**OPEN(2) Linux Programmer's Manual OPEN(2)**

**NAME**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

open, openat, creat - open and possibly create a file

**SYNOPSIS**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**int open(const char \****pathname***, int** *flags***);**

**int open(const char \****pathname***, int** *flags***, mode\_t** *mode***);**

**int creat(const char \****pathname***, mode\_t** *mode***);**

**int openat(int** *dirfd***, const char \****pathname***, int** *flags***);**

**int openat(int** *dirfd***, const char \****pathname***, int** *flags***, mode\_t** *mode***);**

Feature Test Macro Requirements for glibc (see [feature\_test\_macros(7)](http://man7.org/linux/man-pages/man7/feature_test_macros.7.html)):

**openat**():

Since glibc 2.10:

\_POSIX\_C\_SOURCE >= 200809L

Before glibc 2.10:

\_ATFILE\_SOURCE

**DESCRIPTION**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

The **open**() system call opens the file specified by *pathname*. If the

specified file does not exist, it may optionally (if **O\_CREAT** is

specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative

integer that is used in subsequent system calls ([read(2)](http://man7.org/linux/man-pages/man2/read.2.html), [write(2)](http://man7.org/linux/man-pages/man2/write.2.html),

[lseek(2)](http://man7.org/linux/man-pages/man2/lseek.2.html), [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html), etc.) to refer to the open file. The file

descriptor returned by a successful call will be the lowest-numbered

file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an

[execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html) (i.e., the **FD\_CLOEXEC** file descriptor flag described in

[fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) is initially disabled); the **O\_CLOEXEC** flag, described below,

can be used to change this default. The file offset is set to the

beginning of the file (see [lseek(2)](http://man7.org/linux/man-pages/man2/lseek.2.html)).

A call to **open**() creates a new *open file description*, an entry in the

system-wide table of open files. The open file description records

the file offset and the file status flags (see below). A file

descriptor is a reference to an open file description; this reference

is unaffected if *pathname* is subsequently removed or modified to

refer to a different file. For further details on open file

descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*:

**O\_RDONLY**, **O\_WRONLY**, or **O\_RDWR**. These request opening the file read-

only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags

can be bitwise-*or*'d in *flags*. The *file creation flags* are **O\_CLOEXEC**,

**O\_CREAT**, **O\_DIRECTORY**, **O\_EXCL**, **O\_NOCTTY**, **O\_NOFOLLOW**, **O\_TMPFILE**, and

**O\_TRUNC**. The *file status flags* are all of the remaining flags listed

below. The distinction between these two groups of flags is that the

file creation flags affect the semantics of the open operation

itself, while the file status flags affect the semantics of

subsequent I/O operations. The file status flags can be retrieved

and (in some cases) modified; see [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) for details.

The full list of file creation flags and file status flags is as

follows:

**O\_APPEND**

The file is opened in append mode. Before each [write(2)](http://man7.org/linux/man-pages/man2/write.2.html), the

file offset is positioned at the end of the file, as if with

[lseek(2)](http://man7.org/linux/man-pages/man2/lseek.2.html). The modification of the file offset and the write

operation are performed as a single atomic step.

**O\_APPEND** may lead to corrupted files on NFS filesystems if

more than one process appends data to a file at once. This is

because NFS does not support appending to a file, so the

client kernel has to simulate it, which can't be done without

a race condition.

**O\_ASYNC**

Enable signal-driven I/O: generate a signal (**SIGIO** by default,

but this can be changed via [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)) when input or output

becomes possible on this file descriptor. This feature is

available only for terminals, pseudoterminals, sockets, and

(since Linux 2.6) pipes and FIFOs. See [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) for further

details. See also BUGS, below.

**O\_CLOEXEC** (since Linux 2.6.23)

Enable the close-on-exec flag for the new file descriptor.

Specifying this flag permits a program to avoid additional

[fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_SETFD** operations to set the **FD\_CLOEXEC** flag.

Note that the use of this flag is essential in some

multithreaded programs, because using a separate [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)

**F\_SETFD** operation to set the **FD\_CLOEXEC** flag does not suffice

to avoid race conditions where one thread opens a file

descriptor and attempts to set its close-on-exec flag using

[fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) at the same time as another thread does a [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html)

plus [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html). Depending on the order of execution, the race

may lead to the file descriptor returned by **open**() being

unintentionally leaked to the program executed by the child

process created by [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html). (This kind of race is in

principle possible for any system call that creates a file

descriptor whose close-on-exec flag should be set, and various

other Linux system calls provide an equivalent of the

**O\_CLOEXEC** flag to deal with this problem.)

**O\_CREAT**

If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective

user ID of the process.

The group ownership (group ID) of the new file is set either

to the effective group ID of the process (System V semantics)

or to the group ID of the parent directory (BSD semantics).

On Linux, the behavior depends on whether the set-group-ID

mode bit is set on the parent directory: if that bit is set,

then BSD semantics apply; otherwise, System V semantics apply.

For some filesystems, the behavior also depends on the

*bsdgroups* and *sysvgroups* mount options described in [mount(8)](http://man7.org/linux/man-pages/man8/mount.8.html)).

The *mode* argument specifies the file mode bits be applied when

a new file is created. This argument must be supplied when

**O\_CREAT** or **O\_TMPFILE** is specified in *flags*; if neither **O\_CREAT**

nor **O\_TMPFILE** is specified, then *mode* is ignored. The

effective mode is modified by the process's *umask* in the usual

way: in the absence of a default ACL, the mode of the created

file is *(mode & ~umask)*. Note that this mode applies only to

future accesses of the newly created file; the **open**() call

that creates a read-only file may well return a read/write

file descriptor.

The following symbolic constants are provided for *mode*:

**S\_IRWXU** 00700 user (file owner) has read, write, and execute

permission

**S\_IRUSR** 00400 user has read permission

**S\_IWUSR** 00200 user has write permission

**S\_IXUSR** 00100 user has execute permission

**S\_IRWXG** 00070 group has read, write, and execute permission

**S\_IRGRP** 00040 group has read permission

**S\_IWGRP** 00020 group has write permission

**S\_IXGRP** 00010 group has execute permission

**S\_IRWXO** 00007 others have read, write, and execute permission

**S\_IROTH** 00004 others have read permission

**S\_IWOTH** 00002 others have write permission

**S\_IXOTH** 00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode*

is unspecified. On Linux, the following bits are also honored

in *mode*:

**S\_ISUID** 0004000 set-user-ID bit

**S\_ISGID** 0002000 set-group-ID bit (see [inode(7)](http://man7.org/linux/man-pages/man7/inode.7.html)).

**S\_ISVTX** 0001000 sticky bit (see [inode(7)](http://man7.org/linux/man-pages/man7/inode.7.html)).

**O\_DIRECT** (since Linux 2.4.10)

Try to minimize cache effects of the I/O to and from this

file. In general this will degrade performance, but it is

useful in special situations, such as when applications do

their own caching. File I/O is done directly to/from user-

space buffers. The **O\_DIRECT** flag on its own makes an effort

to transfer data synchronously, but does not give the

guarantees of the **O\_SYNC** flag that data and necessary metadata

are transferred. To guarantee synchronous I/O, **O\_SYNC** must be

used in addition to **O\_DIRECT**. See NOTES below for further

discussion.

A semantically similar (but deprecated) interface for block

devices is described in [raw(8)](http://man7.org/linux/man-pages/man8/raw.8.html).

**O\_DIRECTORY**

If *pathname* is not a directory, cause the open to fail. This

flag was added in kernel version 2.1.126, to avoid denial-of-

service problems if [opendir(3)](http://man7.org/linux/man-pages/man3/opendir.3.html) is called on a FIFO or tape

device.

**O\_DSYNC**

Write operations on the file will complete according to the

requirements of synchronized I/O *data* integrity completion.

By the time [write(2)](http://man7.org/linux/man-pages/man2/write.2.html) (and similar) return, the output data has

been transferred to the underlying hardware, along with any

file metadata that would be required to retrieve that data

(i.e., as though each [write(2)](http://man7.org/linux/man-pages/man2/write.2.html) was followed by a call to

[fdatasync(2)](http://man7.org/linux/man-pages/man2/fdatasync.2.html)). *See NOTES below*.

**O\_EXCL** Ensure that this call creates the file: if this flag is

specified in conjunction with **O\_CREAT**, and *pathname* already

exists, then **open**() fails with the error **EEXIST**.

When these two flags are specified, symbolic links are not

followed: if *pathname* is a symbolic link, then **open**() fails

regardless of where the symbolic link points.

In general, the behavior of **O\_EXCL** is undefined if it is used

without **O\_CREAT**. There is one exception: on Linux 2.6 and

later, **O\_EXCL** can be used without **O\_CREAT** if *pathname* refers

to a block device. If the block device is in use by the

system (e.g., mounted), **open**() fails with the error **EBUSY**.

On NFS, **O\_EXCL** is supported only when using NFSv3 or later on

kernel 2.6 or later. In NFS environments where **O\_EXCL** support

is not provided, programs that rely on it for performing

locking tasks will contain a race condition. Portable

programs that want to perform atomic file locking using a

lockfile, and need to avoid reliance on NFS support for

**O\_EXCL**, can create a unique file on the same filesystem (e.g.,

incorporating hostname and PID), and use [link(2)](http://man7.org/linux/man-pages/man2/link.2.html) to make a

link to the lockfile. If [link(2)](http://man7.org/linux/man-pages/man2/link.2.html) returns 0, the lock is

successful. Otherwise, use [stat(2)](http://man7.org/linux/man-pages/man2/stat.2.html) on the unique file to

check if its link count has increased to 2, in which case the

lock is also successful.

**O\_LARGEFILE**

(LFS) Allow files whose sizes cannot be represented in an

*off\_t* (but can be represented in an *off64\_t*) to be opened.

The **\_LARGEFILE64\_SOURCE** macro must be defined (before

including *any* header files) in order to obtain this

definition. Setting the **\_FILE\_OFFSET\_BITS** feature test macro

to 64 (rather than using **O\_LARGEFILE**) is the preferred method

of accessing large files on 32-bit systems (see

[feature\_test\_macros(7)](http://man7.org/linux/man-pages/man7/feature_test_macros.7.html)).

**O\_NOATIME** (since Linux 2.6.8)

Do not update the file last access time (*st\_atime* in the

inode) when the file is [read(2)](http://man7.org/linux/man-pages/man2/read.2.html).

This flag can be employed only if one of the following

conditions is true:

\* The effective UID of the process matches the owner UID of

the file.

\* The calling process has the **CAP\_FOWNER** capability in its

user namespace and the owner UID of the file has a mapping

in the namespace.

This flag is intended for use by indexing or backup programs,

where its use can significantly reduce the amount of disk

activity. This flag may not be effective on all filesystems.

One example is NFS, where the server maintains the access

time.

**O\_NOCTTY**

If *pathname* refers to a terminal device—see [tty(4)](http://man7.org/linux/man-pages/man4/tty.4.html)—it will not

become the process's controlling terminal even if the process

does not have one.

**O\_NOFOLLOW**

If *pathname* is a symbolic link, then the open fails, with the

error **ELOOP**. Symbolic links in earlier components of the

pathname will still be followed. (Note that the **ELOOP** error

that can occur in this case is indistinguishable from the case

where an open fails because there are too many symbolic links

found while resolving components in the prefix part of the

pathname.)

This flag is a FreeBSD extension, which was added to Linux in

version 2.1.126, and has subsequently been standardized in

POSIX.1-2008.

See also **O\_PATH** below.

**O\_NONBLOCK** or **O\_NDELAY**

When possible, the file is opened in nonblocking mode.

Neither the **open**() nor any subsequent operations on the file

descriptor which is returned will cause the calling process to

wait.

Note that this flag has no effect for regular files and block

devices; that is, I/O operations will (briefly) block when

device activity is required, regardless of whether **O\_NONBLOCK**

is set. Since **O\_NONBLOCK** semantics might eventually be

implemented, applications should not depend upon blocking

behavior when specifying this flag for regular files and block

devices.

For the handling of FIFOs (named pipes), see also [fifo(7)](http://man7.org/linux/man-pages/man7/fifo.7.html).

For a discussion of the effect of **O\_NONBLOCK** in conjunction

with mandatory file locks and with file leases, see [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html).

**O\_PATH** (since Linux 2.6.39)

Obtain a file descriptor that can be used for two purposes: to

indicate a location in the filesystem tree and to perform

operations that act purely at the file descriptor level. The

file itself is not opened, and other file operations (e.g.,

[read(2)](http://man7.org/linux/man-pages/man2/read.2.html), [write(2)](http://man7.org/linux/man-pages/man2/write.2.html), [fchmod(2)](http://man7.org/linux/man-pages/man2/fchmod.2.html), [fchown(2)](http://man7.org/linux/man-pages/man2/fchown.2.html), [fgetxattr(2)](http://man7.org/linux/man-pages/man2/fgetxattr.2.html),

[ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html), [mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html)) fail with the error **EBADF**.

The following operations *can* be performed on the resulting

file descriptor:

\* [close(2)](http://man7.org/linux/man-pages/man2/close.2.html).

\* [fchdir(2)](http://man7.org/linux/man-pages/man2/fchdir.2.html), if the file descriptor refers to a directory

(since Linux 3.5).

\* [fstat(2)](http://man7.org/linux/man-pages/man2/fstat.2.html) (since Linux 3.6).

\* [fstatfs(2)](http://man7.org/linux/man-pages/man2/fstatfs.2.html) (since Linux 3.12).

\* Duplicating the file descriptor ([dup(2)](http://man7.org/linux/man-pages/man2/dup.2.html), [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_DUPFD**,

etc.).

\* Getting and setting file descriptor flags ([fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_GETFD**

and **F\_SETFD**).

\* Retrieving open file status flags using the [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)

**F\_GETFL** operation: the returned flags will include the bit

**O\_PATH**.

\* Passing the file descriptor as the *dirfd* argument of

**openat**() and the other "\*at()" system calls. This includes

[linkat(2)](http://man7.org/linux/man-pages/man2/linkat.2.html) with **AT\_EMPTY\_PATH** (or via procfs using

**AT\_SYMLINK\_FOLLOW**) even if the file is not a directory.

\* Passing the file descriptor to another process via a UNIX

domain socket (see **SCM\_RIGHTS** in [unix(7)](http://man7.org/linux/man-pages/man7/unix.7.html)).

When **O\_PATH** is specified in *flags*, flag bits other than

**O\_CLOEXEC**, **O\_DIRECTORY**, and **O\_NOFOLLOW** are ignored.

Opening a file or directory with the **O\_PATH** flag requires no

permissions on the object itself (but does require execute

permission on the directories in the path prefix). Depending

on the subsequent operation, a check for suitable file

permissions may be performed (e.g., [fchdir(2)](http://man7.org/linux/man-pages/man2/fchdir.2.html) requires execute

permission on the directory referred to by its file descriptor

argument). By contrast, obtaining a reference to a filesystem

object by opening it with the **O\_RDONLY** flag requires that the

caller have read permission on the object, even when the

subsequent operation (e.g., [fchdir(2)](http://man7.org/linux/man-pages/man2/fchdir.2.html), [fstat(2)](http://man7.org/linux/man-pages/man2/fstat.2.html)) does not

require read permission on the object.

If *pathname* is a symbolic link and the **O\_NOFOLLOW** flag is also

specified, then the call returns a file descriptor referring

to the symbolic link. This file descriptor can be used as the

*dirfd* argument in calls to [fchownat(2)](http://man7.org/linux/man-pages/man2/fchownat.2.html), [fstatat(2)](http://man7.org/linux/man-pages/man2/fstatat.2.html), [linkat(2)](http://man7.org/linux/man-pages/man2/linkat.2.html),

and [readlinkat(2)](http://man7.org/linux/man-pages/man2/readlinkat.2.html) with an empty pathname to have the calls

operate on the symbolic link.

If *pathname* refers to an automount point that has not yet been

triggered, so no other filesystem is mounted on it, then the

call returns a file descriptor referring to the automount

directory without triggering a mount. [fstatfs(2)](http://man7.org/linux/man-pages/man2/fstatfs.2.html) can then be

used to determine if it is, in fact, an untriggered automount

point (**.f\_type == AUTOFS\_SUPER\_MAGIC**).

One use of **O\_PATH** for regular files is to provide the

equivalent of POSIX.1's **O\_EXEC** functionality. This permits us

to open a file for which we have execute permission but not

read permission, and then execute that file, with steps

something like the following:

char buf[PATH\_MAX];

fd = open("some\_prog", O\_PATH);

snprintf(buf, PATH\_MAX, "/proc/self/fd/%d", fd);

execl(buf, "some\_prog", (char \*) NULL);

An **O\_PATH** file descriptor can also be passed as the argument

of [fexecve(3)](http://man7.org/linux/man-pages/man3/fexecve.3.html).

