several special addresses: **VMADDR_CID_ANY** (−1U) means any address for binding; **VMADDR_CID_HYPERVISOR** (0) is reserved for services built into the hypervisor; **VMADDR_CID_LOCAL** (1) is the well-known address for local communication (loopback); **VMADDR_CID_HOST** (2) is the well-known address of the host.

The special constant **VMADDR_PORT_ANY** (−1U) means any port number for binding.

Live migration

Sockets are affected by live migration of virtual machines. Connected **SOCK_STREAM** sockets become disconnected when the virtual machine migrates to a new host. Applications must reconnect when this happens.

The local CID may change across live migration if the old CID is not available on the new host. Bound

sockets are automatically updated to the new CID.

Ioctls

The following ioctls are available on the */dev/vsock* device.

IOCTL_VM_SOCKETS_GET_LOCAL_CID
Get the CID of the local machine. The argument is a pointer to an *unsigned int*.

```
ioctl(fd, IOCTL_VM_SOCKETS_GET_LOCAL_CID, &cid);
```

Consider using **VMADDR_CID_ANY** when binding instead of getting the local CID with **IOCTL_VM_SOCKETS_GET_LOCAL_CID**.

Local communication

VMADDR_CID_LOCAL (1) directs packets to the same host that generated them. This is useful for testing applications on a single host and for debugging.

The local CID obtained with **IOCTL_VM_SOCKETS_GET_LOCAL_CID** can be used for the same purpose, but it is preferable to use **VMADDR_CID_LOCAL**.

ERRORS

EACCES
Unable to bind to a privileged port without the **CAP_NET_BIND_SERVICE** capability.

EADDRINUSE
Unable to bind to a port that is already in use.

EADDRNOTAVAIL
Unable to find a free port for binding or unable to bind to a nonlocal CID.

EINVAL
Invalid parameters. This includes: attempting to bind a socket that is already bound, providing an invalid struct *sockaddr_vm*, and other input validation errors.

ENOPROTOOPT
Invalid socket option in *setsockopt*(2) or *getsockopt*(2).

ENOTCONN
Unable to perform operation on an unconnected socket.

EOPNOTSUPP
Operation not supported. This includes: the **MSG_OOB** flag that is not implemented for the *send*(2) family of syscalls and **MSG_PEEK** for the *recv*(2) family of syscalls.

EPROTONOSUPPORT
Invalid socket protocol number. The protocol should always be 0.

ESOCKTNOSUPPORT
Unsupported socket type in *socket*(2). Only **SOCK_STREAM** and **SOCK_DGRAM** are valid.

VERSIONS

Support for VMware (VMCI) has been available since Linux 3.9. KVM (virtio) is supported since Linux 4.8. Hyper-V is supported since Linux 4.14.

VMADDR_CID_LOCAL is supported since Linux 5.6. Local communication in the guest and on the host is available since Linux 5.6. Previous versions supported only local communication within a guest (not on the host), and with only some transports (VMCI and virtio).

SEE ALSO

bind(2), *connect*(2), *listen*(2), *recv*(2), *send*(2), *socket*(2), *capabilities*(7)

NAME

x25 − ITU-T X.25 / ISO/IEC 8208 protocol interface

SYNOPSIS

#include <sys/socket.h>
#include <linux/x25.h>

x25_socket = **socket(AF_X25, SOCK_SEQPACKET, 0);**

DESCRIPTION

X25 sockets provide an interface to the X.25 packet layer protocol. This allows applications to communicate over a public X.25 data network as standardized by International Telecommunication Union's recommendation X.25 (X.25 DTE-DCE mode). X25 sockets can also be used for communication without an intermediate X.25 network (X.25 DTE-DTE mode) as described in ISO/IEC 8208.

Message boundaries are preserved — a *read*(2) from a socket will retrieve the same chunk of data as output with the corresponding *write*(2) to the peer socket. When necessary, the kernel takes care of segmenting and reassembling long messages by means of the X.25 M-bit. There is no hard-coded upper limit for the message size. However, reassembling of a long message might fail if there is a temporary lack of system resources or when other constraints (such as socket memory or buffer size limits) become effective. If that occurs, the X.25 connection will be reset.

Socket addresses

The **AF_X25** socket address family uses the *struct sockaddr_x25* for representing network addresses as defined in ITU-T recommendation X.121.

```
struct sockaddr_x25 {
    sa_family_t sx25_family;    /* must be AF_X25 */
    x25_address sx25_addr;      /* X.121 Address */
};
```

sx25_addr contains a char array *x25_addr[]* to be interpreted as a null-terminated string. *sx25_addr.x25_addr[]* consists of up to 15 (not counting the terminating null byte) ASCII characters forming the X.121 address. Only the decimal digit characters from '0' to '9' are allowed.

Socket options

The following X.25-specific socket options can be set by using *setsockopt*(2) and read with *getsockopt*(2) with the *level* argument set to **SOL_X25**.

X25_QBITINCL

Controls whether the X.25 Q-bit (Qualified Data Bit) is accessible by the user. It expects an integer argument. If set to 0 (default), the Q-bit is never set for outgoing packets and the Q-bit of incoming packets is ignored. If set to 1, an additional first byte is prepended to each message read from or written to the socket. For data read from the socket, a 0 first byte indicates that the Q-bits of the corresponding incoming data packets were not set. A first byte with value 1 indicates that the Q-bit of the corresponding incoming data packets was set. If the first byte of the data written to the socket is 1, the Q-bit of the corresponding outgoing data packets will be set. If the first byte is 0, the Q-bit will not be set.

VERSIONS

The AF_X25 protocol family is a new feature of Linux 2.2.

BUGS

Plenty, as the X.25 PLP implementation is **CONFIG_EXPERIMENTAL**.

This man page is incomplete.

There is no dedicated application programmer's header file yet; you need to include the kernel header file *<linux/x25.h>*. **CONFIG_EXPERIMENTAL** might also imply that future versions of the interface are not binary compatible.

X.25 N-Reset events are not propagated to the user process yet. Thus, if a reset occurred, data might be lost without notice.

SEE ALSO

socket(2), *socket*(7)

Jonathan Simon Naylor: "The Re-Analysis and Re-Implementation of X.25." The URL is .

NAME

xattr − Extended attributes

DESCRIPTION

Extended attributes are name:value pairs associated permanently with files and directories, similar to the environment strings associated with a process. An attribute may be defined or undefined. If it is defined, its value may be empty or non-empty.

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system (i.e., the *stat*(2) data). They are often used to provide additional functionality to a filesystem—for example, additional security features such as Access Control Lists (ACLs) may be implemented using extended attributes.

Users with search access to a file or directory may use *listxattr*(2) to retrieve a list of attribute names defined for that file or directory.

Extended attributes are accessed as atomic objects. Reading (**getxattr**(2)) retrieves the whole value of an attribute and stores it in a buffer. Writing (**setxattr**(2)) replaces any previous value with the new value.

Space consumed for extended attributes may be counted towards the disk quotas of the file owner and file group.

Extended attribute namespaces

Attribute names are null-terminated strings. The attribute name is always specified in the fully qualified *namespace.attribute* form, for example, *user.mime_type*, *trusted.md5sum*, *system.posix_acl_access*, or *security.selinux*.

The namespace mechanism is used to define different classes of extended attributes. These different classes exist for several reasons; for example, the permissions and capabilities required for manipulating extended attributes of one namespace may differ to another.

Currently, the *security*, *system*, *trusted*, and *user* extended attribute classes are defined as described below. Additional classes may be added in the future.

Extended security attributes

The security attribute namespace is used by kernel security modules, such as Security Enhanced Linux, and also to implement file capabilities (see *capabilities*(7)). Read and write access permissions to security attributes depend on the policy implemented for each security attribute by the security module. When no security module is loaded, all processes have read access to extended security attributes, and write access is limited to processes that have the **CAP_SYS_ADMIN** capability.

System extended attributes

System extended attributes are used by the kernel to store system objects such as Access Control Lists. Read and write access permissions to system attributes depend on the policy implemented for each system attribute implemented by filesystems in the kernel.

Trusted extended attributes

Trusted extended attributes are visible and accessible only to processes that have the **CAP_SYS_ADMIN** capability. Attributes in this class are used to implement mechanisms in user space (i.e., outside the kernel) which keep information in extended attributes to which ordinary processes should not have access.

User extended attributes

User extended attributes may be assigned to files and directories for storing arbitrary additional information such as the mime type, character set or encoding of a file. The access permissions for user attributes are defined by the file permission bits: read permission is required to retrieve the attribute value, and writer permission is required to change it.