**O\_SYNC** Write operations on the file will complete according to the

requirements of synchronized I/O *file* integrity completion (by

contrast with the synchronized I/O *data* integrity completion

provided by **O\_DSYNC**.)

By the time [write(2)](http://man7.org/linux/man-pages/man2/write.2.html) (or similar) returns, the output data and

associated file metadata have been transferred to the underly‐

ing hardware (i.e., as though each [write(2)](http://man7.org/linux/man-pages/man2/write.2.html) was followed by a

call to [fsync(2)](http://man7.org/linux/man-pages/man2/fsync.2.html)). *See NOTES below*.

**O\_TMPFILE** (since Linux 3.11)

Create an unnamed temporary regular file. The *pathname* argu‐

ment specifies a directory; an unnamed inode will be created

in that directory's filesystem. Anything written to the

resulting file will be lost when the last file descriptor is

closed, unless the file is given a name.

**O\_TMPFILE** must be specified with one of **O\_RDWR** or **O\_WRONLY**

and, optionally, **O\_EXCL**. If **O\_EXCL** is not specified, then

[linkat(2)](http://man7.org/linux/man-pages/man2/linkat.2.html) can be used to link the temporary file into the

filesystem, making it permanent, using code like the follow‐

ing:

char path[PATH\_MAX];

fd = open("/path/to/dir", O\_TMPFILE | O\_RDWR,

S\_IRUSR | S\_IWUSR);

/\* File I/O on 'fd'... \*/

snprintf(path, PATH\_MAX, "/proc/self/fd/%d", fd);

linkat(AT\_FDCWD, path, AT\_FDCWD, "/path/for/file",

AT\_SYMLINK\_FOLLOW);

In this case, the **open**() *mode* argument determines the file

permission mode, as with **O\_CREAT**.

Specifying **O\_EXCL** in conjunction with **O\_TMPFILE** prevents a

temporary file from being linked into the filesystem in the

above manner. (Note that the meaning of **O\_EXCL** in this case

is different from the meaning of **O\_EXCL** otherwise.)

There are two main use cases for **O\_TMPFILE**:

\* Improved [tmpfile(3)](http://man7.org/linux/man-pages/man3/tmpfile.3.html) functionality: race-free creation of

temporary files that (1) are automatically deleted when

closed; (2) can never be reached via any pathname; (3) are

not subject to symlink attacks; and (4) do not require the

caller to devise unique names.

\* Creating a file that is initially invisible, which is then

populated with data and adjusted to have appropriate

filesystem attributes ([fchown(2)](http://man7.org/linux/man-pages/man2/fchown.2.html), [fchmod(2)](http://man7.org/linux/man-pages/man2/fchmod.2.html), [fsetxattr(2)](http://man7.org/linux/man-pages/man2/fsetxattr.2.html),

etc.) before being atomically linked into the filesystem

in a fully formed state (using [linkat(2)](http://man7.org/linux/man-pages/man2/linkat.2.html) as described

above).

**O\_TMPFILE** requires support by the underlying filesystem; only

a subset of Linux filesystems provide that support. In the

initial implementation, support was provided in the ext2,

ext3, ext4, UDF, Minix, and shmem filesystems. Support for

other filesystems has subsequently been added as follows: XFS

(Linux 3.15); Btrfs (Linux 3.16); F2FS (Linux 3.16); and ubifs

(Linux 4.9)

**O\_TRUNC**

If the file already exists and is a regular file and the

access mode allows writing (i.e., is **O\_RDWR** or **O\_WRONLY**) it

will be truncated to length 0. If the file is a FIFO or ter‐

minal device file, the **O\_TRUNC** flag is ignored. Otherwise,

the effect of **O\_TRUNC** is unspecified.

**creat()**

A call to **creat**() is equivalent to calling **open**() with *flags* equal to

**O\_CREAT|O\_WRONLY|O\_TRUNC**.

**openat()**

The **openat**() system call operates in exactly the same way as **open**(),

except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted

relative to the directory referred to by the file descriptor *dirfd*

(rather than relative to the current working directory of the calling

process, as is done by **open**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT\_FDCWD**, then

*pathname* is interpreted relative to the current working directory of

the calling process (like **open**()).

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

**RETURN VALUE**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**open**(), **openat**(), and **creat**() return the new file descriptor, or -1

if an error occurred (in which case, [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set appropriately).

**ERRORS**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**open**(), **openat**(), and **creat**() can fail with the following errors:

**EACCES** The requested access to the file is not allowed, or search

permission is denied for one of the directories in the path

prefix of *pathname*, or the file did not exist yet and write

access to the parent directory is not allowed. (See also

[path\_resolution(7)](http://man7.org/linux/man-pages/man7/path_resolution.7.html).)

**EDQUOT** Where **O\_CREAT** is specified, the file does not exist, and the

user's quota of disk blocks or inodes on the filesystem has

been exhausted.

**EEXIST** *pathname* already exists and **O\_CREAT** and **O\_EXCL** were used.

**EFAULT** *pathname* points outside your accessible address space.

**EFBIG** See **EOVERFLOW**.

**EINTR** While blocked waiting to complete an open of a slow device

(e.g., a FIFO; see [fifo(7)](http://man7.org/linux/man-pages/man7/fifo.7.html)), the call was interrupted by a

signal handler; see [signal(7)](http://man7.org/linux/man-pages/man7/signal.7.html).

**EINVAL** The filesystem does not support the **O\_DIRECT** flag. See **NOTES**

for more information.

**EINVAL** Invalid value in *flags*.

**EINVAL O\_TMPFILE** was specified in *flags*, but neither **O\_WRONLY** nor

**O\_RDWR** was specified.

**EINVAL O\_CREAT** was specified in *flags* and the final component

("basename") of the new file's *pathname* is invalid (e.g., it

contains characters not permitted by the underlying

filesystem).

**EISDIR** *pathname* refers to a directory and the access requested

involved writing (that is, **O\_WRONLY** or **O\_RDWR** is set).

**EISDIR** *pathname* refers to an existing directory, **O\_TMPFILE** and one of

**O\_WRONLY** or **O\_RDWR** were specified in *flags*, but this kernel

version does not provide the **O\_TMPFILE** functionality.

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

*pathname*.

**ELOOP** *pathname* was a symbolic link, and *flags* specified **O\_NOFOLLOW**

but not **O\_PATH**.

**EMFILE** The per-process limit on the number of open file descriptors

has been reached (see the description of **RLIMIT\_NOFILE** in

[getrlimit(2)](http://man7.org/linux/man-pages/man2/getrlimit.2.html)).

**ENAMETOOLONG**

*pathname* was too long.

**ENFILE** The system-wide limit on the total number of open files has

been reached.

**ENODEV** *pathname* refers to a device special file and no corresponding

device exists. (This is a Linux kernel bug; in this situation

**ENXIO** must be returned.)

**ENOENT O\_CREAT** is not set and the named file does not exist. Or, a

directory component in *pathname* does not exist or is a

dangling symbolic link.

**ENOENT** *pathname* refers to a nonexistent directory, **O\_TMPFILE** and one

of **O\_WRONLY** or **O\_RDWR** were specified in *flags*, but this kernel

version does not provide the **O\_TMPFILE** functionality.

**ENOMEM** The named file is a FIFO, but memory for the FIFO buffer can't

be allocated because the per-user hard limit on memory

allocation for pipes has been reached and the caller is not

privileged; see [pipe(7)](http://man7.org/linux/man-pages/man7/pipe.7.html).

**ENOMEM** Insufficient kernel memory was available.

**ENOSPC** *pathname* was to be created but the device containing *pathname*

has no room for the new file.

**ENOTDIR**

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

directory, or **O\_DIRECTORY** was specified and *pathname* was not a

directory.

**ENXIO O\_NONBLOCK** | **O\_WRONLY** is set, the named file is a FIFO, and no

process has the FIFO open for reading.

**ENXIO** The file is a device special file and no corresponding device

exists.

**EOPNOTSUPP**

The filesystem containing *pathname* does not support **O\_TMPFILE**.

**EOVERFLOW**

*pathname* refers to a regular file that is too large to be

opened. The usual scenario here is that an application

compiled on a 32-bit platform without *-D\_FILE\_OFFSET\_BITS=64*

tried to open a file whose size exceeds *(1<<31)-1* bytes; see

also **O\_LARGEFILE** above. This is the error specified by

POSIX.1; in kernels before 2.6.24, Linux gave the error **EFBIG**

for this case.

**EPERM** The **O\_NOATIME** flag was specified, but the effective user ID of

the caller did not match the owner of the file and the caller

was not privileged.

**EPERM** The operation was prevented by a file seal; see [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html).

**EROFS** *pathname* refers to a file on a read-only filesystem and write

access was requested.

**ETXTBSY**

*pathname* refers to an executable image which is currently

being executed and write access was requested.

**ETXTBSY**

*pathname* refers to a file that is currently in use as a swap

file, and the **O\_TRUNC** flag was specified.

**ETXTBSY**

*pathname* refers to a file that is currently being read by the

kernel (e.g. for module/firmware loading), and write access

was requested.

**EWOULDBLOCK**

The **O\_NONBLOCK** flag was specified, and an incompatible lease

was held on the file (see [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)).

The following additional errors can occur for **openat**():

**EBADF** *dirfd* is not a valid file descriptor.

**ENOTDIR**

*pathname* is a relative pathname and *dirfd* is a file descriptor

referring to a file other than a directory.

**VERSIONS**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**openat**() was added to Linux in kernel 2.6.16; library support was

added to glibc in version 2.4.

**CONFORMING TO**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**open**(), **creat**() SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.

**openat**(): POSIX.1-2008.

The **O\_DIRECT**, **O\_NOATIME**, **O\_PATH**, and **O\_TMPFILE** flags are Linux-

specific. One must define **\_GNU\_SOURCE** to obtain their definitions.

The **O\_CLOEXEC**, **O\_DIRECTORY**, and **O\_NOFOLLOW** flags are not specified in

POSIX.1-2001, but are specified in POSIX.1-2008. Since glibc 2.12,

one can obtain their definitions by defining either **\_POSIX\_C\_SOURCE**

with a value greater than or equal to 200809L or **\_XOPEN\_SOURCE** with a

value greater than or equal to 700. In glibc 2.11 and earlier, one

obtains the definitions by defining **\_GNU\_SOURCE**.

As noted in [feature\_test\_macros(7)](http://man7.org/linux/man-pages/man7/feature_test_macros.7.html), feature test macros such as

**\_POSIX\_C\_SOURCE**, **\_XOPEN\_SOURCE**, and **\_GNU\_SOURCE** must be defined

before including *any* header files.

**NOTES**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

Under Linux, the **O\_NONBLOCK** flag indicates that one wants to open but

does not necessarily have the intention to read or write. This is

typically used to open devices in order to get a file descriptor for

use with [ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html).

The (undefined) effect of **O\_RDONLY | O\_TRUNC** varies among

implementations. On many systems the file is actually truncated.

Note that **open**() can open device special files, but **creat**() cannot

create them; use [mknod(2)](http://man7.org/linux/man-pages/man2/mknod.2.html) instead.

If the file is newly created, its *st\_atime*, *st\_ctime*, *st\_mtime* fields

(respectively, time of last access, time of last status change, and

time of last modification; see [stat(2)](http://man7.org/linux/man-pages/man2/stat.2.html)) are set to the current time,

and so are the *st\_ctime* and *st\_mtime* fields of the parent directory.

Otherwise, if the file is modified because of the **O\_TRUNC** flag, its

*st\_ctime* and *st\_mtime* fields are set to the current time.

The files in the */proc/[pid]/fd* directory show the open file

descriptors of the process with the PID *pid*. The files in the

*/proc/[pid]/fdinfo* directory show even more information about these

files descriptors. See [proc(5)](http://man7.org/linux/man-pages/man5/proc.5.html) for further details of both of these

directories.

**Open file descriptions**

The term open file description is the one used by POSIX to refer to

the entries in the system-wide table of open files. In other

contexts, this object is variously also called an "open file object",

a "file handle", an "open file table entry", or—in kernel-developer

parlance—a *struct file*.

When a file descriptor is duplicated (using [dup(2)](http://man7.org/linux/man-pages/man2/dup.2.html) or similar), the

duplicate refers to the same open file description as the original

file descriptor, and the two file descriptors consequently share the

file offset and file status flags. Such sharing can also occur

between processes: a child process created via [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) inherits

duplicates of its parent's file descriptors, and those duplicates

refer to the same open file descriptions.

Each **open**() of a file creates a new open file description; thus,

there may be multiple open file descriptions corresponding to a file

inode.

On Linux, one can use the [kcmp(2)](http://man7.org/linux/man-pages/man2/kcmp.2.html) **KCMP\_FILE** operation to test whether

two file descriptors (in the same process or in two different

processes) refer to the same open file description.

**Synchronized I/O**

The POSIX.1-2008 "synchronized I/O" option specifies different

variants of synchronized I/O, and specifies the **open**() flags **O\_SYNC**,

**O\_DSYNC**, and **O\_RSYNC** for controlling the behavior. Regardless of

whether an implementation supports this option, it must at least

support the use of **O\_SYNC** for regular files.

Linux implements **O\_SYNC** and **O\_DSYNC**, but not **O\_RSYNC**. (Somewhat

incorrectly, glibc defines **O\_RSYNC** to have the same value as **O\_SYNC**.)

**O\_SYNC** provides synchronized I/O *file* integrity completion, meaning

write operations will flush data and all associated metadata to the

underlying hardware. **O\_DSYNC** provides synchronized I/O *data*

integrity completion, meaning write operations will flush data to the

underlying hardware, but will only flush metadata updates that are

required to allow a subsequent read operation to complete

successfully. Data integrity completion can reduce the number of

disk operations that are required for applications that don't need

the guarantees of file integrity completion.

To understand the difference between the two types of completion,

consider two pieces of file metadata: the file last modification

timestamp (*st\_mtime*) and the file length. All write operations will

update the last file modification timestamp, but only writes that add

data to the end of the file will change the file length. The last

modification timestamp is not needed to ensure that a read completes

successfully, but the file length is. Thus, **O\_DSYNC** would only

guarantee to flush updates to the file length metadata (whereas

**O\_SYNC** would also always flush the last modification timestamp

metadata).

Before Linux 2.6.33, Linux implemented only the **O\_SYNC** flag for

**open**(). However, when that flag was specified, most filesystems

actually provided the equivalent of synchronized I/O *data* integrity

completion (i.e., **O\_SYNC** was actually implemented as the equivalent

of **O\_DSYNC**).

Since Linux 2.6.33, proper **O\_SYNC** support is provided. However, to

ensure backward binary compatibility, **O\_DSYNC** was defined with the

same value as the historical **O\_SYNC**, and **O\_SYNC** was defined as a new

(two-bit) flag value that includes the **O\_DSYNC** flag value. This

ensures that applications compiled against new headers get at least

**O\_DSYNC** semantics on pre-2.6.33 kernels.

**C library/kernel differences**

Since version 2.26, the glibc wrapper function for **open**() employs the

**openat**() system call, rather than the kernel's **open**() system call.

For certain architectures, this is also true in glibc versions before

2.26.

**NFS**

There are many infelicities in the protocol underlying NFS, affecting

amongst others **O\_SYNC** and **O\_NDELAY**.

On NFS filesystems with UID mapping enabled, **open**() may return a file

descriptor but, for example, [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) requests are denied with **EACCES**.

This is because the client performs **open**() by checking the

permissions, but UID mapping is performed by the server upon read and

write requests.

**FIFOs**

Opening the read or write end of a FIFO blocks until the other end is

also opened (by another process or thread). See [fifo(7)](http://man7.org/linux/man-pages/man7/fifo.7.html) for further

details.

**File access mode**

Unlike the other values that can be specified in *flags*, the *access*

*mode* values **O\_RDONLY**, **O\_WRONLY**, and **O\_RDWR** do not specify individual

bits. Rather, they define the low order two bits of *flags*, and are

defined respectively as 0, 1, and 2. In other words, the combination

**O\_RDONLY | O\_WRONLY** is a logical error, and certainly does not have

the same meaning as **O\_RDWR**.

Linux reserves the special, nonstandard access mode 3 (binary 11) in

*flags* to mean: check for read and write permission on the file and

return a file descriptor that can't be used for reading or writing.