The file permission bits of regular files and directories are interpreted differently from the file permission bits of special files and symbolic links. For regular files and directories the file permission bits define access to the file's contents, while for device special files they define access to the device described by the special file. The file permissions of symbolic links are not used in access checks. These differences would allow users to consume filesystem resources in a way not controllable by disk quotas for group or world writable special files and directories.

For this reason, user extended attributes are allowed only for regular files and directories, and access to

user extended attributes is restricted to the owner and to users with appropriate capabilities for directories with the sticky bit set (see the *chmod*(1) manual page for an explanation of the sticky bit).

Filesystem differences

The kernel and the filesystem may place limits on the maximum number and size of extended attributes that can be associated with a file. The VFS-imposed limits on attribute names and values are 255 bytes and 64 kB, respectively. The list of attribute names that can be returned is also limited to 64 kB (see BUGS in *listxattr*(2)).

Some filesystems, such as Reiserfs (and, historically, ext2 and ext3), require the filesystem to be mounted with the **user_xattr** mount option in order for user extended attributes to be used.

In the current ext2, ext3, and ext4 filesystem implementations, the total bytes used by the names and values of all of a file's extended attributes must fit in a single filesystem block (1024, 2048 or 4096 bytes, depending on the block size specified when the filesystem was created).

In the Btrfs, XFS, and Reiserfs filesystem implementations, there is no practical limit on the number of extended attributes associated with a file, and the algorithms used to store extended attribute information on disk are scalable.

In the JFS, XFS, and Reiserfs filesystem implementations, the limit on bytes used in an EA value is the ceiling imposed by the VFS.

In the Btrfs filesystem implementation, the total bytes used for the name, value, and implementation overhead bytes is limited to the filesystem *nodesize* value (16 kB by default).

STANDARDS

Extended attributes are not specified in POSIX.1, but some other systems (e.g., the BSDs and Solaris) provide a similar feature.

NOTES

Since the filesystems on which extended attributes are stored might also be used on architectures with a different byte order and machine word size, care should be taken to store attribute values in an architecture-independent format.

This page was formerly named *attr*(5)

SEE ALSO

attr(1), *getfattr*(1), *setfattr*(1), *getxattr*(2), *ioctl_iflags*(2), *listxattr*(2), *removexattr*(2), *setxattr*(2), *acl*(5), *capabilities*(7), *selinux*(8)

NAME

intro − introduction to administration and privileged commands

DESCRIPTION

Section 8 of the manual describes commands which either can be or are used only by the superuser, like system-administration commands, daemons, and hardware-related commands.

As with the commands described in Section 1, the commands described in this section terminate with an exit status that indicates whether the command succeeded or failed. See *intro*(1) for more information.

NOTES

Authors and copyright conditions

Look at the header of the manual page source for the author(s) and copyright conditions. Note that these can be different from page to page!

NAME

iconvconfig − create iconv module configuration cache

SYNOPSIS

iconvconfig [*options*] [*directory*]...

DESCRIPTION

The *iconv*(3) function internally uses *gconv* modules to convert to and from a character set. A configuration file is used to determine the needed modules for a conversion. Loading and parsing such a configuration file would slow down programs that use *iconv*(3), so a caching mechanism is employed.

The **iconvconfig** program reads iconv module configuration files and writes a fast-loading gconv module configuration cache file.

In addition to the system provided gconv modules, the user can specify custom gconv module directories with the environment variable **GCONV_PATH**. However, iconv module configuration caching is used only when the environment variable **GCONV_PATH** is not set.

OPTIONS

−−nostdlib
: Do not search the system default gconv directory, only the directories provided on the command line.

−−output=*outputfile*
−o *outputfile*
: Use *outputfile* for output instead of the system default cache location.

−−prefix=*pathname*
: Set the prefix to be prepended to the system pathnames. See FILES, below. By default, the prefix is empty. Setting the prefix to *foo*, the gconv module configuration would be read from *foo/usr/lib/gconv/gconv−modules* and the cache would be written to *foo/usr/lib/gconv/gconv−modules.cache*.

−−help
−? Print a usage summary and exit.

−−usage
: Print a short usage summary and exit.

−−version
−V Print the version number, license, and disclaimer of warranty for **iconv**.

EXIT STATUS

Zero on success, nonzero on errors.

FILES

/usr/lib/gconv
: Usual default gconv module path.

/usr/lib/gconv/gconv−modules
: Usual system default gconv module configuration file.

/usr/lib/gconv/gconv−modules.cache
: Usual system gconv module configuration cache.

Depending on the architecture, the above files may instead be located at directories with the path prefix */usr/lib64*.

SEE ALSO

iconv(1), *iconv*(3)

NAME

ld.so, ld−linux.so − dynamic linker/loader

SYNOPSIS

The dynamic linker can be run either indirectly by running some dynamically linked program or shared object (in which case no command-line options to the dynamic linker can be passed and, in the ELF case, the dynamic linker which is stored in the **.interp** section of the program is executed) or directly by running:

*/lib/ld−linux.so.** [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION

The programs **ld.so** and **ld−linux.so*** find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.

Linux binaries require dynamic linking (linking at run time) unless the **−static** option was given to *ld*(1) during compilation.

The program **ld.so** handles a.out binaries, a binary format used long ago. The program **ld−linux.so*** (*/lib/ld−linux.so.1* for libc5, */lib/ld−linux.so.2* for glibc2) handles binaries that are in the more modern ELF format. Both programs have the same behavior, and use the same support files and programs (**ldd**(1), *ldconfig*(8), and */etc/ld.so.conf*).

When resolving shared object dependencies, the dynamic linker first inspects each dependency string to see if it contains a slash (this can occur if a shared object pathname containing slashes was specified at link time). If a slash is found, then the dependency string is interpreted as a (relative or absolute) pathname, and the shared object is loaded using that pathname.

If a shared object dependency does not contain a slash, then it is searched for in the following order:

(1) Using the directories specified in the DT_RPATH dynamic section attribute of the binary if present and DT_RUNPATH attribute does not exist. Use of DT_RPATH is deprecated.

(2) Using the environment variable **LD_LIBRARY_PATH**, unless the executable is being run in secure-execution mode (see below), in which case this variable is ignored.

(3) Using the directories specified in the DT_RUNPATH dynamic section attribute of the binary if present. Such directories are searched only to find those objects required by DT_NEEDED (direct dependencies) entries and do not apply to those objects' children, which must themselves have their own DT_RUNPATH entries. This is unlike DT_RPATH, which is applied to searches for all children in the dependency tree.

(4) From the cache file */etc/ld.so.cache*, which contains a compiled list of candidate shared objects previously found in the augmented library path. If, however, the binary was linked with the **−z nodefaultlib** linker option, shared objects in the default paths are skipped. Shared objects installed in hardware capability directories (see below) are preferred to other shared objects.

(5) In the default path */lib*, and then */usr/lib*. (On some 64-bit architectures, the default paths for 64-bit shared objects are */lib64*, and then */usr/lib64*.) If the binary was linked with the **−z nodefaultlib** linker option, this step is skipped.

Dynamic string tokens

In several places, the dynamic linker expands dynamic string tokens:

- In the environment variables **LD_LIBRARY_PATH**, **LD_PRELOAD**, and **LD_AUDIT**,
- inside the values of the dynamic section tags **DT_NEEDED**, **DT_RPATH**, **DT_RUNPATH**, **DT_AUDIT**, and **DT_DEPAUDIT** of ELF binaries,
- in the arguments to the **ld.so** command line options **--audit**, **--library-path**, and **−−preload** (see below), and
- in the filename arguments to the *dlopen*(3) and *dlmopen*(3) functions.

The substituted tokens are as follows:

$ORIGIN (or equivalently *${ORIGIN}*)

This expands to the directory containing the program or shared object. Thus, an application located in *somedir/app* could be compiled with

```
gcc -Wl,-rpath,'$ORIGIN/../lib'
```

so that it finds an associated shared object in *somedir/lib* no matter where *somedir* is located in the directory hierarchy. This facilitates the creation of "turn-key" applications that do not need to be installed into special directories, but can instead be unpacked into any directory and still find their own shared objects.

$LIB (or equivalently *${LIB}*)
This expands to *lib* or *lib64* depending on the architecture (e.g., on x86-64, it expands to *lib64* and on x86-32, it expands to *lib*).