This nonstandard access mode is used by some Linux drivers to return

a file descriptor that is to be used only for device-specific

[ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html) operations.

**Rationale for openat() and other directory file descriptor APIs**

**openat**() and the other system calls and library functions that take a

directory file descriptor argument (i.e., [execveat(2)](http://man7.org/linux/man-pages/man2/execveat.2.html), [faccessat(2)](http://man7.org/linux/man-pages/man2/faccessat.2.html),

[fanotify\_mark(2)](http://man7.org/linux/man-pages/man2/fanotify_mark.2.html), [fchmodat(2)](http://man7.org/linux/man-pages/man2/fchmodat.2.html), [fchownat(2)](http://man7.org/linux/man-pages/man2/fchownat.2.html), [fstatat(2)](http://man7.org/linux/man-pages/man2/fstatat.2.html), [futimesat(2)](http://man7.org/linux/man-pages/man2/futimesat.2.html),

[linkat(2)](http://man7.org/linux/man-pages/man2/linkat.2.html), [mkdirat(2)](http://man7.org/linux/man-pages/man2/mkdirat.2.html), [mknodat(2)](http://man7.org/linux/man-pages/man2/mknodat.2.html), [name\_to\_handle\_at(2)](http://man7.org/linux/man-pages/man2/name_to_handle_at.2.html),

[readlinkat(2)](http://man7.org/linux/man-pages/man2/readlinkat.2.html), [renameat(2)](http://man7.org/linux/man-pages/man2/renameat.2.html), [statx(2)](http://man7.org/linux/man-pages/man2/statx.2.html), [symlinkat(2)](http://man7.org/linux/man-pages/man2/symlinkat.2.html), [unlinkat(2)](http://man7.org/linux/man-pages/man2/unlinkat.2.html),

[utimensat(2)](http://man7.org/linux/man-pages/man2/utimensat.2.html), [mkfifoat(3)](http://man7.org/linux/man-pages/man3/mkfifoat.3.html), and [scandirat(3)](http://man7.org/linux/man-pages/man3/scandirat.3.html)) address two problems

with the older interfaces that preceded them. Here, the explanation

is in terms of the **openat**() call, but the rationale is analogous for

the other interfaces.

First, **openat**() allows an application to avoid race conditions that

could occur when using **open**() to open files in directories other than

the current working directory. These race conditions result from the

fact that some component of the directory prefix given to **open**()

could be changed in parallel with the call to **open**(). Suppose, for

example, that we wish to create the file *dir1/dir2/xxx.dep* if the

file *dir1/dir2/xxx* exists. The problem is that between the existence

check and the file-creation step, *dir1* or *dir2* (which might be

symbolic links) could be modified to point to a different location.

Such races can be avoided by opening a file descriptor for the target

directory, and then specifying that file descriptor as the *dirfd*

argument of (say) [fstatat(2)](http://man7.org/linux/man-pages/man2/fstatat.2.html) and **openat**(). The use of the *dirfd* file

descriptor also has other benefits:

\* the file descriptor is a stable reference to the directory, even

if the directory is renamed; and

\* the open file descriptor prevents the underlying filesystem from

being dismounted, just as when a process has a current working

directory on a filesystem.

Second, **openat**() allows the implementation of a per-thread "current

working directory", via file descriptor(s) maintained by the

application. (This functionality can also be obtained by tricks

based on the use of */proc/self/fd/*dirfd, but less efficiently.)

**O\_DIRECT**

The **O\_DIRECT** flag may impose alignment restrictions on the length and

address of user-space buffers and the file offset of I/Os. In Linux

alignment restrictions vary by filesystem and kernel version and

might be absent entirely. However there is currently no

filesystem-independent interface for an application to discover these

restrictions for a given file or filesystem. Some filesystems

provide their own interfaces for doing so, for example the

**XFS\_IOC\_DIOINFO** operation in [xfsctl(3)](http://man7.org/linux/man-pages/man3/xfsctl.3.html).

Under Linux 2.4, transfer sizes, and the alignment of the user buffer

and the file offset must all be multiples of the logical block size

of the filesystem. Since Linux 2.6.0, alignment to the logical block

size of the underlying storage (typically 512 bytes) suffices. The

logical block size can be determined using the [ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html) **BLKSSZGET**

operation or from the shell using the command:

blockdev --getss

**O\_DIRECT** I/Os should never be run concurrently with the [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) sys‐

tem call, if the memory buffer is a private mapping (i.e., any map‐

ping created with the [mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html) **MAP\_PRIVATE** flag; this includes memory

allocated on the heap and statically allocated buffers). Any such

I/Os, whether submitted via an asynchronous I/O interface or from

another thread in the process, should be completed before [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) is

called. Failure to do so can result in data corruption and undefined

behavior in parent and child processes. This restriction does not

apply when the memory buffer for the **O\_DIRECT** I/Os was created using

[shmat(2)](http://man7.org/linux/man-pages/man2/shmat.2.html) or [mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html) with the **MAP\_SHARED** flag. Nor does this restric‐

tion apply when the memory buffer has been advised as **MADV\_DONTFORK**

with [madvise(2)](http://man7.org/linux/man-pages/man2/madvise.2.html), ensuring that it will not be available to the child

after [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html).

The **O\_DIRECT** flag was introduced in SGI IRIX, where it has alignment

restrictions similar to those of Linux 2.4. IRIX has also a [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)

call to query appropriate alignments, and sizes. FreeBSD 4.x intro‐

duced a flag of the same name, but without alignment restrictions.

**O\_DIRECT** support was added under Linux in kernel version 2.4.10.

Older Linux kernels simply ignore this flag. Some filesystems may

not implement the flag, in which case **open**() fails with the error

**EINVAL** if it is used.

Applications should avoid mixing **O\_DIRECT** and normal I/O to the same

file, and especially to overlapping byte regions in the same file.

Even when the filesystem correctly handles the coherency issues in

this situation, overall I/O throughput is likely to be slower than

using either mode alone. Likewise, applications should avoid mixing

[mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html) of files with direct I/O to the same files.

The behavior of **O\_DIRECT** with NFS will differ from local filesystems.

Older kernels, or kernels configured in certain ways, may not support

this combination. The NFS protocol does not support passing the flag

to the server, so **O\_DIRECT** I/O will bypass the page cache only on the

client; the server may still cache the I/O. The client asks the

server to make the I/O synchronous to preserve the synchronous seman‐

tics of **O\_DIRECT**. Some servers will perform poorly under these cir‐

cumstances, especially if the I/O size is small. Some servers may

also be configured to lie to clients about the I/O having reached

stable storage; this will avoid the performance penalty at some risk

to data integrity in the event of server power failure. The Linux

NFS client places no alignment restrictions on **O\_DIRECT** I/O.

In summary, **O\_DIRECT** is a potentially powerful tool that should be

used with caution. It is recommended that applications treat use of

**O\_DIRECT** as a performance option which is disabled by default.

"The thing that has always disturbed me about O\_DIRECT is that

the whole interface is just stupid, and was probably designed

by a deranged monkey on some serious mind-controlling sub‐

stances."—Linus

**BUGS**[**top**](http://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

Currently, it is not possible to enable signal-driven I/O by

specifying **O\_ASYNC** when calling **open**(); use [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) to enable this

flag.

One must check for two different error codes, **EISDIR** and **ENOENT**, when

trying to determine whether the kernel supports **O\_TMPFILE**

functionality.

When both **O\_CREAT** and **O\_DIRECTORY** are specified in *flags* and the file

specified by *pathname* does not exist, **open**() will create a regular

file (i.e., **O\_DIRECTORY** is ignored).

**SIGNAL(7) Linux Programmer's Manual SIGNAL(7)**

## NAME         [top](http://man7.org/linux/man-pages/man7/signal.7.html#top_of_page)

signal - overview of signals

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man7/signal.7.html#top_of_page)

Linux supports both POSIX reliable signals (hereinafter "standard

signals") and POSIX real-time signals.

**Signal dispositions**

Each signal has a current *disposition*, which determines how the

process behaves when it is delivered the signal.

The entries in the "Action" column of the tables below specify the

default disposition for each signal, as follows:

Term Default action is to terminate the process.

Ign Default action is to ignore the signal.

Core Default action is to terminate the process and dump core (see

[core(5)](http://man7.org/linux/man-pages/man5/core.5.html)).

Stop Default action is to stop the process.

Cont Default action is to continue the process if it is currently

stopped.

A process can change the disposition of a signal using [sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html)

or [signal(2)](http://man7.org/linux/man-pages/man2/signal.2.html). (The latter is less portable when establishing a

signal handler; see [signal(2)](http://man7.org/linux/man-pages/man2/signal.2.html) for details.) Using these system

calls, a process can elect one of the following behaviors to occur on

delivery of the signal: perform the default action; ignore the

signal; or catch the signal with a *signal handler*, a programmer-

defined function that is automatically invoked when the signal is

delivered. (By default, the signal handler is invoked on the normal

process stack. It is possible to arrange that the signal handler

uses an alternate stack; see [sigaltstack(2)](http://man7.org/linux/man-pages/man2/sigaltstack.2.html) for a discussion of how

to do this and when it might be useful.)

The signal disposition is a per-process attribute: in a multithreaded

application, the disposition of a particular signal is the same for

all threads.

A child created via [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) inherits a copy of its parent's signal

dispositions. During an [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html), the dispositions of handled

signals are reset to the default; the dispositions of ignored signals

are left unchanged.

**Sending a signal**

The following system calls and library functions allow the caller to

send a signal:

[raise(3)](http://man7.org/linux/man-pages/man3/raise.3.html) Sends a signal to the calling thread.

[kill(2)](http://man7.org/linux/man-pages/man2/kill.2.html) Sends a signal to a specified process, to all members

of a specified process group, or to all processes on

the system.

[killpg(3)](http://man7.org/linux/man-pages/man3/killpg.3.html) Sends a signal to all of the members of a specified

process group.

[pthread\_kill(3)](http://man7.org/linux/man-pages/man3/pthread_kill.3.html) Sends a signal to a specified POSIX thread in the

same process as the caller.

[tgkill(2)](http://man7.org/linux/man-pages/man2/tgkill.2.html) Sends a signal to a specified thread within a

specific process. (This is the system call used to

implement [pthread\_kill(3)](http://man7.org/linux/man-pages/man3/pthread_kill.3.html).)

[sigqueue(3)](http://man7.org/linux/man-pages/man3/sigqueue.3.html) Sends a real-time signal with accompanying data to a

specified process.

**Waiting for a signal to be caught**

The following system calls suspend execution of the calling process

or thread until a signal is caught (or an unhandled signal terminates

the process):

[pause(2)](http://man7.org/linux/man-pages/man2/pause.2.html) Suspends execution until any signal is caught.

[sigsuspend(2)](http://man7.org/linux/man-pages/man2/sigsuspend.2.html) Temporarily changes the signal mask (see below) and

suspends execution until one of the unmasked signals

is caught.

**Synchronously accepting a signal**

Rather than asynchronously catching a signal via a signal handler, it

is possible to synchronously accept the signal, that is, to block

execution until the signal is delivered, at which point the kernel

returns information about the signal to the caller. There are two

general ways to do this:

\* [sigwaitinfo(2)](http://man7.org/linux/man-pages/man2/sigwaitinfo.2.html), [sigtimedwait(2)](http://man7.org/linux/man-pages/man2/sigtimedwait.2.html), and [sigwait(3)](http://man7.org/linux/man-pages/man3/sigwait.3.html) suspend execution

until one of the signals in a specified set is delivered. Each of

these calls returns information about the delivered signal.

\* [signalfd(2)](http://man7.org/linux/man-pages/man2/signalfd.2.html) returns a file descriptor that can be used to read

information about signals that are delivered to the caller. Each

[read(2)](http://man7.org/linux/man-pages/man2/read.2.html) from this file descriptor blocks until one of the signals

in the set specified in the [signalfd(2)](http://man7.org/linux/man-pages/man2/signalfd.2.html) call is delivered to the

caller. The buffer returned by [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) contains a structure

describing the signal.

**Signal mask and pending signals**

A signal may be *blocked*, which means that it will not be delivered

until it is later unblocked. Between the time when it is generated

and when it is delivered a signal is said to be *pending*.

Each thread in a process has an independent *signal mask*, which

indicates the set of signals that the thread is currently blocking.

A thread can manipulate its signal mask using [pthread\_sigmask(3)](http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html). In

a traditional single-threaded application, [sigprocmask(2)](http://man7.org/linux/man-pages/man2/sigprocmask.2.html) can be used

to manipulate the signal mask.

A child created via [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) inherits a copy of its parent's signal

mask; the signal mask is preserved across [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html).

A signal may be generated (and thus pending) for a process as a whole

(e.g., when sent using [kill(2)](http://man7.org/linux/man-pages/man2/kill.2.html)) or for a specific thread (e.g.,

certain signals, such as **SIGSEGV** and **SIGFPE**, generated as a

consequence of executing a specific machine-language instruction are

thread directed, as are signals targeted at a specific thread using

[pthread\_kill(3)](http://man7.org/linux/man-pages/man3/pthread_kill.3.html)). A process-directed signal may be delivered to any

one of the threads that does not currently have the signal blocked.

If more than one of the threads has the signal unblocked, then the

kernel chooses an arbitrary thread to which to deliver the signal.

A thread can obtain the set of signals that it currently has pending

using [sigpending(2)](http://man7.org/linux/man-pages/man2/sigpending.2.html). This set will consist of the union of the set

of pending process-directed signals and the set of signals pending

for the calling thread.

A child created via [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) initially has an empty pending signal

set; the pending signal set is preserved across an [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html).

**Standard signals**

Linux supports the standard signals listed below. Several signal

numbers are architecture-dependent, as indicated in the "Value"

column. (Where three values are given, the first one is usually

valid for alpha and sparc, the middle one for x86, arm, and most

other architectures, and the last one for mips. (Values for parisc

are *not* shown; see the Linux kernel source for signal numbering on

that architecture.) A dash (-) denotes that a signal is absent on

the corresponding architecture.

First the signals described in the original POSIX.1-1990 standard.

Signal Value Action Comment

──────────────────────────────────────────────────────────────────────

**SIGHUP** 1 Term Hangup detected on controlling terminal

or death of controlling process

**SIGINT** 2 Term Interrupt from keyboard

**SIGQUIT** 3 Core Quit from keyboard

**SIGILL** 4 Core Illegal Instruction

**SIGABRT** 6 Core Abort signal from [abort(3)](http://man7.org/linux/man-pages/man3/abort.3.html)

**SIGFPE** 8 Core Floating-point exception

**SIGKILL** 9 Term Kill signal

**SIGSEGV** 11 Core Invalid memory reference

**SIGPIPE** 13 Term Broken pipe: write to pipe with no

readers; see [pipe(7)](http://man7.org/linux/man-pages/man7/pipe.7.html)

**SIGALRM** 14 Term Timer signal from [alarm(2)](http://man7.org/linux/man-pages/man2/alarm.2.html)

**SIGTERM** 15 Term Termination signal

**SIGUSR1** 30,10,16 Term User-defined signal 1

**SIGUSR2** 31,12,17 Term User-defined signal 2

**SIGCHLD** 20,17,18 Ign Child stopped or terminated

**SIGCONT** 19,18,25 Cont Continue if stopped

**SIGSTOP** 17,19,23 Stop Stop process

**SIGTSTP** 18,20,24 Stop Stop typed at terminal

**SIGTTIN** 21,21,26 Stop Terminal input for background process

**SIGTTOU** 22,22,27 Stop Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or

ignored.

Next the signals not in the POSIX.1-1990 standard but described in

SUSv2 and POSIX.1-2001.

Signal Value Action Comment

────────────────────────────────────────────────────────────────────

**SIGBUS** 10,7,10 Core Bus error (bad memory access)

**SIGPOLL** Term Pollable event (Sys V).

Synonym for **SIGIO**

**SIGPROF** 27,27,29 Term Profiling timer expired

**SIGSYS** 12,31,12 Core Bad system call (SVr4);

see also [seccomp(2)](http://man7.org/linux/man-pages/man2/seccomp.2.html)

**SIGTRAP** 5 Core Trace/breakpoint trap

**SIGURG** 16,23,21 Ign Urgent condition on socket (4.2BSD)

**SIGVTALRM** 26,26,28 Term Virtual alarm clock (4.2BSD)

**SIGXCPU** 24,24,30 Core CPU time limit exceeded (4.2BSD);

see [setrlimit(2)](http://man7.org/linux/man-pages/man2/setrlimit.2.html)

**SIGXFSZ** 25,25,31 Core File size limit exceeded (4.2BSD);

see [setrlimit(2)](http://man7.org/linux/man-pages/man2/setrlimit.2.html)

Up to and including Linux 2.2, the default behavior for **SIGSYS**,

**SIGXCPU**, **SIGXFSZ**, and (on architectures other than SPARC and MIPS)

**SIGBUS** was to terminate the process (without a core dump). (On some

other UNIX systems the default action for **SIGXCPU** and **SIGXFSZ** is to

terminate the process without a core dump.) Linux 2.4 conforms to

the POSIX.1-2001 requirements for these signals, terminating the

process with a core dump.