$PLATFORM (or equivalently *${PLATFORM}*)
This expands to a string corresponding to the processor type of the host system (e.g., "x86_64"). On some architectures, the Linux kernel doesn't provide a platform string to the dynamic linker. The value of this string is taken from the **AT_PLATFORM** value in the auxiliary vector (see *getauxval*(3)).

Note that the dynamic string tokens have to be quoted properly when set from a shell, to prevent their expansion as shell or environment variables.

OPTIONS

--argv0 *string* (since glibc 2.33)
Set *argv[0]* to the value *string* before running the program.

−−audit *list*
Use objects named in *list* as auditors. The objects in *list* are delimited by colons.

−−glibc-hwcaps-mask *list*
only search built-in subdirectories if in *list*.

−−glibc-hwcaps-prepend *list*
Search glibc-hwcaps subdirectories in *list*.

−−inhibit−cache
Do not use */etc/ld.so.cache*.

−−library−path *path*
Use *path* instead of **LD_LIBRARY_PATH** environment variable setting (see below). The names *ORIGIN*, *LIB*, and *PLATFORM* are interpreted as for the **LD_LIBRARY_PATH** environment variable.

−−inhibit−rpath *list*
Ignore RPATH and RUNPATH information in object names in *list*. This option is ignored when running in secure-execution mode (see below). The objects in *list* are delimited by colons or spaces.

−−list List all dependencies and how they are resolved.

--list-diagnostics (since glibc 2.33)
Print system diagnostic information in a machine-readable format, such as some internal loader variables, the auxiliary vector (see *getauxval*(3)), and the environment variables. On some architectures, the command might print additional information (like the cpu features used in GNU indirect function selection on x86). **--list-tunables** (since glibc 2.33) Print the names and values of all tunables, along with the minimum and maximum allowed values.

--preload *list* (since glibc 2.30)
Preload the objects specified in *list*. The objects in *list* are delimited by colons or spaces. The objects are preloaded as explained in the description of the **LD_PRELOAD** environment variable below.

By contrast with **LD_PRELOAD**, the **−−preload** option provides a way to perform preloading for a single executable without affecting preloading performed in any child process that executes a new program.

−−verify
Verify that program is dynamically linked and this dynamic linker can handle it.

ENVIRONMENT

Various environment variables influence the operation of the dynamic linker.

Secure-execution mode

For security reasons, if the dynamic linker determines that a binary should be run in secure-execution mode, the effects of some environment variables are voided or modified, and furthermore those environment variables are stripped from the environment, so that the program does not even see the definitions. Some of these environment variables affect the operation of the dynamic linker itself, and are described below. Other environment variables treated in this way include: **GCONV_PATH**, **GETCONF_DIR**, **HOSTALIASES**, **LOCALDOMAIN**, **LD_AUDIT**, **LD_DEBUG**, **LD_DEBUG_OUTPUT**, **LD_DYNAMIC_WEAK**, **LD_HWCAP_MASK**, **LD_LIBRARY_PATH**, **LD_ORIGIN_PATH**, **LD_PRELOAD**, **LD_PROFILE**, **LD_SHOW_AUXV**, **LOCALDOMAIN**, **LOCPATH**, **MALLOC_TRACE**, **NIS_PATH**, **NLSPATH**, **RESOLV_HOST_CONF**, **RES_OPTIONS**, **TMPDIR**, and **TZDIR**.

A binary is executed in secure-execution mode if the **AT_SECURE** entry in the auxiliary vector (see *getauxval*(3)) has a nonzero value. This entry may have a nonzero value for various reasons, including:

- The process's real and effective user IDs differ, or the real and effective group IDs differ. This typically occurs as a result of executing a set-user-ID or set-group-ID program.
- A process with a non-root user ID executed a binary that conferred capabilities to the process.
- A nonzero value may have been set by a Linux Security Module.

Environment variables

Among the more important environment variables are the following:

LD_ASSUME_KERNEL (from glibc 2.2.3 to glibc 2.36)

Each shared object can inform the dynamic linker of the minimum kernel ABI version that it requires. (This requirement is encoded in an ELF note section that is viewable via *readelf −n* as a section labeled **NT_GNU_ABI_TAG**.) At run time, the dynamic linker determines the ABI version of the running kernel and will reject loading shared objects that specify minimum ABI versions that exceed that ABI version.

LD_ASSUME_KERNEL can be used to cause the dynamic linker to assume that it is running on a system with a different kernel ABI version. For example, the following command line causes the dynamic linker to assume it is running on Linux 2.2.5 when loading the shared objects required by *myprog*:

```
$ LD_ASSUME_KERNEL=2.2.5 ./myprog
```

On systems that provide multiple versions of a shared object (in different directories in the search path) that have different minimum kernel ABI version requirements, **LD_ASSUME_KERNEL** can be used to select the version of the object that is used (dependent on the directory search order).

Historically, the most common use of the **LD_ASSUME_KERNEL** feature was to manually select the older LinuxThreads POSIX threads implementation on systems that provided both LinuxThreads and NPTL (which latter was typically the default on such systems); see *pthreads*(7).

LD_BIND_NOW (since glibc 2.1.1)

If set to a nonempty string, causes the dynamic linker to resolve all symbols at program startup instead of deferring function call resolution to the point when they are first referenced. This is useful when using a debugger.

LD_LIBRARY_PATH

A list of directories in which to search for ELF libraries at execution time. The items in the list are separated by either colons or semicolons, and there is no support for escaping either separator. A zero-length directory name indicates the current working directory.

This variable is ignored in secure-execution mode.

Within the pathnames specified in **LD_LIBRARY_PATH**, the dynamic linker expands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Dynamic string tokens*. Thus, for example, the following would cause a library to be searched for in either the *lib* or *lib64* subdirectory below the directory

containing the program to be executed:

```
$ LD_LIBRARY_PATH='$ORIGIN/$LIB' prog
```

(Note the use of single quotes, which prevent expansion of *$ORIGIN* and *$LIB* as shell variables!)

LD_PRELOAD
A list of additional, user-specified, ELF shared objects to be loaded before all others. This feature can be used to selectively override functions in other shared objects.

The items of the list can be separated by spaces or colons, and there is no support for escaping either separator. The objects are searched for using the rules given under DESCRIPTION. Objects are searched for and added to the link map in the left-to-right order specified in the list.

In secure-execution mode, preload pathnames containing slashes are ignored. Furthermore, shared objects are preloaded only from the standard search directories and only if they have set-user-ID mode bit enabled (which is not typical).

Within the names specified in the **LD_PRELOAD** list, the dynamic linker understands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Dynamic string tokens*. (See also the discussion of quoting under the description of **LD_LIBRARY_PATH**.)

There are various methods of specifying libraries to be preloaded, and these are handled in the following order:

(1) The **LD_PRELOAD** environment variable.

(2) The **--preload** command-line option when invoking the dynamic linker directly.

(3) The */etc/ld.so.preload* file (described below).

LD_TRACE_LOADED_OBJECTS
If set (to any value), causes the program to list its dynamic dependencies, as if run by *ldd*(1), instead of running normally.

Then there are lots of more or less obscure variables, many obsolete or only for internal use.

LD_AUDIT (since glibc 2.4)
A list of user-specified, ELF shared objects to be loaded before all others in a separate linker namespace (i.e., one that does not intrude upon the normal symbol bindings that would occur in the process) These objects can be used to audit the operation of the dynamic linker. The items in the list are colon-separated, and there is no support for escaping the separator.

LD_AUDIT is ignored in secure-execution mode.

The dynamic linker will notify the audit shared objects at so-called auditing checkpoints—for example, loading a new shared object, resolving a symbol, or calling a symbol from another shared object—by calling an appropriate function within the audit shared object. For details, see *rtld-audit*(7). The auditing interface is largely compatible with that provided on Solaris, as described in its *Linker and Libraries Guide*, in the chapter *Runtime Linker Auditing Interface*.

Within the names specified in the **LD_AUDIT** list, the dynamic linker understands the tokens *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Dynamic string tokens*. (See also the discussion of quoting under the description of **LD_LIBRARY_PATH**.)

Since glibc 2.13, in secure-execution mode, names in the audit list that contain slashes are ignored, and only shared objects in the standard search directories that have the set-user-ID mode bit enabled are loaded.

LD_BIND_NOT (since glibc 2.1.95)
If this environment variable is set to a nonempty string, do not update the GOT (global offset table) and PLT (procedure linkage table) after resolving a function symbol. By combining the use of this variable with **LD_DEBUG** (with the categories *bindings* and *symbols*), one can observe all run-time function bindings.

LD_DEBUG (since glibc 2.1)
Output verbose debugging information about operation of the dynamic linker. The content of this variable is one of more of the following categories, separated by colons, commas, or (if the value is quoted) spaces:

help	Specifying *help* in the value of this variable does not run the specified program, and displays a help message about which categories can be specified in this environment variable.
all	Print all debugging information (except *statistics* and *unused*; see below).
bindings	Display information about which definition each symbol is bound to.
files	Display progress for input file.
libs	Display library search paths.
reloc	Display relocation processing.
scopes	Display scope information.
statistics	Display relocation statistics.
symbols	Display search paths for each symbol look-up.
unused	Determine unused DSOs.
versions	Display version dependencies.

Since glibc 2.3.4, **LD_DEBUG** is ignored in secure-execution mode, unless the file */etc/suid−debug* exists (the content of the file is irrelevant).

LD_DEBUG_OUTPUT (since glibc 2.1)
By default, **LD_DEBUG** output is written to standard error. If **LD_DEBUG_OUTPUT** is defined, then output is written to the pathname specified by its value, with the suffix "." (dot) followed by the process ID appended to the pathname.

LD_DEBUG_OUTPUT is ignored in secure-execution mode.

LD_DYNAMIC_WEAK (since glibc 2.1.91)
By default, when searching shared libraries to resolve a symbol reference, the dynamic linker will resolve to the first definition it finds.

Old glibc versions (before glibc 2.2), provided a different behavior: if the linker found a symbol that was weak, it would remember that symbol and keep searching in the remaining shared libraries. If it subsequently found a strong definition of the same symbol, then it would instead use that definition. (If no further symbol was found, then the dynamic linker would use the weak symbol that it initially found.)

The old glibc behavior was nonstandard. (Standard practice is that the distinction between weak and strong symbols should have effect only at static link time.) In glibc 2.2, the dynamic linker was modified to provide the current behavior (which was the behavior that was provided by most other implementations at that time).

Defining the **LD_DYNAMIC_WEAK** environment variable (with any value) provides the old (nonstandard) glibc behavior, whereby a weak symbol in one shared library may be overridden by a strong symbol subsequently discovered in another shared library. (Note that even when this variable is set, a strong symbol in a shared library will not override a weak definition of the same symbol in the main program.)

Since glibc 2.3.4, **LD_DYNAMIC_WEAK** is ignored in secure-execution mode.

LD_HWCAP_MASK (from glibc 2.1 to glibc 2.38)
Mask for hardware capabilities. Since glibc 2.26, the option might be ignored if glibc does not support tunables.

LD_ORIGIN_PATH (since glibc 2.1)
Path where the binary is found.

Since glibc 2.4, **LD_ORIGIN_PATH** is ignored in secure-execution mode.

LD_POINTER_GUARD (from glibc 2.4 to glibc 2.22)
Set to 0 to disable pointer guarding. Any other value enables pointer guarding, which is also the default. Pointer guarding is a security mechanism whereby some pointers to code stored in writable program memory (return addresses saved by *setjmp*(3) or function pointers used by various glibc internals) are mangled semi-randomly to make it more difficult for an attacker to hijack the pointers for use in the event of a buffer overrun or stack-smashing attack. Since glibc 2.23, **LD_POINTER_GUARD** can no longer be used to disable pointer guarding, which is now always enabled.

LD_PROFILE (since glibc 2.1)
The name of a (single) shared object to be profiled, specified either as a pathname or a soname. Profiling output is appended to the file whose name is: $LD_PROFILE_OUTPUT/$LD_PROFILE.*profile*.

Since glibc 2.2.5, **LD_PROFILE** uses a different default path in secure-execution mode.

LD_PROFILE_OUTPUT (since glibc 2.1)
Directory where **LD_PROFILE** output should be written. If this variable is not defined, or is defined as an empty string, then the default is */var/tmp*.

LD_PROFILE_OUTPUT is ignored in secure-execution mode; instead */var/profile* is always used.

LD_SHOW_AUXV (since glibc 2.1)
If this environment variable is defined (with any value), show the auxiliary array passed up from the kernel (see also *getauxval*(3)).

Since glibc 2.3.4, **LD_SHOW_AUXV** is ignored in secure-execution mode.

LD_TRACE_PRELINKING (from glibc 2.4 to glibc 2.35)
If this environment variable is defined, trace prelinking of the object whose name is assigned to this environment variable. (Use *ldd*(1) to get a list of the objects that might be traced.) If the object name is not recognized, then all prelinking activity is traced.

LD_USE_LOAD_BIAS (from glibc 2.3.3 to glibc 2.35)
By default (i.e., if this variable is not defined), executables and prelinked shared objects will honor base addresses of their dependent shared objects and (nonprelinked) position-independent executables (PIEs) and other shared objects will not honor them. If **LD_USE_LOAD_BIAS** is defined with the value 1, both executables and PIEs will honor the base addresses. If **LD_USE_LOAD_BIAS** is defined with the value 0, neither executables nor PIEs will honor the base addresses.

Since glibc 2.3.3, this variable is ignored in secure-execution mode.

LD_VERBOSE (since glibc 2.1)
If set to a nonempty string, output symbol versioning information about the program if the **LD_TRACE_LOADED_OBJECTS** environment variable has been set.

LD_WARN (since glibc 2.1.3)
If set to a nonempty string, warn about unresolved symbols.

LD_PREFER_MAP_32BIT_EXEC (x86-64 only; since glibc 2.23)
According to the Intel Silvermont software optimization guide, for 64-bit applications, branch prediction performance can be negatively impacted when the target of a branch is more than 4 GB away from the branch. If this environment variable is set (to any value), the dynamic linker will first try to map executable pages using the *mmap*(2) **MAP_32BIT** flag, and fall back to mapping without that flag if that attempt fails. NB: MAP_32BIT will map to the low 2 GB (not 4 GB) of the address space.

Because **MAP_32BIT** reduces the address range available for address space layout randomization (ASLR), **LD_PREFER_MAP_32BIT_EXEC** is always disabled in secure-execution mode.

FILES

/lib/ld.so
a.out dynamic linker/loader

/lib/ld−linux.so.{1,2}
: ELF dynamic linker/loader

/etc/ld.so.cache
: File containing a compiled list of directories in which to search for shared objects and an ordered list of candidate shared objects. See *ldconfig*(8).

/etc/ld.so.preload
: File containing a whitespace-separated list of ELF shared objects to be loaded before the program. See the discussion of **LD_PRELOAD** above. If both **LD_PRELOAD** and */etc/ld.so.preload* are employed, the libraries specified by **LD_PRELOAD** are preloaded first. */etc/ld.so.preload* has a system-wide effect, causing the specified libraries to be preloaded for all programs that are executed on the system. (This is usually undesirable, and is typically employed only as an emergency remedy, for example, as a temporary workaround to a library misconfiguration issue.)

lib.so**
: shared objects

NOTES

Legacy Hardware capabilities (from glibc 2.5 to glibc 2.37)

Some shared objects are compiled using hardware-specific instructions which do not exist on every CPU. Such objects should be installed in directories whose names define the required hardware capabilities, such as */usr/lib/sse2/*. The dynamic linker checks these directories against the hardware of the machine and selects the most suitable version of a given shared object. Hardware capability directories can be cascaded to combine CPU features. The list of supported hardware capability names depends on the CPU. The following names are currently recognized:

Alpha ev4, ev5, ev56, ev6, ev67

MIPS loongson2e, loongson2f, octeon, octeon2

PowerPC
: 4xxmac, altivec, arch_2_05, arch_2_06, booke, cellbe, dfp, efpdouble, efpsingle, fpu, ic_snoop, mmu, notb, pa6t, power4, power5, power5+, power6x, ppc32, ppc601, ppc64, smt, spe, ucache, vsx

SPARC
: flush, muldiv, stbar, swap, ultra3, v9, v9v, v9v2

s390 dfp, eimm, esan3, etf3enh, g5, highgprs, hpage, ldisp, msa, stfle, z900, z990, z9-109, z10, zarch

x86 (32-bit only)
: acpi, apic, clflush, cmov, cx8, dts, fxsr, ht, i386, i486, i586, i686, mca, mmx, mtrr, pat, pbe, pge, pn, pse36, sep, ss, sse, sse2, tm

The legacy hardware capabilities support has the drawback that each new feature added grows the search path exponentially, because it has to be added to every combination of the other existing features.