Next various other signals.

Signal Value Action Comment

────────────────────────────────────────────────────────────────────

**SIGIOT** 6 Core IOT trap. A synonym for **SIGABRT**

**SIGEMT** 7,-,7 Term Emulator trap

**SIGSTKFLT** -,16,- Term Stack fault on coprocessor (unused)

**SIGIO** 23,29,22 Term I/O now possible (4.2BSD)

**SIGCLD** -,-,18 Ign A synonym for **SIGCHLD**

**SIGPWR** 29,30,19 Term Power failure (System V)

**SIGINFO** 29,-,- A synonym for **SIGPWR**

**SIGLOST** -,-,- Term File lock lost (unused)

**SIGWINCH** 28,28,20 Ign Window resize signal (4.3BSD, Sun)

**SIGUNUSED** -,31,- Core Synonymous with **SIGSYS**

(Signal 29 is **SIGINFO** / **SIGPWR** on an alpha but **SIGLOST** on a sparc.)

**SIGEMT** is not specified in POSIX.1-2001, but nevertheless appears on

most other UNIX systems, where its default action is typically to

terminate the process with a core dump.

**SIGPWR** (which is not specified in POSIX.1-2001) is typically ignored

by default on those other UNIX systems where it appears.

**SIGIO** (which is not specified in POSIX.1-2001) is ignored by default

on several other UNIX systems.

Where defined, **SIGUNUSED** is synonymous with **SIGSYS** on most

architectures. Since glibc 2.26, **SIGUNUSED** is no longer defined on

any architecture.

**Real-time signals**

Starting with version 2.2, Linux supports real-time signals as

originally defined in the POSIX.1b real-time extensions (and now

included in POSIX.1-2001). The range of supported real-time signals

is defined by the macros **SIGRTMIN** and **SIGRTMAX**. POSIX.1-2001

requires that an implementation support at least **\_POSIX\_RTSIG\_MAX** (8)

real-time signals.

The Linux kernel supports a range of 33 different real-time signals,

numbered 32 to 64. However, the glibc POSIX threads implementation

internally uses two (for NPTL) or three (for LinuxThreads) real-time

signals (see [pthreads(7)](http://man7.org/linux/man-pages/man7/pthreads.7.html)), and adjusts the value of **SIGRTMIN** suitably

(to 34 or 35). Because the range of available real-time signals

varies according to the glibc threading implementation (and this

variation can occur at run time according to the available kernel and

glibc), and indeed the range of real-time signals varies across UNIX

systems, programs should *never refer to real-time signals using hard-*

*coded numbers*, but instead should always refer to real-time signals

using the notation **SIGRTMIN**+n, and include suitable (run-time) checks

that **SIGRTMIN**+n does not exceed **SIGRTMAX**.

Unlike standard signals, real-time signals have no predefined

meanings: the entire set of real-time signals can be used for

application-defined purposes.

The default action for an unhandled real-time signal is to terminate

the receiving process.

Real-time signals are distinguished by the following:

1. Multiple instances of real-time signals can be queued. By

contrast, if multiple instances of a standard signal are

delivered while that signal is currently blocked, then only one

instance is queued.

2. If the signal is sent using [sigqueue(3)](http://man7.org/linux/man-pages/man3/sigqueue.3.html), an accompanying value

(either an integer or a pointer) can be sent with the signal. If

the receiving process establishes a handler for this signal using

the **SA\_SIGINFO** flag to [sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html), then it can obtain this data

via the *si\_value* field of the *siginfo\_t* structure passed as the

second argument to the handler. Furthermore, the *si\_pid* and

*si\_uid* fields of this structure can be used to obtain the PID and

real user ID of the process sending the signal.

3. Real-time signals are delivered in a guaranteed order. Multiple

real-time signals of the same type are delivered in the order

they were sent. If different real-time signals are sent to a

process, they are delivered starting with the lowest-numbered

signal. (I.e., low-numbered signals have highest priority.) By

contrast, if multiple standard signals are pending for a process,

the order in which they are delivered is unspecified.

If both standard and real-time signals are pending for a process,

POSIX leaves it unspecified which is delivered first. Linux, like

many other implementations, gives priority to standard signals in

this case.

According to POSIX, an implementation should permit at least

**\_POSIX\_SIGQUEUE\_MAX** (32) real-time signals to be queued to a process.

However, Linux does things differently. In kernels up to and

including 2.6.7, Linux imposes a system-wide limit on the number of

queued real-time signals for all processes. This limit can be viewed

and (with privilege) changed via the */proc/sys/kernel/rtsig-max* file.

A related file, */proc/sys/kernel/rtsig-nr*, can be used to find out

how many real-time signals are currently queued. In Linux 2.6.8,

these */proc* interfaces were replaced by the **RLIMIT\_SIGPENDING**

resource limit, which specifies a per-user limit for queued signals;

see [setrlimit(2)](http://man7.org/linux/man-pages/man2/setrlimit.2.html) for further details.

The addition of real-time signals required the widening of the signal

set structure (*sigset\_t*) from 32 to 64 bits. Consequently, various

system calls were superseded by new system calls that supported the

larger signal sets. The old and new system calls are as follows:

**Linux 2.0 and earlier Linux 2.2 and later**

[sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html) [rt\_sigaction(2)](http://man7.org/linux/man-pages/man2/rt_sigaction.2.html)

[sigpending(2)](http://man7.org/linux/man-pages/man2/sigpending.2.html) [rt\_sigpending(2)](http://man7.org/linux/man-pages/man2/rt_sigpending.2.html)

[sigprocmask(2)](http://man7.org/linux/man-pages/man2/sigprocmask.2.html) [rt\_sigprocmask(2)](http://man7.org/linux/man-pages/man2/rt_sigprocmask.2.html)

[sigreturn(2)](http://man7.org/linux/man-pages/man2/sigreturn.2.html) [rt\_sigreturn(2)](http://man7.org/linux/man-pages/man2/rt_sigreturn.2.html)

[sigsuspend(2)](http://man7.org/linux/man-pages/man2/sigsuspend.2.html) [rt\_sigsuspend(2)](http://man7.org/linux/man-pages/man2/rt_sigsuspend.2.html)

[sigtimedwait(2)](http://man7.org/linux/man-pages/man2/sigtimedwait.2.html) [rt\_sigtimedwait(2)](http://man7.org/linux/man-pages/man2/rt_sigtimedwait.2.html)

**Interruption of system calls and library functions by signal handlers**

If a signal handler is invoked while a system call or library

function call is blocked, then either:

\* the call is automatically restarted after the signal handler

returns; or

\* the call fails with the error **EINTR**.

Which of these two behaviors occurs depends on the interface and

whether or not the signal handler was established using the

**SA\_RESTART** flag (see [sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html)). The details vary across UNIX

systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted

by a signal handler, then the call is automatically restarted after

the signal handler returns if the **SA\_RESTART** flag was used; otherwise

the call fails with the error **EINTR**:

\* [read(2)](http://man7.org/linux/man-pages/man2/read.2.html), [readv(2)](http://man7.org/linux/man-pages/man2/readv.2.html), [write(2)](http://man7.org/linux/man-pages/man2/write.2.html), [writev(2)](http://man7.org/linux/man-pages/man2/writev.2.html), and [ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html) calls on

"slow" devices. A "slow" device is one where the I/O call may

block for an indefinite time, for example, a terminal, pipe, or

socket. If an I/O call on a slow device has already transferred

some data by the time it is interrupted by a signal handler, then

the call will return a success status (normally, the number of

bytes transferred). Note that a (local) disk is not a slow device

according to this definition; I/O operations on disk devices are

not interrupted by signals.

\* [open(2)](http://man7.org/linux/man-pages/man2/open.2.html), if it can block (e.g., when opening a FIFO; see [fifo(7)](http://man7.org/linux/man-pages/man7/fifo.7.html)).

\* [wait(2)](http://man7.org/linux/man-pages/man2/wait.2.html), [wait3(2)](http://man7.org/linux/man-pages/man2/wait3.2.html), [wait4(2)](http://man7.org/linux/man-pages/man2/wait4.2.html), [waitid(2)](http://man7.org/linux/man-pages/man2/waitid.2.html), and [waitpid(2)](http://man7.org/linux/man-pages/man2/waitpid.2.html).

\* Socket interfaces: [accept(2)](http://man7.org/linux/man-pages/man2/accept.2.html), [connect(2)](http://man7.org/linux/man-pages/man2/connect.2.html), [recv(2)](http://man7.org/linux/man-pages/man2/recv.2.html), [recvfrom(2)](http://man7.org/linux/man-pages/man2/recvfrom.2.html),

[recvmmsg(2)](http://man7.org/linux/man-pages/man2/recvmmsg.2.html), [recvmsg(2)](http://man7.org/linux/man-pages/man2/recvmsg.2.html), [send(2)](http://man7.org/linux/man-pages/man2/send.2.html), [sendto(2)](http://man7.org/linux/man-pages/man2/sendto.2.html), and [sendmsg(2)](http://man7.org/linux/man-pages/man2/sendmsg.2.html), unless

a timeout has been set on the socket (see below).

\* File locking interfaces: [flock(2)](http://man7.org/linux/man-pages/man2/flock.2.html) and the **F\_SETLKW** and **F\_OFD\_SETLKW**

operations of [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)

\* POSIX message queue interfaces: [mq\_receive(3)](http://man7.org/linux/man-pages/man3/mq_receive.3.html), [mq\_timedreceive(3)](http://man7.org/linux/man-pages/man3/mq_timedreceive.3.html),

[mq\_send(3)](http://man7.org/linux/man-pages/man3/mq_send.3.html), and [mq\_timedsend(3)](http://man7.org/linux/man-pages/man3/mq_timedsend.3.html).

\* [futex(2)](http://man7.org/linux/man-pages/man2/futex.2.html) **FUTEX\_WAIT** (since Linux 2.6.22; beforehand, always failed

with **EINTR**).

\* [getrandom(2)](http://man7.org/linux/man-pages/man2/getrandom.2.html).

\* **pthread\_mutex\_lock**(3), **pthread\_cond\_wait**(3), and related APIs.

\* [futex(2)](http://man7.org/linux/man-pages/man2/futex.2.html) **FUTEX\_WAIT\_BITSET**.

\* POSIX semaphore interfaces: [sem\_wait(3)](http://man7.org/linux/man-pages/man3/sem_wait.3.html) and [sem\_timedwait(3)](http://man7.org/linux/man-pages/man3/sem_timedwait.3.html) (since

Linux 2.6.22; beforehand, always failed with **EINTR**).

\* [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) from an [inotify(7)](http://man7.org/linux/man-pages/man7/inotify.7.html) file descriptor (since Linux 3.8;

beforehand, always failed with **EINTR**).

The following interfaces are never restarted after being interrupted

by a signal handler, regardless of the use of **SA\_RESTART**; they always

fail with the error **EINTR** when interrupted by a signal handler:

\* "Input" socket interfaces, when a timeout (**SO\_RCVTIMEO**) has been

set on the socket using [setsockopt(2)](http://man7.org/linux/man-pages/man2/setsockopt.2.html): [accept(2)](http://man7.org/linux/man-pages/man2/accept.2.html), [recv(2)](http://man7.org/linux/man-pages/man2/recv.2.html),

[recvfrom(2)](http://man7.org/linux/man-pages/man2/recvfrom.2.html), [recvmmsg(2)](http://man7.org/linux/man-pages/man2/recvmmsg.2.html) (also with a non-NULL *timeout* argument),

and [recvmsg(2)](http://man7.org/linux/man-pages/man2/recvmsg.2.html).

\* "Output" socket interfaces, when a timeout (**SO\_RCVTIMEO**) has been

set on the socket using [setsockopt(2)](http://man7.org/linux/man-pages/man2/setsockopt.2.html): [connect(2)](http://man7.org/linux/man-pages/man2/connect.2.html), [send(2)](http://man7.org/linux/man-pages/man2/send.2.html),

[sendto(2)](http://man7.org/linux/man-pages/man2/sendto.2.html), and [sendmsg(2)](http://man7.org/linux/man-pages/man2/sendmsg.2.html).

\* Interfaces used to wait for signals: [pause(2)](http://man7.org/linux/man-pages/man2/pause.2.html), [sigsuspend(2)](http://man7.org/linux/man-pages/man2/sigsuspend.2.html),

[sigtimedwait(2)](http://man7.org/linux/man-pages/man2/sigtimedwait.2.html), and [sigwaitinfo(2)](http://man7.org/linux/man-pages/man2/sigwaitinfo.2.html).

\* File descriptor multiplexing interfaces: [epoll\_wait(2)](http://man7.org/linux/man-pages/man2/epoll_wait.2.html),

[epoll\_pwait(2)](http://man7.org/linux/man-pages/man2/epoll_pwait.2.html), [poll(2)](http://man7.org/linux/man-pages/man2/poll.2.html), [ppoll(2)](http://man7.org/linux/man-pages/man2/ppoll.2.html), [select(2)](http://man7.org/linux/man-pages/man2/select.2.html), and [pselect(2)](http://man7.org/linux/man-pages/man2/pselect.2.html).

\* System V IPC interfaces: [msgrcv(2)](http://man7.org/linux/man-pages/man2/msgrcv.2.html), [msgsnd(2)](http://man7.org/linux/man-pages/man2/msgsnd.2.html), [semop(2)](http://man7.org/linux/man-pages/man2/semop.2.html), and

[semtimedop(2)](http://man7.org/linux/man-pages/man2/semtimedop.2.html).

\* Sleep interfaces: [clock\_nanosleep(2)](http://man7.org/linux/man-pages/man2/clock_nanosleep.2.html), [nanosleep(2)](http://man7.org/linux/man-pages/man2/nanosleep.2.html), and [usleep(3)](http://man7.org/linux/man-pages/man3/usleep.3.html).

\* [io\_getevents(2)](http://man7.org/linux/man-pages/man2/io_getevents.2.html).

The [sleep(3)](http://man7.org/linux/man-pages/man3/sleep.3.html) function is also never restarted if interrupted by a

handler, but gives a success return: the number of seconds remaining

to sleep.

**Interruption of system calls and library functions by stop signals**

On Linux, even in the absence of signal handlers, certain blocking

interfaces can fail with the error **EINTR** after the process is stopped

by one of the stop signals and then resumed via **SIGCONT**. This

behavior is not sanctioned by POSIX.1, and doesn't occur on other

systems.

**OUTB(2) Linux Programmer's Manual OUTB(2)**

## NAME         [top](http://man7.org/linux/man-pages/man2/inl.2.html#top_of_page)

outb, outw, outl, outsb, outsw, outsl, inb, inw, inl, insb, insw,

insl, outb\_p, outw\_p, outl\_p, inb\_p, inw\_p, inl\_p - port I/O

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man2/inl.2.html#top_of_page)

**#include <sys/io.h>**

**unsigned char inb(unsigned short int** *port***);**

**unsigned char inb\_p(unsigned short int** *port***);**

**unsigned short int inw(unsigned short int** *port***);**

**unsigned short int inw\_p(unsigned short int** *port***);**

**unsigned int inl(unsigned short int** *port***);**

**unsigned int inl\_p(unsigned short int** *port***);**

**void outb(unsigned char** *value***, unsigned short int** *port***);**

**void outb\_p(unsigned char** *value***, unsigned short int** *port***);**

**void outw(unsigned short int** *value***, unsigned short int** *port***);**

**void outw\_p(unsigned short int** *value***, unsigned short int** *port***);**

**void outl(unsigned int** *value***, unsigned short int** *port***);**

**void outl\_p(unsigned int** *value***, unsigned short int** *port***);**

**void insb(unsigned short int** *port***, void \****addr***,**

**unsigned long int** *count***);**

**void insw(unsigned short int** *port***, void \****addr***,**

**unsigned long int** *count***);**

**void insl(unsigned short int** *port***, void \****addr***,**

**unsigned long int** *count***);**

**void outsb(unsigned short int** *port***, const void \****addr***,**

**unsigned long int** *count***);**

**void outsw(unsigned short int** *port***, const void \****addr***,**

**unsigned long int** *count***);**

**void outsl(unsigned short int** *port***, const void \****addr***,**

**unsigned long int** *count***);**

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man2/inl.2.html#top_of_page)

This family of functions is used to do low-level port input and

output. The out\* functions do port output, the in\* functions do port

input; the b-suffix functions are byte-width and the w-suffix

functions word-width; the \_p-suffix functions pause until the I/O

completes.