For instance, on x86 32-bit, if the hardware supports **i686** and **sse2**, the resulting search path will be **i686/sse2:i686:sse2:.**. A new capability **newcap** will set the search path to **newcap/i686/sse2:newcap/i686:newcap/sse2:newcap:i686/sse2:i686:sse2:**.

glibc Hardware capabilities (from glibc 2.33)

glibc 2.33 added a new hardware capability scheme,
: where under each CPU architecture, certain levels can be defined, grouping support for certain features or special instructions. Each architecture level has a fixed set of paths that it adds to the dynamic linker search list, depending on the hardware of the machine. Since each new architecture level is not combined with previously existing ones, the new scheme does not have the drawback of growing the dynamic linker search list uncontrollably.

For instance, on x86 64-bit, if the hardware supports **x86_64-v3** (for instance Intel Haswell or AMD Excavator), the resulting search path will be **glibc-hwcaps/x86-64-v3:glibc-hwcaps/x86-64-v2:.** The following paths are currently supported, in priority order.

PowerPC (64-bit little-endian only)
power10, power9

s390 (64-bit only)
z16, z15, z14, z13

x86 (64-bit only)
x86-64-v4, x86-64-v3, x86-64-v2

glibc 2.37 removed support for the legacy hardware capabilities.

SEE ALSO

ld(1), *ldd*(1), *pldd*(1), *sprof*(1), *dlopen*(3), *getauxval*(3), *elf*(5), *capabilities*(7), *rtld-audit*(7), *ldconfig*(8), *sln*(8)

NAME

ldconfig − configure dynamic linker run-time bindings

SYNOPSIS

/sbin/ldconfig [**−nNvVX**] [**−C** *cache*] [**−f** *conf*] [**−r** *root*] *directory* . . .

/sbin/ldconfig −l [**−v**] *library* . . .

/sbin/ldconfig −p

DESCRIPTION

ldconfig creates the necessary links and cache to the most recent shared libraries found in the directories specified on the command line, in the file */etc/ld.so.conf* , and in the trusted directories, */lib* and */usr/lib*. On some 64-bit architectures such as x86-64, */lib* and */usr/lib* are the trusted directories for 32-bit libraries, while */lib64* and */usr/lib64* are used for 64-bit libraries.

The cache is used by the run-time linker, *ld.so* or *ld−linux.so*. **ldconfig** checks the header and filenames of the libraries it encounters when determining which versions should have their links updated. **ldconfig** should normally be run by the superuser as it may require write permission on some root owned directories and files.

ldconfig will look only at files that are named *lib**.so** (for regular shared objects) or *ld−**.so** (for the dynamic loader itself). Other files will be ignored. Also, **ldconfig** expects a certain pattern to how the symbolic links are set up, like this example, where the middle file (**libfoo.so.1** here) is the SONAME for the library:

```
libfoo.so −> libfoo.so.1 −> libfoo.so.1.12
```

Failure to follow this pattern may result in compatibility issues after an upgrade.

OPTIONS

−−format= *fmt*

−c *fmt* (Since glibc 2.2) Use cache format *fmt*, which is one of **old**, **new**, or **compat**. Since glibc 2.32, the default is **new**. Before that, it was **compat**.

−C *cache*
Use *cache* instead of */etc/ld.so.cache*.

−f *conf* Use *conf* instead of */etc/ld.so.conf* .

−−ignore−aux−cache

−i (Since glibc 2.7) Ignore auxiliary cache file.

−l (Since glibc 2.2) Interpret each operand as a library name and configure its links. Intended for use only by experts.

−n Process only the directories specified on the command line; don't process the trusted directories, nor those specified in */etc/ld.so.conf* . Implies **-N**.

−N Don't rebuild the cache. Unless **−X** is also specified, links are still updated.

−−print−cache

−p Print the lists of directories and candidate libraries stored in the current cache.

−r *root* Change to and use *root* as the root directory.

−−verbose

−v Verbose mode. Print current version number, the name of each directory as it is scanned, and any links that are created. Overrides quiet mode.

−−version

−V Print program version.

−X Don't update links. Unless **−N** is also specified, the cache is still rebuilt.

FILES

/lib/ld.so
is the run-time linker/loader.

/etc/ld.so.conf
: contains a list of directories, one per line, in which to search for libraries.

/etc/ld.so.cache
: contains an ordered list of libraries found in the directories specified in */etc/ld.so.conf*, as well as those found in the trusted directories.

SEE ALSO

ldd(1), *ld.so*(8)

NAME

nscd − name service cache daemon

DESCRIPTION

nscd is a daemon that provides a cache for the most common name service requests. The default configuration file, */etc/nscd.conf* , determines the behavior of the cache daemon. See *nscd.conf*(5).

nscd provides caching for accesses of the *passwd*(5), *group*(5), *hosts*(5) *services*(5) and *netgroup* databases through standard libc interfaces, such as *getpwnam*(3), *getpwuid*(3), *getgrnam*(3), *getgrgid*(3), *gethostbyname*(3), and others.

There are two caches for each database: a positive one for items found, and a negative one for items not found. Each cache has a separate TTL (time-to-live) period for its data. Note that the shadow file is specifically not cached. *getspnam*(3) calls remain uncached as a result.

OPTIONS

−−help will give you a list with all options and what they do.

NOTES

The daemon will try to watch for changes in configuration files appropriate for each database (e.g., */etc/passwd* for the *passwd* database or */etc/hosts* and */etc/resolv.conf* for the *hosts* database), and flush the cache when these are changed. However, this will happen only after a short delay (unless the *inotify*(7) mechanism is available and glibc 2.9 or later is available), and this auto-detection does not cover configuration files required by nonstandard NSS modules, if any are specified in */etc/nsswitch.conf* . In that case, you need to run the following command after changing the configuration file of the database so that **nscd** invalidates its cache:

```
$ nscd -i <database>
```

SEE ALSO

nscd.conf(5), *nsswitch.conf*(5)

NAME

sln − create symbolic links

SYNOPSIS

sln *source dest*
sln *filelist*

DESCRIPTION

The **sln** program creates symbolic links. Unlike the *ln*(1) program, it is statically linked. This means that if for some reason the dynamic linker is not working, **sln** can be used to make symbolic links to dynamic libraries.

The command line has two forms. In the first form, it creates *dest* as a new symbolic link to *source*.

In the second form, *filelist* is a list of space-separated pathname pairs, and the effect is as if **sln** was executed once for each line of the file, with the two pathnames as the arguments.

The **sln** program supports no command-line options.

SEE ALSO

ln(1), *ld.so*(8), *ldconfig*(8)

NAME

tzselect − select a timezone

SYNOPSIS

tzselect [**-c** *coord*] [**-n** *limit*] [**--help**] [**--version**]

DESCRIPTION

The **tzselect** program asks the user for information about the current location, and outputs the resulting timezone to standard output. The output is suitable as a value for the TZ environment variable.

All interaction with the user is done via standard input and standard error.

OPTIONS

-c *coord*

Instead of asking for continent and then country and then city, ask for selection from time zones whose largest cities are closest to the location with geographical coordinates *coord*. Use ISO 6709 notation for *coord*, that is, a latitude immediately followed by a longitude. The latitude and longitude should be signed integers followed by an optional decimal point and fraction: positive numbers represent north and east, negative south and west. Latitudes with two and longitudes with three integer digits are treated as degrees; latitudes with four or six and longitudes with five or seven integer digits are treated as *DDMM*, *DDDMM*, *DDMMSS*, or *DDDMMSS* representing *DD* or *DDD* degrees, *MM* minutes, and zero or *SS* seconds, with any trailing fractions represent fractional minutes or (if *SS* is present) seconds. The decimal point is that of the current locale. For example, in the (default) C locale, **-c +40.689-074.045** specifies 40.689° N, 74.045° W, **-c +4041.4-07402.7** specifies 40° 41.4′ N, 74° 2.7′ W, and **-c +404121-0740240** specifies 40° 41′ 21″ N, 74° 2′ 40″ W. If *coord* is not one of the documented forms, the resulting behavior is unspecified.

-n *limit*

When **-c** is used, display the closest *limit* locations (default 10).

--help Output help information and exit.

--version

Output version information and exit.

ENVIRONMENT VARIABLES

AWK Name of a POSIX-compliant **awk** program (default: **awk**).

TZDIR

Name of the directory containing timezone data files (default: **/usr/share/zoneinfo**).

FILES

TZDIR/**iso3166.tab**

Table of ISO 3166 2-letter country codes and country names.

TZDIR/**zone1970.tab**

Table of country codes, latitude and longitude, timezones, and descriptive comments.

TZDIR/*TZ*

Timezone data file for timezone *TZ*.

EXIT STATUS

The exit status is zero if a timezone was successfully obtained from the user, nonzero otherwise.

SEE ALSO

newctime(3), tzfile(5), zdump(8), zic(8)

NOTES

Applications should not assume that **tzselect**'s output matches the user's political preferences.

NAME

zdump − timezone dumper

SYNOPSIS

zdump [*option* ...] [*timezone* ...]

DESCRIPTION

The **zdump** program prints the current time in each *timezone* named on the command line.

OPTIONS

--version
: Output version information and exit.

--help Output short usage message and exit.

-i Output a description of time intervals. For each *timezone* on the command line, output an interval-format description of the timezone. See "INTERVAL FORMAT" below.

-v Output a verbose description of time intervals. For each *timezone* on the command line, print the times at the two extreme time values, the times (if present) at and just beyond the boundaries of years that *localtime*(3) and *gmtime*(3) can represent, and the times both one second before and exactly at each detected time discontinuity. Each line is followed by **isdst=***D* where *D* is positive, zero, or negative depending on whether the given time is daylight saving time, standard time, or an unknown time type, respectively. Each line is also followed by **gmtoff=***N* if the given local time is known to be *N* seconds east of Greenwich.

-V Like **-v**, except omit output concerning extreme time and year values. This generates output that is easier to compare to that of implementations with different time representations.

-c [*loyear***,**]*hiyear*
: Cut off interval output at the given year(s). Cutoff times are computed using the proleptic Gregorian calendar with year 0 and with Universal Time (UT) ignoring leap seconds. Cutoffs are at the start of each year, where the lower-bound timestamp is inclusive and the upper is exclusive; for example, **-c 1970,2070** selects transitions on or after 1970-01-01 00:00:00 UTC and before 2070-01-01 00:00:00 UTC. The default cutoff is **-500,2500**.

-t [*lotime***,**]*hitime*
: Cut off interval output at the given time(s), given in decimal seconds since 1970-01-01 00:00:00 Coordinated Universal Time (UTC). The *timezone* determines whether the count includes leap seconds. As with **-c**, the cutoff's lower bound is inclusive and its upper bound is exclusive.

INTERVAL FORMAT

The interval format is a compact text representation that is intended to be both human- and machine-readable. It consists of an empty line, then a line "TZ=*string*" where *string* is a double-quoted string giving the timezone, a second line "- - *interval*" describing the time interval before the first transition if any, and zero or more following lines "*date time interval*", one line for each transition time and following interval. Fields are separated by single tabs.

Dates are in *yyyy*-*mm*-*dd* format and times are in 24-hour *hh*:*mm*:*ss* format where *hh*<24. Times are in local time immediately after the transition. A time interval description consists of a UT offset in signed ±*hhmmss* format, a time zone abbreviation, and an isdst flag. An abbreviation that equals the UT offset is omitted; other abbreviations are double-quoted strings unless they consist of one or more alphabetic characters. An isdst flag is omitted for standard time, and otherwise is a decimal integer that is unsigned and positive (typically 1) for daylight saving time and negative for unknown.

In times and in UT offsets with absolute value less than 100 hours, the seconds are omitted if they are zero, and the minutes are also omitted if they are also zero. Positive UT offsets are east of Greenwich. The UT offset -00 denotes a UT placeholder in areas where the actual offset is unspecified; by convention, this occurs when the UT offset is zero and the time zone abbreviation begins with "-" or is "zzz".

In double-quoted strings, escape sequences represent unusual characters. The escape sequences are \s for space, and \", \\, \f, \n, \r, \t, and \v with their usual meaning in the C programming language. E.g., the double-quoted string ""CET\s\"\\"" represents the character sequence "CET "\".

Here is an example of the output, with the leading empty line omitted. (This example is shown with tab stops set far enough apart so that the tabbed columns line up.)

```
TZ="Pacific/Honolulu"
-           -         -103126  LMT
1896-01-13  12:01:26  -1030    HST
1933-04-30  03        -0930    HDT  1
1933-05-21  11        -1030    HST
1942-02-09  03        -0930    HWT  1
1945-08-14  13:30     -0930    HPT  1
1945-09-30  01        -1030    HST
1947-06-08  02:30     -10      HST
```

Here, local time begins 10 hours, 31 minutes and 26 seconds west of UT, and is a standard time abbreviated LMT. Immediately after the first transition, the date is 1896-01-13 and the time is 12:01:26, and the following time interval is 10.5 hours west of UT, a standard time abbreviated HST. Immediately after the second transition, the date is 1933-04-30 and the time is 03:00:00 and the following time interval is 9.5 hours west of UT, is abbreviated HDT, and is daylight saving time. Immediately after the last transition the date is 1947-06-08 and the time is 02:30:00, and the following time interval is 10 hours west of UT, a standard time abbreviated HST.

Here are excerpts from another example:

```
TZ="Europe/Astrakhan"
-           -         +031212  LMT
1924-04-30  23:47:48  +03
1930-06-21  01        +04
1981-04-01  01        +05           1
1981-09-30  23        +04
...
2014-10-26  01        +03
2016-03-27  03        +04
```

This time zone is east of UT, so its UT offsets are positive. Also, many of its time zone abbreviations are omitted since they duplicate the text of the UT offset.

LIMITATIONS

Time discontinuities are found by sampling the results returned by *localtime*(3) at twelve-hour intervals. This works in all real-world cases; one can construct artificial time zones for which this fails.

In the **-v** and **-V** output, "UT" denotes the value returned by *gmtime*(3), which uses UTC for modern timestamps and some other UT flavor for timestamps that predate the introduction of UTC. No attempt is currently made to have the output use "UTC" for newer and "UT" for older timestamps, partly because the exact date of the introduction of UTC is problematic.

SEE ALSO

tzfile(5), *zic*(8)

NAME

zic − timezone compiler

SYNOPSIS

zic [*option* ...] [*filename* ...]

DESCRIPTION

The **zic** program reads text from the file(s) named on the command line and creates the timezone information format (TZif) files specified in this input. If a *filename* is "**-**", standard input is read.

OPTIONS

--version

Output version information and exit.

--help Output short usage message and exit.

-b *bloat*

Output backward-compatibility data as specified by *bloat*. If *bloat* is **fat**, generate additional data entries that work around potential bugs or incompatibilities in older software, such as software that mishandles the 64-bit generated data. If *bloat* is **slim**, keep the output files small; this can help check for the bugs and incompatibilities. The default is **slim**, as software that mishandles 64-bit data typically mishandles timestamps after the year 2038 anyway. Also see the **-r** option for another way to alter output size.

-d *directory*

Create time conversion information files in the named directory rather than in the standard directory named below.

-l *timezone*

Use *timezone* as local time. **zic** will act as if the input contained a link line of the form

Link *timezone* localtime

If *timezone* is **-**, any already-existing link is removed.

-L *leapsecondfilename*

Read leap second information from the file with the given name. If this option is not used, no leap second information appears in output files.

-p *timezone*

Use *timezone*'s rules when handling nonstandard TZ strings like "EET-2EEST" that lack transition rules. **zic** will act as if the input contained a link line of the form

Link *timezone* posixrules

If *timezone* is "**-**" (the default), any already-existing link is removed.

Unless *timezone is* "**-**", this option is obsolete and poorly supported. Among other things it should not be used for timestamps after the year 2037, and it should not be combined with **-b slim** if *timezone*'s transitions are at standard time or Universal Time (UT) instead of local time.

-r [**@***lo*][/**@***hi*]

Limit the applicability of output files to timestamps in the range from *lo* (inclusive) to *hi* (exclusive), where *lo* and *hi* are possibly signed decimal counts of seconds since the Epoch (1970-01-01 00:00:00 UTC). Omitted counts default to extreme values. The output files use UT offset 0 and abbreviation "-00" in place of the omitted timestamp data. For example, "zic -r @0" omits data intended for negative timestamps (i.e., before the Epoch), and "zic -r @0/@2147483648" outputs data intended only for nonnegative timestamps that fit into 31-bit signed integers. On platforms with GNU **date**, "zic -r @$(date +%s)" omits data intended for past timestamps. Although this option typically reduces the output file's size, the size can increase due to the need to represent the timestamp range boundaries, particularly if *hi* causes a TZif file to contain explicit entries for pre-*hi* transitions rather than concisely representing them with an extended POSIX.1-2017 TZ string. Also see the **-b slim** option for another way to shrink output size.