They are primarily designed for internal kernel use, but can be used

from user space.

You must compile with **-O** or **-O2** or similar. The functions are

defined as inline macros, and will not be substituted in without

optimization enabled, causing unresolved references at link time.

You use [ioperm(2)](http://man7.org/linux/man-pages/man2/ioperm.2.html) or alternatively [iopl(2)](http://man7.org/linux/man-pages/man2/iopl.2.html) to tell the kernel to

allow the user space application to access the I/O ports in question.

Failure to do this will cause the application to receive a

segmentation fault.

## CONFORMING TO         [top](http://man7.org/linux/man-pages/man2/inl.2.html#top_of_page)

**outb**() and friends are hardware-specific. The *value* argument is

passed first and the *port* argument is passed second, which is the

opposite order from most DOS implementations.

**EXEC(3) Linux Programmer's Manual EXEC(3)**

## NAME         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

execl, execlp, execle, execv, execvp, execvpe - execute a file

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

**#include <unistd.h>**

**extern char \*\*environ;**

**int execl(const char \****path***, const char \****arg***, ...**

**/\* (char \*) NULL \*/);**

**int execlp(const char \****file***, const char \****arg***, ...**

**/\* (char \*) NULL \*/);**

**int execle(const char \****path***, const char \****arg***, ...**

**/\*, (char \*) NULL, char \* const** *envp***[] \*/);**

**int execv(const char \****path***, char \*const** *argv***[]);**

**int execvp(const char \****file***, char \*const** *argv***[]);**

**int execvpe(const char \****file***, char \*const** *argv***[],**

**char \*const** *envp***[]);**

Feature Test Macro Requirements for glibc (see [feature\_test\_macros(7)](http://man7.org/linux/man-pages/man7/feature_test_macros.7.html)):

**execvpe**(): \_GNU\_SOURCE

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

The **exec**() family of functions replaces the current process image

with a new process image. The functions described in this manual

page are front-ends for [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html). (See the manual page for

[execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html) for further details about the replacement of the current

process image.)

The initial argument for these functions is the name of a file that

is to be executed.

The *const char \*arg* and subsequent ellipses in the **execl**(), **execlp**(),

and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*.

Together they describe a list of one or more pointers to null-

terminated strings that represent the argument list available to the

executed program. The first argument, by convention, should point to

the filename associated with the file being executed. The list of

arguments *must* be terminated by a null pointer, and, since these are

variadic functions, this pointer must be cast *(char \*) NULL*.

The **execv**(), **execvp**(), and **execvpe**() functions provide an array of

pointers to null-terminated strings that represent the argument list

available to the new program. The first argument, by convention,

should point to the filename associated with the file being executed.

The array of pointers *must* be terminated by a null pointer.

The **execle**() and **execvpe**() functions allow the caller to specify the

environment of the executed program via the argument *envp*. The *envp*

argument is an array of pointers to null-terminated strings and *must*

be terminated by a null pointer. The other functions take the

environment for the new process image from the external variable

*environ* in the calling process.

**Special semantics for execlp() and execvp()**

The **execlp**(), **execvp**(), and **execvpe**() functions duplicate the actions

of the shell in searching for an executable file if the specified

filename does not contain a slash (/) character. The file is sought

in the colon-separated list of directory pathnames specified in the

**PATH** environment variable. If this variable isn't defined, the path

list defaults to a list that includes the directories returned by

*confstr(\_CS\_PATH)* (which typically returns the value "/bin:/usr/bin")

and possibly also the current working directory; see NOTES for

further details.

If the specified filename includes a slash character, then **PATH** is

ignored, and the file at the specified pathname is executed.

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html) failed

with the error **EACCES**), these functions will continue searching the

rest of the search path. If no other file is found, however, they

will return with [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) set to **EACCES**.

If the header of a file isn't recognized (the attempted [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html)

failed with the error **ENOEXEC**), these functions will execute the

shell (*/bin/sh*) with the path of the file as its first argument. (If

this attempt fails, no further searching is done.)

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

The **exec**() functions return only if an error has occurred. The

return value is -1, and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate the error.

## ERRORS         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

All of these functions may fail and set [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) for any of the errors

specified for [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html).

## VERSIONS         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

The **execvpe**() function first appeared in glibc 2.11.

## ATTRIBUTES         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

For an explanation of the terms used in this section, see

[attributes(7)](http://man7.org/linux/man-pages/man7/attributes.7.html).

┌──────────────────────────────┬───────────────┬─────────────┐

│**Interface** │ **Attribute** │ **Value** │

├──────────────────────────────┼───────────────┼─────────────┤

│**execl**(), **execle**(), **execv**() │ Thread safety │ MT-Safe │

├──────────────────────────────┼───────────────┼─────────────┤

│**execlp**(), **execvp**(), **execvpe**() │ Thread safety │ MT-Safe env │

└──────────────────────────────┴───────────────┴─────────────┘

## NOTES         [top](http://man7.org/linux/man-pages/man3/exec.3.html#top_of_page)

The default search path (used when the environment does not contain

the variable **PATH**) shows some variation across systems. It generally

includes */bin* and */usr/bin* (in that order) and may also include the

current working directory. On some other systems, the current

working is included after */bin* and */usr/bin*, as an anti-Trojan-horse

measure. The glibc implementation long followed the traditional

default where the current working directory is included at the start

of the search path. However, some code refactoring during the

development of glibc 2.24 caused the current working directory to be

dropped altogether from the default search path. This accidental

behavior change is considered mildly beneficial, and won't be

reverted.

The behavior of **execlp**() and **execvp**() when errors occur while

attempting to execute the file is historic practice, but has not

traditionally been documented and is not specified by the POSIX

standard. BSD (and possibly other systems) do an automatic sleep and

retry if **ETXTBSY** is encountered. Linux treats it as a hard error and

returns immediately.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors

except for the ones described above and **ENOMEM** and **E2BIG**, upon which

they returned. They now return if any error other than the ones

described above occurs.

**WAIT(2) Linux Programmer's Manual WAIT(2)**

## NAME         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

wait, waitpid, waitid - wait for process to change state

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

**#include <sys/types.h>**

**#include <sys/wait.h>**

**pid\_t wait(int \****wstatus***);**

**pid\_t waitpid(pid\_t** *pid***, int \****wstatus***, int** *options***);**

**int waitid(idtype\_t** *idtype***, id\_t** *id***, siginfo\_t \****infop***, int** *options***);**

/\* This is the glibc and POSIX interface; see

NOTES for information on the raw system call. \*/

Feature Test Macro Requirements for glibc (see [feature\_test\_macros(7)](http://man7.org/linux/man-pages/man7/feature_test_macros.7.html)):

**waitid**():

Since glibc 2.26: \_XOPEN\_SOURCE >= 500 ||

\_POSIX\_C\_SOURCE >= 200809L

Glibc 2.25 and earlier:

\_XOPEN\_SOURCE

|| /\* Since glibc 2.12: \*/ \_POSIX\_C\_SOURCE >= 200809L

|| /\* Glibc versions <= 2.19: \*/ \_BSD\_SOURCE

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

All of these system calls are used to wait for state changes in a

child of the calling process, and obtain information about the child

whose state has changed. A state change is considered to be: the

child terminated; the child was stopped by a signal; or the child was

resumed by a signal. In the case of a terminated child, performing a

wait allows the system to release the resources associated with the

child; if a wait is not performed, then the terminated child remains

in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return

immediately. Otherwise, they block until either a child changes

state or a signal handler interrupts the call (assuming that system

calls are not automatically restarted using the **SA\_RESTART** flag of

[sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html)). In the remainder of this page, a child whose state

has changed and which has not yet been waited upon by one of these

system calls is termed *waitable*.

**wait() and waitpid()**

The **wait**() system call suspends execution of the calling thread until

one of its children terminates. The call *wait(&wstatus)* is

equivalent to:

waitpid(-1, &wstatus, 0);

The **waitpid**() system call suspends execution of the calling thread

until a child specified by *pid* argument has changed state. By

default, **waitpid**() waits only for terminated children, but this

behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< -1 meaning wait for any child process whose process group ID is

equal to the absolute value of *pid*.

-1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is

equal to that of the calling process.

> 0 meaning wait for the child whose process ID is equal to the

value of *pid*.

The value of *options* is an OR of zero or more of the following con‐

stants:

**WNOHANG** return immediately if no child has exited.

**WUNTRACED** also return if a child has stopped (but not traced via

[ptrace(2)](http://man7.org/linux/man-pages/man2/ptrace.2.html)). Status for *traced* children which have

stopped is provided even if this option is not specified.

**WCONTINUED** (since Linux 2.6.10)

also return if a stopped child has been resumed by deliv‐

ery of **SIGCONT**.

(For Linux-only options, see below.)

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information

in the *int* to which it points. This integer can be inspected with

the following macros (which take the integer itself as an argument,

not a pointer to it, as is done in **wait**() and **waitpid**()!):

**WIFEXITED(***wstatus***)**

returns true if the child terminated normally, that is, by

calling [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html) or [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html), or by returning from main().

**WEXITSTATUS(***wstatus***)**

returns the exit status of the child. This consists of the

least significant 8 bits of the *status* argument that the child

specified in a call to [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html) or [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html) or as the argument

for a return statement in main(). This macro should be

employed only if **WIFEXITED** returned true.

**WIFSIGNALED(***wstatus***)**

returns true if the child process was terminated by a signal.

**WTERMSIG(***wstatus***)**

returns the number of the signal that caused the child process

to terminate. This macro should be employed only if **WIFSIG‐**

**NALED** returned true.

**WCOREDUMP(***wstatus***)**

returns true if the child produced a core dump. This macro

should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not avail‐

able on some UNIX implementations (e.g., AIX, SunOS). There‐

fore, enclose its use inside *#ifdef WCOREDUMP ... #endif*.

**WIFSTOPPED(***wstatus***)**

returns true if the child process was stopped by delivery of a

signal; this is possible only if the call was done using **WUN‐**

**TRACED** or when the child is being traced (see [ptrace(2)](http://man7.org/linux/man-pages/man2/ptrace.2.html)).

**WSTOPSIG(***wstatus***)**

returns the number of the signal which caused the child to

stop. This macro should be employed only if **WIFSTOPPED**

returned true.

**WIFCONTINUED(***wstatus***)**

(since Linux 2.6.10) returns true if the child process was

resumed by delivery of **SIGCONT**.

**waitid()**

The **waitid**() system call (available since Linux 2.6.9) provides more

precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as

follows:

*idtype* == **P\_PID**

Wait for the child whose process ID matches *id*.

*idtype* == **P\_PGID**

Wait for any child whose process group ID matches *id*.

*idtype* == **P\_ALL**

Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or

more of the following flags in *options*:

**WEXITED** Wait for children that have terminated.

**WSTOPPED** Wait for children that have been stopped by delivery of a

signal.

**WCONTINUED** Wait for (previously stopped) children that have been

resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

**WNOHANG** As for **waitpid**().

**WNOWAIT** Leave the child in a waitable state; a later wait call

can be used to again retrieve the child status informa‐

tion.

Upon successful return, **waitid**() fills in the following fields of the

*siginfo\_t* structure pointed to by *infop*:

*si\_pid* The process ID of the child.

*si\_uid* The real user ID of the child. (This field is not set on

most other implementations.)

*si\_signo* Always set to **SIGCHLD**.

*si\_status* Either the exit status of the child, as given to [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html)

(or [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html)), or the signal that caused the child to ter‐

minate, stop, or continue. The *si\_code* field can be used

to determine how to interpret this field.

*si\_code* Set to one of: **CLD\_EXITED** (child called [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html));

**CLD\_KILLED** (child killed by signal); **CLD\_DUMPED** (child

killed by signal, and dumped core); **CLD\_STOPPED** (child

stopped by signal); **CLD\_TRAPPED** (traced child has

trapped); or **CLD\_CONTINUED** (child continued by **SIGCONT**).

If **WNOHANG** was specified in *options* and there were no children in a

waitable state, then **waitid**() returns 0 immediately and the state of

the *siginfo\_t* structure pointed to by *infop* depends on the implemen‐

tation. To (portably) distinguish this case from that where a child

was in a waitable state, zero out the *si\_pid* field before the call

and check for a nonzero value in this field after the call returns.

POSIX.1-2008 Technical Corrigendum 1 (2013) adds the requirement that

when **WNOHANG** is specified in *options* and there were no children in a

waitable state, then **waitid**() should zero out the *si\_pid* and *si\_signo*

fields of the structure. On Linux and other implementations that

adhere to this requirement, it is not necessary to zero out the

*si\_pid* field before calling **waitid**(). However, not all implementa‐

tions follow the POSIX.1 specification on this point.

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

**wait**(): on success, returns the process ID of the terminated child;

on error, -1 is returned.

**waitpid**(): on success, returns the process ID of the child whose

state has changed; if **WNOHANG** was specified and one or more

child(ren) specified by *pid* exist, but have not yet changed state,

then 0 is returned. On error, -1 is returned.

**waitid**(): returns 0 on success or if **WNOHANG** was specified and no

child(ren) specified by *id* has yet changed state; on error, -1 is

returned.

Each of these calls sets [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) to an appropriate value in the case of

an error.

## ERRORS         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

**ECHILD** (for **wait**()) The calling process does not have any unwaited-

for children.

**ECHILD** (for **waitpid**() or **waitid**()) The process specified by *pid*

(**waitpid**()) or *idtype* and *id* (**waitid**()) does not exist or is

not a child of the calling process. (This can happen for

one's own child if the action for **SIGCHLD** is set to **SIG\_IGN**.

See also the *Linux Notes* section about threads.)

**EINTR WNOHANG** was not set and an unblocked signal or a **SIGCHLD** was

caught; see [signal(7)](http://man7.org/linux/man-pages/man7/signal.7.html).

**EINVAL** The *options* argument was invalid.

## CONFORMING TO         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

SVr4, 4.3BSD, POSIX.1-2001.

## NOTES         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

A child that terminates, but has not been waited for becomes a

"zombie". The kernel maintains a minimal set of information about

the zombie process (PID, termination status, resource usage

information) in order to allow the parent to later perform a wait to

obtain information about the child. As long as a zombie is not

removed from the system via a wait, it will consume a slot in the

kernel process table, and if this table fills, it will not be

possible to create further processes. If a parent process

terminates, then its "zombie" children (if any) are adopted by

[init(1)](http://man7.org/linux/man-pages/man1/init.1.html), (or by the nearest "subreaper" process as defined through

the use of the [prctl(2)](http://man7.org/linux/man-pages/man2/prctl.2.html) **PR\_SET\_CHILD\_SUBREAPER** operation); [init(1)](http://man7.org/linux/man-pages/man1/init.1.html)

automatically performs a wait to remove the zombies.

POSIX.1-2001 specifies that if the disposition of **SIGCHLD** is set to

**SIG\_IGN** or the **SA\_NOCLDWAIT** flag is set for **SIGCHLD** (see

[sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html)), then children that terminate do not become zombies and

a call to **wait**() or **waitpid**() will block until all children have

terminated, and then fail with [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) set to **ECHILD**. (The original

POSIX standard left the behavior of setting **SIGCHLD** to **SIG\_IGN**

unspecified. Note that even though the default disposition of

**SIGCHLD** is "ignore", explicitly setting the disposition to **SIG\_IGN**

results in different treatment of zombie process children.)

Linux 2.6 conforms to the POSIX requirements. However, Linux 2.4

(and earlier) does not: if a **wait**() or **waitpid**() call is made while

**SIGCHLD** is being ignored, the call behaves just as though **SIGCHLD**

were not being ignored, that is, the call blocks until the next child

terminates and then returns the process ID and status of that child.

**Linux notes**

In the Linux kernel, a kernel-scheduled thread is not a distinct

construct from a process. Instead, a thread is simply a process that

is created using the Linux-unique [clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html) system call; other

routines such as the portable [pthread\_create(3)](http://man7.org/linux/man-pages/man3/pthread_create.3.html) call are implemented

using [clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html). Before Linux 2.4, a thread was just a special case

of a process, and as a consequence one thread could not wait on the

children of another thread, even when the latter belongs to the same

thread group. However, POSIX prescribes such functionality, and

since Linux 2.4 a thread can, and by default will, wait on children

of other threads in the same thread group.

The following Linux-specific *options* are for use with children

created using [clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html); they can also, since Linux 4.7, be used with

**waitid**():

**\_\_WCLONE**

Wait for "clone" children only. If omitted, then wait for

"non-clone" children only. (A "clone" child is one which

delivers no signal, or a signal other than **SIGCHLD** to its

parent upon termination.) This option is ignored if **\_\_WALL** is

also specified.

**\_\_WALL** (since Linux 2.4)

Wait for all children, regardless of type ("clone" or "non-

clone").

**\_\_WNOTHREAD** (since Linux 2.4)

Do not wait for children of other threads in the same thread

group. This was the default before Linux 2.4.

Since Linux 4.7, the **\_\_WALL** flag is automatically implied if the

child is being ptraced.

**C library/kernel differences**

**wait**() is actually a library function that (in glibc) is implemented

as a call to [wait4(2)](http://man7.org/linux/man-pages/man2/wait4.2.html).

On some architectures, there is no **waitpid**() system call; instead,

this interface is implemented via a C library wrapper function that

calls [wait4(2)](http://man7.org/linux/man-pages/man2/wait4.2.html).

The raw **waitid**() system call takes a fifth argument, of type *struct*

*rusage \**. If this argument is non-NULL, then it is used to return

resource usage information about the child, in the same manner as

[wait4(2)](http://man7.org/linux/man-pages/man2/wait4.2.html). See [getrusage(2)](http://man7.org/linux/man-pages/man2/getrusage.2.html) for details.

## BUGS         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

According to POSIX.1-2008, an application calling **waitid**() must

ensure that *infop* points to a *siginfo\_t* structure (i.e., that it is a

non-null pointer). On Linux, if *infop* is NULL, **waitid**() succeeds,

and returns the process ID of the waited-for child. Applications

should avoid relying on this inconsistent, nonstandard, and

unnecessary feature.

## EXAMPLE         [top](http://man7.org/linux/man-pages/man2/wait.2.html#top_of_page)

The following program demonstrates the use of [fork(2)](http://man7.org/linux/man-pages/man2/fork.2.html) and **waitpid**().

The program creates a child process. If no command-line argument is

supplied to the program, then the child suspends its execution using

[pause(2)](http://man7.org/linux/man-pages/man2/pause.2.html), to allow the user to send signals to the child. Otherwise,

if a command-line argument is supplied, then the child exits

immediately, using the integer supplied on the command line as the

exit status. The parent process executes a loop that monitors the

child using **waitpid**(), and uses the W\*() macros described above to

analyze the wait status value.

The following shell session demonstrates the use of the program:

$ **./a.out &**

Child PID is 32360

[1] 32359

$ **kill -STOP 32360**

stopped by signal 19

$ **kill -CONT 32360**

continued

$ **kill -TERM 32360**

killed by signal 15

[1]+ Done ./a.out

$

**Program source**

#include <sys/wait.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

int

main(int argc, char \*argv[])

{

pid\_t cpid, w;

int wstatus;

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Code executed by child \*/

printf("Child PID is %ld\n", (long) getpid());

if (argc == 1)

pause(); /\* Wait for signals \*/

\_exit(atoi(argv[1]));

} else { /\* Code executed by parent \*/

do {

w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);

if (w == -1) {

perror("waitpid");

exit(EXIT\_FAILURE);

}

if (WIFEXITED(wstatus)) {

printf("exited, status=%d\n", WEXITSTATUS(wstatus));

} else if (WIFSIGNALED(wstatus)) {

printf("killed by signal %d\n", WTERMSIG(wstatus));

} else if (WIFSTOPPED(wstatus)) {

printf("stopped by signal %d\n", WSTOPSIG(wstatus));

} else if (WIFCONTINUED(wstatus)) {

printf("continued\n");

}

} while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));

exit(EXIT\_SUCCESS);

}

}

**PTHREAD\_CREATE(3) Linux Programmer's Manual PTHREAD\_CREATE(3)**

## NAME         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

pthread\_create - create a new thread

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

**#include <pthread.h>**

**int pthread\_create(pthread\_t \****thread***, const pthread\_attr\_t \****attr***,**

**void \*(\****start\_routine***) (void \*), void \****arg***);**

Compile and link with *-pthread*.

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

The **pthread\_create**() function starts a new thread in the calling

process. The new thread starts execution by invoking

*start\_routine*(); *arg* is passed as the sole argument of

*start\_routine*().

The new thread terminates in one of the following ways:

\* It calls [pthread\_exit(3)](http://man7.org/linux/man-pages/man3/pthread_exit.3.html), specifying an exit status value that is

available to another thread in the same process that calls

[pthread\_join(3)](http://man7.org/linux/man-pages/man3/pthread_join.3.html).

\* It returns from *start\_routine*(). This is equivalent to calling

[pthread\_exit(3)](http://man7.org/linux/man-pages/man3/pthread_exit.3.html) with the value supplied in the *return* statement.

\* It is canceled (see [pthread\_cancel(3)](http://man7.org/linux/man-pages/man3/pthread_cancel.3.html)).

\* Any of the threads in the process calls [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html), or the main thread

performs a return from *main*(). This causes the termination of all

threads in the process.

The *attr* argument points to a *pthread\_attr\_t* structure whose contents

are used at thread creation time to determine attributes for the new

thread; this structure is initialized using [pthread\_attr\_init(3)](http://man7.org/linux/man-pages/man3/pthread_attr_init.3.html) and

related functions. If *attr* is NULL, then the thread is created with

default attributes.

Before returning, a successful call to **pthread\_create**() stores the ID

of the new thread in the buffer pointed to by *thread*; this identifier

is used to refer to the thread in subsequent calls to other pthreads

functions.

The new thread inherits a copy of the creating thread's signal mask

([pthread\_sigmask(3)](http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html)). The set of pending signals for the new thread

is empty ([sigpending(2)](http://man7.org/linux/man-pages/man2/sigpending.2.html)). The new thread does not inherit the

creating thread's alternate signal stack ([sigaltstack(2)](http://man7.org/linux/man-pages/man2/sigaltstack.2.html)).

The new thread inherits the calling thread's floating-point

environment ([fenv(3)](http://man7.org/linux/man-pages/man3/fenv.3.html)).

The initial value of the new thread's CPU-time clock is 0 (see

[pthread\_getcpuclockid(3)](http://man7.org/linux/man-pages/man3/pthread_getcpuclockid.3.html)).

**Linux-specific details**

The new thread inherits copies of the calling thread's capability

sets (see [capabilities(7)](http://man7.org/linux/man-pages/man7/capabilities.7.html)) and CPU affinity mask (see

[sched\_setaffinity(2)](http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html)).

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

On success, **pthread\_create**() returns 0; on error, it returns an error

number, and the contents of *\*thread* are undefined.

## ERRORS         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

**EAGAIN** Insufficient resources to create another thread.

**EAGAIN** A system-imposed limit on the number of threads was

encountered. There are a number of limits that may trigger

this error: the **RLIMIT\_NPROC** soft resource limit (set via

[setrlimit(2)](http://man7.org/linux/man-pages/man2/setrlimit.2.html)), which limits the number of processes and

threads for a real user ID, was reached; the kernel's system-

wide limit on the number of processes and threads,

*/proc/sys/kernel/threads-max*, was reached (see [proc(5)](http://man7.org/linux/man-pages/man5/proc.5.html)); or

the maximum number of PIDs, */proc/sys/kernel/pid\_max*, was

reached (see [proc(5)](http://man7.org/linux/man-pages/man5/proc.5.html)).

**EINVAL** Invalid settings in *attr*.

**EPERM** No permission to set the scheduling policy and parameters

specified in *attr*.

## ATTRIBUTES         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

For an explanation of the terms used in this section, see

[attributes(7)](http://man7.org/linux/man-pages/man7/attributes.7.html).

┌─────────────────┬───────────────┬─────────┐

│**Interface** │ **Attribute** │ **Value** │

├─────────────────┼───────────────┼─────────┤

│**pthread\_create**() │ Thread safety │ MT-Safe │

└─────────────────┴───────────────┴─────────┘

## CONFORMING TO         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

POSIX.1-2001, POSIX.1-2008.

## NOTES         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

See [pthread\_self(3)](http://man7.org/linux/man-pages/man3/pthread_self.3.html) for further information on the thread ID returned

in *\*thread* by **pthread\_create**(). Unless real-time scheduling policies

are being employed, after a call to **pthread\_create**(), it is

indeterminate which thread—the caller or the new thread—will next

execute.

A thread may either be *joinable* or *detached*. If a thread is

joinable, then another thread can call [pthread\_join(3)](http://man7.org/linux/man-pages/man3/pthread_join.3.html) to wait for

the thread to terminate and fetch its exit status. Only when a

terminated joinable thread has been joined are the last of its

resources released back to the system. When a detached thread

terminates, its resources are automatically released back to the

system: it is not possible to join with the thread in order to obtain

its exit status. Making a thread detached is useful for some types

of daemon threads whose exit status the application does not need to

care about. By default, a new thread is created in a joinable state,

unless *attr* was set to create the thread in a detached state (using

[pthread\_attr\_setdetachstate(3)](http://man7.org/linux/man-pages/man3/pthread_attr_setdetachstate.3.html)).

Under the NPTL threading implementation, if the **RLIMIT\_STACK** soft

resource limit *at the time the program started* has any value other

than "unlimited", then it determines the default stack size of new

threads. Using [pthread\_attr\_setstacksize(3)](http://man7.org/linux/man-pages/man3/pthread_attr_setstacksize.3.html), the stack size

attribute can be explicitly set in the *attr* argument used to create a

thread, in order to obtain a stack size other than the default. If

the **RLIMIT\_STACK** resource limit is set to "unlimited", a per-

architecture value is used for the stack size. Here is the value for

a few architectures:

┌─────────────┬────────────────────┐

│**Architecture** │ **Default stack size** │

├─────────────┼────────────────────┤

│i386 │ 2 MB │

├─────────────┼────────────────────┤

│IA-64 │ 32 MB │

├─────────────┼────────────────────┤

│PowerPC │ 4 MB │

├─────────────┼────────────────────┤

│S/390 │ 2 MB │

├─────────────┼────────────────────┤

│Sparc-32 │ 2 MB │

├─────────────┼────────────────────┤

│Sparc-64 │ 4 MB │

├─────────────┼────────────────────┤

│x86\_64 │ 2 MB │

└─────────────┴────────────────────┘

## BUGS         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

In the obsolete LinuxThreads implementation, each of the threads in a

process has a different process ID. This is in violation of the

POSIX threads specification, and is the source of many other

nonconformances to the standard; see [pthreads(7)](http://man7.org/linux/man-pages/man7/pthreads.7.html).

## EXAMPLE         [top](http://man7.org/linux/man-pages/man3/pthread_create.3.html#top_of_page)

The program below demonstrates the use of **pthread\_create**(), as well

as a number of other functions in the pthreads API.

In the following run, on a system providing the NPTL threading

implementation, the stack size defaults to the value given by the

"stack size" resource limit:

$ **ulimit -s**

8192 # The stack size limit is 8 MB (0x800000 bytes)

$ **./a.out hola salut servus**

Thread 1: top of stack near 0xb7dd03b8; argv\_string=hola

Thread 2: top of stack near 0xb75cf3b8; argv\_string=salut

Thread 3: top of stack near 0xb6dce3b8; argv\_string=servus

Joined with thread 1; returned value was HOLA

Joined with thread 2; returned value was SALUT

Joined with thread 3; returned value was SERVUS

In the next run, the program explicitly sets a stack size of 1 MB

(using [pthread\_attr\_setstacksize(3)](http://man7.org/linux/man-pages/man3/pthread_attr_setstacksize.3.html)) for the created threads:

$ **./a.out -s 0x100000 hola salut servus**

Thread 1: top of stack near 0xb7d723b8; argv\_string=hola

Thread 2: top of stack near 0xb7c713b8; argv\_string=salut

Thread 3: top of stack near 0xb7b703b8; argv\_string=servus

Joined with thread 1; returned value was HOLA

Joined with thread 2; returned value was SALUT

Joined with thread 3; returned value was SERVUS

**Program source**

#include <pthread.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <ctype.h>

#define handle\_error\_en(en, msg) \

do { errno = en; perror(msg); exit(EXIT\_FAILURE); } while (0)

#define handle\_error(msg) \

do { perror(msg); exit(EXIT\_FAILURE); } while (0)

struct thread\_info { /\* Used as argument to thread\_start() \*/

pthread\_t thread\_id; /\* ID returned by pthread\_create() \*/

int thread\_num; /\* Application-defined thread # \*/

char \*argv\_string; /\* From command-line argument \*/

};

/\* Thread start function: display address near top of our stack,

and return upper-cased copy of argv\_string \*/

static void \*

thread\_start(void \*arg)

{

struct thread\_info \*tinfo = arg;

char \*uargv, \*p;

printf("Thread %d: top of stack near %p; argv\_string=%s\n",

tinfo->thread\_num, &p, tinfo->argv\_string);

uargv = strdup(tinfo->argv\_string);

if (uargv == NULL)

handle\_error("strdup");

for (p = uargv; \*p != '\0'; p++)

\*p = toupper(\*p);

return uargv;

}

int

main(int argc, char \*argv[])

{

int s, tnum, opt, num\_threads;

struct thread\_info \*tinfo;

pthread\_attr\_t attr;

int stack\_size;

void \*res;

/\* The "-s" option specifies a stack size for our threads \*/

stack\_size = -1;

while ((opt = getopt(argc, argv, "s:")) != -1) {

switch (opt) {

case 's':

stack\_size = strtoul(optarg, NULL, 0);

break;

default:

fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",

argv[0]);

exit(EXIT\_FAILURE);

}

}

num\_threads = argc - optind;

/\* Initialize thread creation attributes \*/

s = pthread\_attr\_init(&attr);

if (s != 0)

handle\_error\_en(s, "pthread\_attr\_init");

if (stack\_size > 0) {

s = pthread\_attr\_setstacksize(&attr, stack\_size);

if (s != 0)

handle\_error\_en(s, "pthread\_attr\_setstacksize");

}

/\* Allocate memory for pthread\_create() arguments \*/

tinfo = calloc(num\_threads, sizeof(struct thread\_info));

if (tinfo == NULL)

handle\_error("calloc");

/\* Create one thread for each command-line argument \*/

for (tnum = 0; tnum < num\_threads; tnum++) {

tinfo[tnum].thread\_num = tnum + 1;

tinfo[tnum].argv\_string = argv[optind + tnum];

/\* The pthread\_create() call stores the thread ID into

corresponding element of tinfo[] \*/

s = pthread\_create(&tinfo[tnum].thread\_id, &attr,

&thread\_start, &tinfo[tnum]);

if (s != 0)

handle\_error\_en(s, "pthread\_create");

}

/\* Destroy the thread attributes object, since it is no

longer needed \*/

s = pthread\_attr\_destroy(&attr);

if (s != 0)

handle\_error\_en(s, "pthread\_attr\_destroy");

/\* Now join with each thread, and display its returned value \*/

for (tnum = 0; tnum < num\_threads; tnum++) {

s = pthread\_join(tinfo[tnum].thread\_id, &res);

if (s != 0)

handle\_error\_en(s, "pthread\_join");

printf("Joined with thread %d; returned value was %s\n",

tinfo[tnum].thread\_num, (char \*) res);

free(res); /\* Free memory allocated by thread \*/

}

free(tinfo);

exit(EXIT\_SUCCESS);

}

**PTHREAD\_JOIN(3) Linux Programmer's Manual PTHREAD\_JOIN(3)**

## NAME         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

pthread\_join - join with a terminated thread

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

**#include <pthread.h>**

**int pthread\_join(pthread\_t** *thread***, void \*\****retval***);**

Compile and link with *-pthread*.

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

The **pthread\_join**() function waits for the thread specified by *thread*

to terminate. If that thread has already terminated, then

**pthread\_join**() returns immediately. The thread specified by *thread*

must be joinable.

If *retval* is not NULL, then **pthread\_join**() copies the exit status of

the target thread (i.e., the value that the target thread supplied to

[pthread\_exit(3)](http://man7.org/linux/man-pages/man3/pthread_exit.3.html)) into the location pointed to by *retval*. If the

target thread was canceled, then **PTHREAD\_CANCELED** is placed in the

location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread,

the results are undefined. If the thread calling **pthread\_join**() is

canceled, then the target thread will remain joinable (i.e., it will

not be detached).

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

On success, **pthread\_join**() returns 0; on error, it returns an error

number.