-R @*hi*

Generate redundant trailing explicit transitions for timestamps that occur less than *hi* seconds since the Epoch, even though the transitions could be more concisely represented via the extended POSIX.1-2017 TZ string. This option does not affect the represented timestamps. Although it accommodates nonstandard TZif readers that ignore the extended POSIX.1-2017 TZ string, it increases the size of the altered output files.

-t *file* When creating local time information, put the configuration link in the named file rather than in the standard location.

-v Be more verbose, and complain about the following situations:

The input specifies a link to a link, something not supported by some older parsers, including **zic** itself through release 2022e.

A year that appears in a data file is outside the range of representable years.

A time of 24:00 or more appears in the input. Pre-1998 versions of **zic** prohibit 24:00, and pre-2007 versions prohibit times greater than 24:00.

A rule goes past the start or end of the month. Pre-2004 versions of **zic** prohibit this.

A time zone abbreviation uses a **%z** format. Pre-2015 versions of **zic** do not support this.

A timestamp contains fractional seconds. Pre-2018 versions of **zic** do not support this.

The input contains abbreviations that are mishandled by pre-2018 versions of **zic** due to a longstanding coding bug. These abbreviations include "L" for "Link", "mi" for "min", "Sa" for "Sat", and "Su" for "Sun".

The output file does not contain all the information about the long-term future of a timezone, because the future cannot be summarized as an extended POSIX.1-2017 TZ string. For example, as of 2023 this problem occurs for Morocco's daylight-saving rules, as these rules are based on predictions for when Ramadan will be observed, something that an extended POSIX.1-2017 TZ string cannot represent.

The output contains data that may not be handled properly by client code designed for older **zic** output formats. These compatibility issues affect only timestamps before 1970 or after the start of 2038.

The output contains a truncated leap second table, which can cause some older TZif readers to misbehave. This can occur if the **-L** option is used, and either an Expires line is present or the **-r** option is also used.

The output file contains more than 1200 transitions, which may be mishandled by some clients. The current reference client supports at most 2000 transitions; pre-2014 versions of the reference client support at most 1200 transitions.

A time zone abbreviation has fewer than 3 or more than 6 characters. POSIX requires at least 3, and requires implementations to support at least 6.

An output file name contains a byte that is not an ASCII letter, "-", "/", or "_"; or it contains a file name component that contains more than 14 bytes or that starts with "-".

FILES

Input files use the format described in this section; output files use *tzfile*(5) format.

Input files should be text files, that is, they should be a series of zero or more lines, each ending in a newline byte and containing at most 2048 bytes counting the newline, and without any NUL bytes. The input text's encoding is typically UTF-8 or ASCII; it should have a unibyte representation for the POSIX Portable Character Set (PPCS) ⟨https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap06.html⟩ and the encoding's non-unibyte characters should consist entirely of non-PPCS bytes. Non-PPCS characters typically occur only in comments: although output file names and time zone abbreviations can contain nearly any character, other software will work better if these are limited to the restricted syntax described under the **-v** option.

Input lines are made up of fields. Fields are separated from one another by one or more white space characters. The white space characters are space, form feed, carriage return, newline, tab, and vertical tab. Leading and trailing white space on input lines is ignored. An unquoted sharp character (#) in the

input introduces a comment which extends to the end of the line the sharp character appears on. White space characters and sharp characters may be enclosed in double quotes (") if they're to be used as part of a field. Any line that is blank (after comment stripping) is ignored. Nonblank lines are expected to be of one of three types: rule lines, zone lines, and link lines.

Names must be in English and are case insensitive. They appear in several contexts, and include month and weekday names and keywords such as **maximum**, **only**, **Rolling**, and **Zone**. A name can be abbreviated by omitting all but an initial prefix; any abbreviation must be unambiguous in context.

A rule line has the form

```
Rule  NAME  FROM  TO    -  IN   ON       AT     SAVE  LETTER/S
```

For example:

```
Rule  US    1967  1973  -  Apr  lastSun  2:00w  1:00d  D
```

The fields that make up a rule line are:

NAME Gives the name of the rule set that contains this line. The name must start with a character that is neither an ASCII digit nor "`-`" nor "+". To allow for future extensions, an unquoted name should not contain characters from the set "`!$%&'()*,/:;<=>?@[\]^`{|}~`". "!$%&'()*,/:;<=>?@[\]^`{|}~".

FROM Gives the first year in which the rule applies. Any signed integer year can be supplied; the proleptic Gregorian calendar is assumed, with year 0 preceding year 1. Rules can describe times that are not representable as time values, with the unrepresentable times ignored; this allows rules to be portable among hosts with differing time value types.

TO Gives the final year in which the rule applies. The word **maximum** (or an abbreviation) means the indefinite future, and the word **only** (or an abbreviation) may be used to repeat the value of the **FROM** field.

`-` Is a reserved field and should always contain "`-`" for compatibility with older versions of **zic**. It was previously known as the **TYPE** field, which could contain values to allow a separate script to further restrict in which "types" of years the rule would apply.

IN Names the month in which the rule takes effect. Month names may be abbreviated.

ON Gives the day on which the rule takes effect. Recognized forms include:

5	the fifth of the month
lastSun	the last Sunday in the month
lastMon	the last Monday in the month
Sun>=8	first Sunday on or after the eighth
Sun<=25	last Sunday on or before the 25th

A weekday name (e.g., **Sunday**) or a weekday name preceded by "last" (e.g., **lastSunday**) may be abbreviated or spelled out in full. There must be no white space characters within the **ON** field. The "<=" and ">=" constructs can result in a day in the neighboring month; for example, the IN-ON combination "Oct Sun>=31" stands for the first Sunday on or after October 31, even if that Sunday occurs in November.

AT Gives the time of day at which the rule takes effect, relative to 00:00, the start of a calendar day. Recognized forms include:

2	time in hours
2:00	time in hours and minutes
01:28:14	time in hours, minutes, and seconds
00:19:32.13	time with fractional seconds
12:00	midday, 12 hours after 00:00
15:00	3 PM, 15 hours after 00:00
24:00	end of day, 24 hours after 00:00
260:00	260 hours after 00:00

-2:30	2.5 hours before 00:00
-	equivalent to 0

Although **zic** rounds times to the nearest integer second (breaking ties to the even integer), the fractions may be useful to other applications requiring greater precision. The source format does not specify any maximum precision. Any of these forms may be followed by the letter **w** if the given time is local or "wall clock" time, **s** if the given time is standard time without any adjustment for daylight saving, or **u** (or **g** or **z**) if the given time is universal time; in the absence of an indicator, local (wall clock) time is assumed. These forms ignore leap seconds; for example, if a leap second occurs at 00:59:60 local time, "1:00" stands for 3601 seconds after local midnight instead of the usual 3600 seconds. The intent is that a rule line describes the instants when a clock/calendar set to the type of time specified in the **AT** field would show the specified date and time of day.

SAVE Gives the amount of time to be added to local standard time when the rule is in effect, and whether the resulting time is standard or daylight saving. This field has the same format as the **AT** field except with a different set of suffix letters: **s** for standard time and **d** for daylight saving time. The suffix letter is typically omitted, and defaults to **s** if the offset is zero and to **d** otherwise. Negative offsets are allowed; in Ireland, for example, daylight saving time is observed in winter and has a negative offset relative to Irish Standard Time. The offset is merely added to standard time; for example, **zic** does not distinguish a 10:30 standard time plus an 0:30 **SAVE** from a 10:00 standard time plus a 1:00 **SAVE**.

LETTER/S
Gives the "variable part" (for example, the "S" or "D" in "EST" or "EDT") of time zone abbreviations to be used when this rule is in effect. If this field is "-", the variable part is null.

A zone line has the form

```
Zone  NAME          STDOFF   RULES  FORMAT   [UNTIL]
```

For example:

```
Zone  Asia/Amman    2:00     Jordan EE%sT    2017 Oct 27 01:00
```

The fields that make up a zone line are:

NAME The name of the timezone. This is the name used in creating the time conversion information file for the timezone. It should not contain a file name component "." or ".."; a file name component is a maximal substring that does not contain "/".

STDOFF
The amount of time to add to UT to get standard time, without any adjustment for daylight saving. This field has the same format as the **AT** and **SAVE** fields of rule lines, except without suffix letters; begin the field with a minus sign if time must be subtracted from UT.