## ERRORS         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

**EDEADLK**

A deadlock was detected (e.g., two threads tried to join with

each other); or *thread* specifies the calling thread.

**EINVAL** *thread* is not a joinable thread.

**EINVAL** Another thread is already waiting to join with this thread.

**ESRCH** No thread with the ID *thread* could be found.

## ATTRIBUTES         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

For an explanation of the terms used in this section, see

[attributes(7)](http://man7.org/linux/man-pages/man7/attributes.7.html).

┌───────────────┬───────────────┬─────────┐

│**Interface** │ **Attribute** │ **Value** │

├───────────────┼───────────────┼─────────┤

│**pthread\_join**() │ Thread safety │ MT-Safe │

└───────────────┴───────────────┴─────────┘

## CONFORMING TO         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

POSIX.1-2001, POSIX.1-2008.

## NOTES         [top](http://man7.org/linux/man-pages/man3/pthread_join.3.html#top_of_page)

After a successful call to **pthread\_join**(), the caller is guaranteed

that the target thread has terminated. The caller may then choose to

do any clean-up that is required after termination of the thread

(e.g., freeing memory or other resources that were allocated to the

target thread).

Joining with a thread that has previously been joined results in

undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not

detached), produces a "zombie thread". Avoid doing this, since each

zombie thread consumes some system resources, and when enough zombie

threads have accumulated, it will no longer be possible to create new

threads (or processes).

There is no pthreads analog of *waitpid(-1, &status, 0)*, that is,

"join with any terminated thread". If you believe you need this

functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with

any other thread in the process.

**LSEEK(2) Linux Programmer's Manual LSEEK(2)**

## NAME         [top](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

lseek - reposition read/write file offset

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

**#include <sys/types.h>**

**#include <unistd.h>**

**off\_t lseek(int** *fd***, off\_t** *offset***, int** *whence***);**

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

**lseek**() repositions the file offset of the open file description

associated with the file descriptor *fd* to the argument *offset*

according to the directive *whence* as follows:

**SEEK\_SET**

The file offset is set to *offset* bytes.

**SEEK\_CUR**

The file offset is set to its current location plus *offset*

bytes.

**SEEK\_END**

The file offset is set to the size of the file plus *offset*

bytes.

**lseek**() allows the file offset to be set beyond the end of the file

(but this does not change the size of the file). If data is later

written at this point, subsequent reads of the data in the gap (a

"hole") return null bytes ('\0') until data is actually written into

the gap.

**Seeking file data and holes**

Since version 3.1, Linux supports the following additional values for

*whence*:

**SEEK\_DATA**

Adjust the file offset to the next location in the file

greater than or equal to *offset* containing data. If *offset*

points to data, then the file offset is set to *offset*.

**SEEK\_HOLE**

Adjust the file offset to the next hole in the file greater

than or equal to *offset*. If *offset* points into the middle of

a hole, then the file offset is set to *offset*. If there is no

hole past *offset*, then the file offset is adjusted to the end

of the file (i.e., there is an implicit hole at the end of any

file).

In both of the above cases, **lseek**() fails if *offset* points past the

end of the file.

These operations allow applications to map holes in a sparsely

allocated file. This can be useful for applications such as file

backup tools, which can save space when creating backups and preserve

holes, if they have a mechanism for discovering holes.

For the purposes of these operations, a hole is a sequence of zeros

that (normally) has not been allocated in the underlying file

storage. However, a filesystem is not obliged to report holes, so

these operations are not a guaranteed mechanism for mapping the

storage space actually allocated to a file. (Furthermore, a sequence

of zeros that actually has been written to the underlying storage may

not be reported as a hole.) In the simplest implementation, a

filesystem can support the operations by making **SEEK\_HOLE** always

return the offset of the end of the file, and making **SEEK\_DATA** always

return *offset* (i.e., even if the location referred to by *offset* is a

hole, it can be considered to consist of data that is a sequence of

zeros).

The **\_GNU\_SOURCE** feature test macro must be defined in order to obtain

the definitions of **SEEK\_DATA** and **SEEK\_HOLE** from *<unistd.h>*.

The **SEEK\_HOLE** and **SEEK\_DATA** operations are supported for the

following filesystems:

\* Btrfs (since Linux 3.1)

\* OCFS (since Linux 3.2)

\* XFS (since Linux 3.5)

\* ext4 (since Linux 3.8)

\* [tmpfs(5)](http://man7.org/linux/man-pages/man5/tmpfs.5.html) (since Linux 3.8)

\* NFS (since Linux 3.18)

\* FUSE (since Linux 4.5)

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

Upon successful completion, **lseek**() returns the resulting offset

location as measured in bytes from the beginning of the file. On

error, the value *(off\_t) -1* is returned and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate

the error.

## ERRORS         [top](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

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

**EINVAL** *whence* is not valid. Or: the resulting file offset would be

negative, or beyond the end of a seekable device.

**ENXIO** *whence* is **SEEK\_DATA** or **SEEK\_HOLE**, and the file offset is

beyond the end of the file.

**EOVERFLOW**

The resulting file offset cannot be represented in an *off\_t*.

**ESPIPE** *fd* is associated with a pipe, socket, or FIFO.

**FORK(2) Linux Programmer's Manual FORK(2)**

## NAME         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

fork - create a child process

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

**#include <sys/types.h>**

**#include <unistd.h>**

**pid\_t fork(void);**

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

**fork**() creates a new process by duplicating the calling process. The

new process is referred to as the *child* process. The calling process

is referred to as the *parent* process.

The child process and the parent process run in separate memory

spaces. At the time of **fork**() both memory spaces have the same

content. Memory writes, file mappings ([mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html)), and unmappings

([munmap(2)](http://man7.org/linux/man-pages/man2/munmap.2.html)) performed by one of the processes do not affect the

other.

The child process is an exact duplicate of the parent process except

for the following points:

\* The child has its own unique process ID, and this PID does not

match the ID of any existing process group ([setpgid(2)](http://man7.org/linux/man-pages/man2/setpgid.2.html)) or

session.

\* The child's parent process ID is the same as the parent's process

ID.

\* The child does not inherit its parent's memory locks ([mlock(2)](http://man7.org/linux/man-pages/man2/mlock.2.html),

[mlockall(2)](http://man7.org/linux/man-pages/man2/mlockall.2.html)).

\* Process resource utilizations ([getrusage(2)](http://man7.org/linux/man-pages/man2/getrusage.2.html)) and CPU time counters

([times(2)](http://man7.org/linux/man-pages/man2/times.2.html)) are reset to zero in the child.

\* The child's set of pending signals is initially empty

([sigpending(2)](http://man7.org/linux/man-pages/man2/sigpending.2.html)).

\* The child does not inherit semaphore adjustments from its parent

([semop(2)](http://man7.org/linux/man-pages/man2/semop.2.html)).

\* The child does not inherit process-associated record locks from

its parent ([fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)). (On the other hand, it does inherit

[fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) open file description locks and [flock(2)](http://man7.org/linux/man-pages/man2/flock.2.html) locks from its

parent.)

\* The child does not inherit timers from its parent ([setitimer(2)](http://man7.org/linux/man-pages/man2/setitimer.2.html),

[alarm(2)](http://man7.org/linux/man-pages/man2/alarm.2.html), [timer\_create(2)](http://man7.org/linux/man-pages/man2/timer_create.2.html)).

\* The child does not inherit outstanding asynchronous I/O operations

from its parent ([aio\_read(3)](http://man7.org/linux/man-pages/man3/aio_read.3.html), [aio\_write(3)](http://man7.org/linux/man-pages/man3/aio_write.3.html)), nor does it inherit

any asynchronous I/O contexts from its parent (see [io\_setup(2)](http://man7.org/linux/man-pages/man2/io_setup.2.html)).

The process attributes in the preceding list are all specified in

POSIX.1. The parent and child also differ with respect to the

following Linux-specific process attributes:

\* The child does not inherit directory change notifications

(dnotify) from its parent (see the description of **F\_NOTIFY** in

[fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)).

\* The [prctl(2)](http://man7.org/linux/man-pages/man2/prctl.2.html) **PR\_SET\_PDEATHSIG** setting is reset so that the child

does not receive a signal when its parent terminates.

\* The default timer slack value is set to the parent's current timer

slack value. See the description of **PR\_SET\_TIMERSLACK** in

[prctl(2)](http://man7.org/linux/man-pages/man2/prctl.2.html).

\* Memory mappings that have been marked with the [madvise(2)](http://man7.org/linux/man-pages/man2/madvise.2.html)

**MADV\_DONTFORK** flag are not inherited across a **fork**().

\* Memory in address ranges that have been marked with the [madvise(2)](http://man7.org/linux/man-pages/man2/madvise.2.html)

**MADV\_WIPEONFORK** flag is zeroed in the child after a **fork**(). (The

**MADV\_WIPEONFORK** setting remains in place for those address ranges

in the child.)

\* The termination signal of the child is always **SIGCHLD** (see

[clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html)).

\* The port access permission bits set by [ioperm(2)](http://man7.org/linux/man-pages/man2/ioperm.2.html) are not inherited

by the child; the child must turn on any bits that it requires

using [ioperm(2)](http://man7.org/linux/man-pages/man2/ioperm.2.html).

Note the following further points:

\* The child process is created with a single thread—the one that

called **fork**(). The entire virtual address space of the parent is

replicated in the child, including the states of mutexes,

condition variables, and other pthreads objects; the use of

[pthread\_atfork(3)](http://man7.org/linux/man-pages/man3/pthread_atfork.3.html) may be helpful for dealing with problems that

this can cause.

\* After a **fork**() in a multithreaded program, the child can safely

call only async-signal-safe functions (see [signal-safety(7)](http://man7.org/linux/man-pages/man7/signal-safety.7.html)) until

such time as it calls [execve(2)](http://man7.org/linux/man-pages/man2/execve.2.html).

\* The child inherits copies of the parent's set of open file

descriptors. Each file descriptor in the child refers to the same

open file description (see [open(2)](http://man7.org/linux/man-pages/man2/open.2.html)) as the corresponding file

descriptor in the parent. This means that the two file

descriptors share open file status flags, file offset, and signal-

driven I/O attributes (see the description of **F\_SETOWN** and

**F\_SETSIG** in [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html)).

\* The child inherits copies of the parent's set of open message

queue descriptors (see [mq\_overview(7)](http://man7.org/linux/man-pages/man7/mq_overview.7.html)). Each file descriptor in

the child refers to the same open message queue description as the

corresponding file descriptor in the parent. This means that the

two file descriptors share the same flags (*mq\_flags*).

\* The child inherits copies of the parent's set of open directory

streams (see [opendir(3)](http://man7.org/linux/man-pages/man3/opendir.3.html)). POSIX.1 says that the corresponding

directory streams in the parent and child *may* share the directory

stream positioning; on Linux/glibc they do not.

## RETURN VALUE         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

On success, the PID of the child process is returned in the parent,

and 0 is returned in the child. On failure, -1 is returned in the

parent, no child process is created, and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set appropriately.

## ERRORS         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

**EAGAIN** A system-imposed limit on the number of threads was

encountered. There are a number of limits that may trigger

this error:

\* the **RLIMIT\_NPROC** soft resource limit (set via

[setrlimit(2)](http://man7.org/linux/man-pages/man2/setrlimit.2.html)), which limits the number of processes and

threads for a real user ID, was reached;

\* the kernel's system-wide limit on the number of processes

and threads, */proc/sys/kernel/threads-max*, was reached (see

[proc(5)](http://man7.org/linux/man-pages/man5/proc.5.html));

\* the maximum number of PIDs, */proc/sys/kernel/pid\_max*, was

reached (see [proc(5)](http://man7.org/linux/man-pages/man5/proc.5.html)); or

\* the PID limit (*pids.max*) imposed by the cgroup "process

number" (PIDs) controller was reached.

**EAGAIN** The caller is operating under the **SCHED\_DEADLINE** scheduling

policy and does not have the reset-on-fork flag set. See

[sched(7)](http://man7.org/linux/man-pages/man7/sched.7.html).

**ENOMEM fork**() failed to allocate the necessary kernel structures

because memory is tight.

**ENOMEM** An attempt was made to create a child process in a PID

namespace whose "init" process has terminated. See

[pid\_namespaces(7)](http://man7.org/linux/man-pages/man7/pid_namespaces.7.html).

**ENOSYS fork**() is not supported on this platform (for example,

hardware without a Memory-Management Unit).

**ERESTARTNOINTR** (since Linux 2.6.17)

System call was interrupted by a signal and will be restarted.

(This can be seen only during a trace.)

## CONFORMING TO         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

## NOTES         [top](http://man7.org/linux/man-pages/man2/fork.2.html#top_of_page)

Under Linux, **fork**() is implemented using copy-on-write pages, so the

only penalty that it incurs is the time and memory required to

duplicate the parent's page tables, and to create a unique task

structure for the child.

**C library/kernel differences**

Since version 2.3.3, rather than invoking the kernel's **fork**() system

call, the glibc **fork**() wrapper that is provided as part of the NPTL

threading implementation invokes [clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html) with flags that provide the

same effect as the traditional system call. (A call to **fork**() is

equivalent to a call to [clone(2)](http://man7.org/linux/man-pages/man2/clone.2.html) specifying *flags* as just **SIGCHLD**.)

The glibc wrapper invokes any fork handlers that have been

established using [pthread\_atfork(3)](http://man7.org/linux/man-pages/man3/pthread_atfork.3.html).

**CS111 Spring 2009**

Modularity and virtualization

April 06, 2009

*Julio Ceballos, Lorena Topete, Cody Prestwood*

**MODULARITY -** Split the program into pieces   
  
**ABSTRACTION -** use "nice" boundaries between pieces   
       L lines of code and M modules

**Problems with out paranoid word-counter program**

**A.** Performance issues

1. cache answer

2. so IO in parallel  with computation

\* will help a bit for out OS but can be better

**B.** Flexibility

-Most Important ( common needs in Software Engineering)

1. to much of a pain to change // ex: recompile  and install

2. to much of a pain to reuse parts

-Least Important

3. to much of a pain to run several programs at the same time

4. to much of a pain to recover from crash

**C.** Complexity

1. Coping techniques

\* Modularity - break programs into pieces

\* Abstraction - use "nice" boundaries  between pieces

2. Modularity-----> Find bugs Faster

-assume L lines of codes and  B bugs

- assume B is proportional to L

- assume time to find bugs is proportional to L

-debug time: is proportional to B\*L which is proportional to L

\*Now assume M modules

-now Debug time is proportional to

B/M \* L/M \* M => (L\*L)/M

\* this is assuming bugs don't cross module boundaries, and modules are the same size,

  and bugs are evenly distributed

\* Obvious which Module owns the bug

\* Modularity can greatly help you find the bugs in your code

\* API = application programing interface

**Ex: Modularity Example:**Rewriting read\_ide\_sector

**void read\_ide\_sector (int s, int a);**

  ^  ide is only good fro reading ide Disks

**void read\_sector (int s, int a);**  \*higher level of abstraction

**void read\_sector (int s, off\_t s, int a);**

  ^    ^          ^  address

  ^   ^  Sector number

  ^ Disk number

**ssize\_t read\_bytes(int s, off\_t o, char\* a, size\_t n);**

^   ^    ^          ^  address

^   ^   ^  offset from Disk start, in bytes

^   ^ Disk number

^ signed, returns # of bytes read or - number for error

**OFFICIAL UNIX**

**ssize\_t read(int d, char\* a, size\_t n);**

**off\_t lseek( int d, off\_t o, int whence);**

^ ^ ^

^ ^ ^  0 from beginning; 1 from current; 2 from end

^ ^ signed

^  returns new seek location

**in Unix:**

ssize\_t read(int d, char \*a, size\_t n)

**-always read from last seek position**

off\_t lseek(int d, off\_t o, int whence)

o - is positive or negative

off\_t is signed int

**whence: 0 from beginning**

1 current position

2 from end

**Unix design:**

1. Advantage - can share lseek with read/write functions

2. Disadvantage - Takes more time if application does

 random access

**Abstraction of function**

read\_ide\_sector(int **s**, int **a**)

**s**-sector

**a**-returned result

**change to:**read\_sector(int **d**, int **s**, int **a**)

**d**-disk ID to support multiple disks

**generalize interface: read\_sector(int d, off\_t s, char \*a)**

-provides 64-bit support for disks larger that 4Gig

s-offset into disk

a-pointer to receiving buffer

1. OS needs to maintain mapping between device and

sectors. So device number and secotr number are

separate arguments

2. Assume secotr is 512 bytes and only one sector per

read

**Positives and Negatives**

+ many apps don't care about current seek location

+ share lseek with read and write

+ encourages sequential reads

- take more time if apps generally do Random Access

**\*\*How To enforce Modularity\*\***

**1.** Do Nothing – use a single program, global variables, no functions

**2.** Function call modularity where functions are a black box

**a.** Put in BIOS and the program is called from the BIOS

**b.** Main is in RAM instead of BIOS

**c.** Stack has call – this approach has a problem

**Partition of memory**

-------------------------------------------------------------

**|** BootRecord**|**Main**|**      **|**Stack**|**     **|**BIOS **|**

-------------------------------------------------------------

**Caller and Callee Contract**

\*\* Because the  callee can potentially overwrite the caller's memory,

 a contract exists between the caller and callee preventing this.