RULES
The name of the rules that apply in the timezone or, alternatively, a field in the same format as a rule-line SAVE column, giving the amount of time to be added to local standard time and whether the resulting time is standard or daylight saving. If this field is - then standard time always applies. When an amount of time is given, only the sum of standard time and this amount matters.

FORMAT
The format for time zone abbreviations. The pair of characters **%s** is used to show where the "variable part" of the time zone abbreviation goes. Alternatively, a format can use the pair of characters **%z** to stand for the UT offset in the form ±*hh*, ±*hhmm*, or ±*hhmmss*, using the shortest form that does not lose information, where *hh*, *mm*, and *ss* are the hours, minutes, and seconds east (+) or west (−) of UT. Alternatively, a slash (/) separates standard and daylight abbreviations. To conform to POSIX, a time zone abbreviation should contain only alphanumeric ASCII characters, "+" and "-". By convention, the time zone abbreviation "-00" is a placeholder that means local time is unspecified.

UNTIL

The time at which the UT offset or the rule(s) change for a location. It takes the form of one to four fields YEAR [MONTH [DAY [TIME]]]. If this is specified, the time zone information is generated from the given UT offset and rule change until the time specified, which is interpreted using the rules in effect just before the transition. The month, day, and time of day have the same format as the IN, ON, and AT fields of a rule; trailing fields can be omitted, and default to the earliest possible value for the missing fields.

The next line must be a "continuation" line; this has the same form as a zone line except that the string "Zone" and the name are omitted, as the continuation line will place information starting at the time specified as the "until" information in the previous line in the file used by the previous line. Continuation lines may contain "until" information, just as zone lines do, indicating that the next line is a further continuation.

If a zone changes at the same instant that a rule would otherwise take effect in the earlier zone or continuation line, the rule is ignored. A zone or continuation line *L* with a named rule set starts with standard time by default: that is, any of *L*'s timestamps preceding *L*'s earliest rule use the rule in effect after *L*'s first transition into standard time. In a single zone it is an error if two rules take effect at the same instant, or if two zone changes take effect at the same instant.

If a continuation line subtracts *N* seconds from the UT offset after a transition that would be interpreted to be later if using the continuation line's UT offset and rules, the "until" time of the previous zone or continuation line is interpreted according to the continuation line's UT offset and rules, and any rule that would otherwise take effect in the next *N* seconds is instead assumed to take effect simultaneously. For example:

```
# Rule  NAME    FROM    TO      -   IN   ON       AT    SAVE   LETTER/S
Rule    US      1967    2006    -   Oct  lastSun  2:00  0      S
Rule    US      1967    1973    -   Apr  lastSun  2:00  1:00   D
# Zone  NAME                    STDOFF  RULES   FORMAT  [UNTIL]
Zone    America/Menominee       -5:00   -       EST     1973 Apr 29 2:00
                                -6:00   US      C%sT
```

Here, an incorrect reading would be there were two clock changes on 1973-04-29, the first from 02:00 EST (-05) to 01:00 CST (-06), and the second an hour later from 02:00 CST (-06) to 03:00 CDT (-05). However, **zic** interprets this more sensibly as a single transition from 02:00 CST (-05) to 02:00 CDT (-05).

A link line has the form

```
Link    TARGET          LINK-NAME
```

For example:

```
Link    Europe/Istanbul    Asia/Istanbul
```

The **TARGET** field should appear as the **NAME** field in some zone line or as the **LINK-NAME** field in some link line. The **LINK-NAME** field is used as an alternative name for that zone; it has the same syntax as a zone line's **NAME** field. Links can chain together, although the behavior is unspecified if a chain of one or more links does not terminate in a Zone name. A link line can appear before the line that defines the link target. For example:

```
Link    Greenwich   G_M_T
Link    Etc/GMT     Greenwich
Zone    Etc/GMT     0   -   GMT
```

The two links are chained together, and G_M_T, Greenwich, and Etc/GMT all name the same zone.

Except for continuation lines, lines may appear in any order in the input. However, the behavior is unspecified if multiple zone or link lines define the same name.

The file that describes leap seconds can have leap lines and an expiration line. Leap lines have the

following form:

```
Leap  YEAR  MONTH  DAY  HH:MM:SS  CORR  R/S
```

For example:

```
Leap  2016  Dec  31  23:59:60  +  S
```

The **YEAR**, **MONTH**, **DAY**, and **HH:MM:SS** fields tell when the leap second happened. The **CORR** field should be "+" if a second was added or "-" if a second was skipped. The **R/S** field should be (an abbreviation of) "Stationary" if the leap second time given by the other fields should be interpreted as UTC or (an abbreviation of) "Rolling" if the leap second time given by the other fields should be interpreted as local (wall clock) time.

Rolling leap seconds were implemented back when it was not clear whether common practice was rolling or stationary, with concerns that one would see Times Square ball drops where there'd be a "3... 2... 1... leap... Happy New Year" countdown, placing the leap second at midnight New York time rather than midnight UTC. However, this countdown style does not seem to have caught on, which means rolling leap seconds are not used in practice; also, they are not supported if the **-r** option is used.

The expiration line, if present, has the form:

```
Expires  YEAR  MONTH  DAY  HH:MM:SS
```

For example:

```
Expires  2020  Dec  28  00:00:00
```

The **YEAR**, **MONTH**, **DAY**, and **HH:MM:SS** fields give the expiration timestamp in UTC for the leap second table.

EXTENDED EXAMPLE

Here is an extended example of **zic** input, intended to illustrate many of its features.

```
# Rule  NAME   FROM  TO    -  IN   ON       AT    SAVE  LETTER/S
Rule    Swiss  1941  1942  -  May  Mon>=1   1:00  1:00  S
Rule    Swiss  1941  1942  -  Oct  Mon>=1   2:00  0     -

Rule    EU     1977  1980  -  Apr  Sun>=1   1:00u 1:00  S
Rule    EU     1977  only  -  Sep  lastSun  1:00u 0     -
Rule    EU     1978  only  -  Oct  1        1:00u 0     -
Rule    EU     1979  1995  -  Sep  lastSun  1:00u 0     -
Rule    EU     1981  max   -  Mar  lastSun  1:00u 1:00  S
Rule    EU     1996  max   -  Oct  lastSun  1:00u 0     -

# Zone  NAME          STDOFF     RULES  FORMAT  [UNTIL]
Zone    Europe/Zurich 0:34:08    -      LMT     1853 Jul 16
                      0:29:45.50 -      BMT     1894 Jun
                      1:00       Swiss  CE%sT   1981
                      1:00       EU     CE%sT

Link    Europe/Zurich Europe/Vaduz
```

In this example, the EU rules are for the European Union and for its predecessor organization, the European Communities. The timezone is named Europe/Zurich and it has the alias Europe/Vaduz. This example says that Zurich was 34 minutes and 8 seconds east of UT until 1853-07-16 at 00:00, when the legal offset was changed to 7° 26′ 22.50″, which works out to 0:29:45.50; **zic** treats this by rounding it to 0:29:46. After 1894-06-01 at 00:00 the UT offset became one hour and Swiss daylight saving rules (defined with lines beginning with "Rule Swiss") apply. From 1981 to the present, EU daylight saving rules have applied, and the UTC offset has remained at one hour.

In 1941 and 1942, daylight saving time applied from the first Monday in May at 01:00 to the first

Monday in October at 02:00. The pre-1981 EU daylight-saving rules have no effect here, but are included for completeness. Since 1981, daylight saving has begun on the last Sunday in March at 01:00 UTC. Until 1995 it ended the last Sunday in September at 01:00 UTC, but this changed to the last Sunday in October starting in 1996.

For purposes of display, "LMT" and "BMT" were initially used, respectively. Since Swiss rules and later EU rules were applied, the time zone abbreviation has been CET for standard time and CEST for daylight saving time.

FILES

/etc/localtime
: Default local timezone file.

/usr/share/zoneinfo
: Default timezone information directory.

NOTES

For areas with more than two types of local time, you may need to use local standard time in the **AT** field of the earliest transition time's rule to ensure that the earliest transition time recorded in the compiled file is correct.

If, for a particular timezone, a clock advance caused by the start of daylight saving coincides with and is equal to a clock retreat caused by a change in UT offset, **zic** produces a single transition to daylight saving at the new UT offset without any change in local (wall clock) time. To get separate transitions use multiple zone continuation lines specifying transition instants using universal time.

SEE ALSO

tzfile(5), *zdump*(8)