**Example:**  recursive factorial function

int fact (int n)

{

If(n==0)

    return 1;

else

    return n\* fact (n-1);

}

**In x86 code:**

fact:

pushl  %ebp                   # push caller's frame Pointer

movl   $1, %eax               # move constant 1 into ax reg. ( this is preparing the register to return 1)

movl   %esp, %ebf             # move sp into fp

subl   $8, %esp              # allocate our frame

movl   %ebx, -4 (%ebp)       # save caller's bx

movl   8(%ebp), %ebx         # bx: =n

tstl   %ebx, %ebx            # bx=0

jne    L5                     # Jump if not zero

.L1:

movl   -4(%ebp), %ebx       # restore bx

movl   %ebp, %esp            # restore sp

popl   %ebp                  #restore bp

ret                           #restore ip

**Now let's look at the caller**

.L5:

leal   -1(%ebx), %eax        # ax =bx-1

movl   %eax, %esp            # store n-1 as argument

call   fact                         # push "\*+1" into Stack, go to fact

imull  %ebx, %eax            # ax \* =n

jmp    .L1

**Let's practice:**

**what happens if we type:   fact (6);**

pushl     $6                   # push arg into stack

call      fact

addl      $4, %esp             # discard the 6 we pushed in stack

       # 4 because a word is 4 bytes long

**We can san that we have a contract between the caller & callee.**

**Caller must:**

 top of stack =ra

 2nd of stack =n

 (ip) p.counter = fact

**Callee must:**

 put ans into ax

 POP stack

 Put result in ip

**What can go wrong with function call modularity?**

**callee:**step into caller local variables to mess the program up

i.e movl $0, 12(%esp)         # set caller's r. a. to 0

i.e movl $0, 0(ebp)           # set caller's f p to 0

i.e callee can loop

i.e subl $ 1000000000, %esp   # stack overflow

i.e callee can halt

**Here, we have SOFT MODULARITY**

 Upside :

 It is fast

Downside:

Does not prevent bugs

We often want **HARD MODULARITY** because with this, failure in one module doesn't leak into  the other.

i.e  write an x86 simulator and run callee inside it.

**HARD** Modularity

Upside:

   This is robust and safe

Downside:

   It is very slow

**What we want is a virtualizable processor that can do the following:**

-full speed on normal instructions

-slow down only when it checks anything weird

-gets control on any "bad" instruction

-executes some software to execute this bad instruction

LIST of SYSCALL

System call Kernel Notes

\_llseek(2) 1.2

\_newselect(2) 2.0

\_sysctl(2) 2.0

accept(2) 2.0 See notes on socketcall(2)

accept4(2) 2.6.28

access(2) 1.0

acct(2) 1.0

add\_key(2) 2.6.10

adjtimex(2) 1.0

alarm(2) 1.0

alloc\_hugepages(2) 2.5.36 Removed in 2.5.44

bdflush(2) 1.2 Deprecated (does nothing)

since 2.6

bind(2) 2.0 See notes on socketcall(2)

bpf(2) 3.18

brk(2) 1.0

cacheflush(2) 1.2 Not on x86

capget(2) 2.2

capset(2) 2.2

chdir(2) 1.0

chmod(2) 1.0

chown(2) 2.2 See chown(2) for

version details

chown32(2) 2.4

chroot(2) 1.0

clock\_adjtime(2) 2.6.39

clock\_getres(2) 2.6

clock\_gettime(2) 2.6

clock\_nanosleep(2) 2.6

clock\_settime(2) 2.6

clone(2) 1.0

close(2) 1.0

connect(2) 2.0 See notes on socketcall(2)

copy\_file\_range(2) 4.5

creat(2) 1.0

create\_module(2) 1.0 Removed in 2.6

delete\_module(2) 1.0

dup(2) 1.0

dup2(2) 1.0

dup3(2) 2.6.27

epoll\_create(2) 2.6

epoll\_create1(2) 2.6.27

epoll\_ctl(2) 2.6

epoll\_pwait(2) 2.6.19

epoll\_wait(2) 2.6

eventfd(2) 2.6.22

eventfd2(2) 2.6.27

execve(2) 1.0

execveat(2) 3.19

exit(2) 1.0

exit\_group(2) 2.6

faccessat(2) 2.6.16

fadvise64(2) 2.6

fadvise64\_64(2) 2.6

fallocate(2) 2.6.23

fanotify\_init(2) 2.6.37

fanotify\_mark(2) 2.6.37

fchdir(2) 1.0

fchmod(2) 1.0

fchmodat(2) 2.6.16

fchown(2) 1.0

fchown32(2) 2.4

fchownat(2) 2.6.16

fcntl(2) 1.0

fcntl64(2) 2.4

fdatasync(2) 2.0

fgetxattr(2) 2.6; 2.4.18

finit\_module(2) 3.8

flistxattr(2) 2.6; 2.4.18

flock(2) 2.0

fork(2) 1.0

free\_hugepages(2) 2.5.36 Removed in 2.5.44

fremovexattr(2) 2.6; 2.4.18

fsetxattr(2) 2.6; 2.4.18

fstat(2) 1.0

fstat64(2) 2.4

fstatat64(2) 2.6.16

fstatfs(2) 1.0

fstatfs64(2) 2.6

fsync(2) 1.0

ftruncate(2) 1.0

ftruncate64(2) 2.4

futex(2) 2.6

futimesat(2) 2.6.16

get\_kernel\_syms(2) 1.0 Removed in 2.6

get\_mempolicy(2) 2.6.6

get\_robust\_list(2) 2.6.17

get\_thread\_area(2) 2.6

getcpu(2) 2.6.19

getcwd(2) 2.2

getdents(2) 2.0

getdents64(2) 2.4

getegid(2) 1.0

getegid32(2) 2.4

geteuid(2) 1.0

geteuid32(2) 2.4

getgid(2) 1.0

getgid32(2) 2.4

getgroups(2) 1.0

getgroups32(2) 2.4

getitimer(2) 1.0

getpeername(2) 2.0 See notes on socketcall(2)

getpagesize(2) 2.0 Not on x86

getpgid(2) 1.0

getpgrp(2) 1.0

getpid(2) 1.0

getppid(2) 1.0

getpriority(2) 1.0

getrandom(2) 3.17

getresgid(2) 2.2

getresgid32(2) 2.4

getresuid(2) 2.2

getresuid32(2) 2.4

getrlimit(2) 1.0

getrusage(2) 1.0

getsid(2) 2.0

getsockname(2) 2.0 See notes on socketcall(2)

getsockopt(2) 2.0 See notes on socketcall(2)

gettid(2) 2.4.11

gettimeofday(2) 1.0

getuid(2) 1.0

getuid32(2) 2.4

getunwind(2) 2.4.8 ia64; deprecated

getxattr(2) 2.6; 2.4.18

init\_module(2) 1.0

inotify\_add\_watch(2) 2.6.13

inotify\_init(2) 2.6.13

inotify\_init1(2) 2.6.27

inotify\_rm\_watch(2) 2.6.13

io\_cancel(2) 2.6

io\_destroy(2) 2.6

io\_getevents(2) 2.6

io\_setup(2) 2.6

io\_submit(2) 2.6

ioctl(2) 1.0

ioperm(2) 1.0

iopl(2) 1.0

ioprio\_get(2) 2.6.13

ioprio\_set(2) 2.6.13

ipc(2) 1.0

kcmp(2) 3.5

kern\_features(2) 3.7 Sparc64

kexec\_file\_load(2) 3.17

kexec\_load(2) 2.6.13

keyctl(2) 2.6.10

kill(2) 1.0

lchown(2) 1.0 See chown(2) for

version details

lchown32(2) 2.4

lgetxattr(2) 2.6; 2.4.18

link(2) 1.0

linkat(2) 2.6.16

listen(2) 2.0 See notes on socketcall(2)

listxattr(2) 2.6; 2.4.18

llistxattr(2) 2.6; 2.4.18

lookup\_dcookie(2) 2.6

lremovexattr(2) 2.6; 2.4.18

lseek(2) 1.0

lsetxattr(2) 2.6; 2.4.18

lstat(2) 1.0

lstat64(2) 2.4

madvise(2) 2.4

mbind(2) 2.6.6

membarrier(2) 3.17

memfd\_create(2) 3.17

migrate\_pages(2) 2.6.16

mincore(2) 2.4

mkdir(2) 1.0

mkdirat(2) 2.6.16

mknod(2) 1.0

mknodat(2) 2.6.16

mlock(2) 2.0

mlock2(2) 4.4

mlockall(2) 2.0

mmap(2) 1.0

mmap2(2) 2.4

modify\_ldt(2) 1.0

mount(2) 1.0

move\_pages(2) 2.6.18

mprotect(2) 1.0

mq\_getsetattr(2) 2.6.6

mq\_notify(2) 2.6.6

mq\_open(2) 2.6.6

mq\_timedreceive(2) 2.6.6

mq\_timedsend(2) 2.6.6

mq\_unlink(2) 2.6.6

mremap(2) 2.0

msgctl(2) 2.0 See notes on ipc(2)

msgget(2) 2.0 See notes on ipc(2)

msgrcv(2) 2.0 See notes on ipc(2)

msgsnd(2) 2.0 See notes on ipc(2)

msync(2) 2.0

munlock(2) 2.0

munlockall(2) 2.0

munmap(2) 1.0

name\_to\_handle\_at(2) 2.6.39

nanosleep(2) 2.0

nfsservctl(2) 2.2 Removed in 3.1

nice(2) 1.0

oldfstat(2) 1.0

oldlstat(2) 1.0

oldolduname(2) 1.0

oldstat(2) 1.0

olduname(2) 1.0

open(2) 1.0

open\_by\_handle\_at(2) 2.6.39

openat(2) 2.6.16

pause(2) 1.0

pciconfig\_iobase(2) 2.2.15; 2.4 Not on x86

pciconfig\_read(2) 2.0.26; 2.2 Not on x86

pciconfig\_write(2) 2.0.26; 2.2 Not on x86

perf\_event\_open(2) 2.6.31 Was perf\_counter\_open() in

2.6.31; renamed in 2.6.32

personality(2) 1.2

perfctr(2) 2.2 Sparc; removed in 2.6.34

perfmonctl(2) 2.4 ia64

pipe(2) 1.0

pipe2(2) 2.6.27

pivot\_root(2) 2.4

pkey\_alloc(2) 4.8

pkey\_free(2) 4.8

pkey\_mprotect(2) 4.8

poll(2) 2.0.36; 2.2

ppc\_rtas(2) 2.6.2 PowerPC only

ppc\_swapcontext(2) 2.6.3 PowerPC only

ppoll(2) 2.6.16

prctl(2) 2.2

pread64(2) Added as "pread" in 2.2;

renamed "pread64" in 2.6

preadv(2) 2.6.30

preadv2(2) 4.6

prlimit64(2) 2.6.36

process\_vm\_readv(2) 3.2

process\_vm\_writev(2) 3.2

pselect6(2) 2.6.16

ptrace(2) 1.0

pwrite64(2) Added as "pwrite" in 2.2;

renamed "pwrite64" in 2.6

pwritev(2) 2.6.30

pwritev2(2) 4.6

query\_module(2) 2.2 Removed in 2.6

quotactl(2) 1.0

read(2) 1.0

readahead(2) 2.4.13

readdir(2) 1.0

readlink(2) 1.0

readlinkat(2) 2.6.16

readv(2) 2.0

reboot(2) 1.0

recv(2) 2.0 See notes on socketcall(2)

recvfrom(2) 2.0 See notes on socketcall(2)

recvmsg(2) 2.0 See notes on socketcall(2)

recvmmsg(2) 2.6.33

remap\_file\_pages(2) 2.6 Deprecated since 3.16

removexattr(2) 2.6; 2.4.18

rename(2) 1.0

renameat(2) 2.6.16

renameat2(2) 3.15

request\_key(2) 2.6.10

restart\_syscall(2) 2.6

rmdir(2) 1.0

rt\_sigaction(2) 2.2

rt\_sigpending(2) 2.2

rt\_sigprocmask(2) 2.2

rt\_sigqueueinfo(2) 2.2

rt\_sigreturn(2) 2.2

rt\_sigsuspend(2) 2.2

rt\_sigtimedwait(2) 2.2

rt\_tgsigqueueinfo(2) 2.6.31

s390\_runtime\_instr(2) 3.7 s390 only

s390\_pci\_mmio\_read(2) 3.19 s390 only

s390\_pci\_mmio\_write(2) 3.19 s390 only

s390\_sthyi(2) 4.15 s390 only

sched\_get\_priority\_max(2) 2.0

sched\_get\_priority\_min(2) 2.0

sched\_getaffinity(2) 2.6

sched\_getattr(2) 3.14

sched\_getparam(2) 2.0

sched\_getscheduler(2) 2.0

sched\_rr\_get\_interval(2) 2.0

sched\_setaffinity(2) 2.6

sched\_setattr(2) 3.14

sched\_setparam(2) 2.0

sched\_setscheduler(2) 2.0

sched\_yield(2) 2.0

seccomp(2) 3.17

select(2) 1.0

semctl(2) 2.0 See notes on ipc(2)

semget(2) 2.0 See notes on ipc(2)

semop(2) 2.0 See notes on ipc(2)

semtimedop(2) 2.6; 2.4.22

send(2) 2.0 See notes on socketcall(2)

sendfile(2) 2.2

sendfile64(2) 2.6; 2.4.19

sendmmsg(2) 3.0

sendmsg(2) 2.0 See notes on socketcall(2)

sendto(2) 2.0 See notes on socketcall(2)

set\_mempolicy(2) 2.6.6

set\_robust\_list(2) 2.6.17

set\_thread\_area(2) 2.6

set\_tid\_address(2) 2.6

setdomainname(2) 1.0

setfsgid(2) 1.2

setfsgid32(2) 2.4

setfsuid(2) 1.2

setfsuid32(2) 2.4

setgid(2) 1.0

setgid32(2) 2.4

setgroups(2) 1.0

setgroups32(2) 2.4

sethostname(2) 1.0

setitimer(2) 1.0

setns(2) 3.0

setpgid(2) 1.0

setpriority(2) 1.0

setregid(2) 1.0

setregid32(2) 2.4

setresgid(2) 2.2

setresgid32(2) 2.4

setresuid(2) 2.2

setresuid32(2) 2.4

setreuid(2) 1.0

setreuid32(2) 2.4

setrlimit(2) 1.0

setsid(2) 1.0

setsockopt(2) 2.0 See notes on socketcall(2)

settimeofday(2) 1.0

setuid(2) 1.0

setuid32(2) 2.4

setup(2) 1.0 Removed in 2.2

setxattr(2) 2.6; 2.4.18

sgetmask(2) 1.0

shmat(2) 2.0 See notes on ipc(2)

shmctl(2) 2.0 See notes on ipc(2)

shmdt(2) 2.0 See notes on ipc(2)

shmget(2) 2.0 See notes on ipc(2)

shutdown(2) 2.0 See notes on socketcall(2)

sigaction(2) 1.0

sigaltstack(2) 2.2

signal(2) 1.0

signalfd(2) 2.6.22

signalfd4(2) 2.6.27

sigpending(2) 1.0

sigprocmask(2) 1.0

sigreturn(2) 1.0

sigsuspend(2) 1.0

socket(2) 2.0 See notes on socketcall(2)

socketcall(2) 1.0

socketpair(2) 2.0 See notes on socketcall(2)

splice(2) 2.6.17

spu\_create(2) 2.6.16 PowerPC only

spu\_run(2) 2.6.16 PowerPC only

ssetmask(2) 1.0

stat(2) 1.0

stat64(2) 2.4

statfs(2) 1.0

statfs64(2) 2.6

statx(2) 4.11

stime(2) 1.0

subpage\_prot(2) 2.6.25 PowerPC only

swapoff(2) 1.0

swapon(2) 1.0

symlink(2) 1.0

symlinkat(2) 2.6.16

sync(2) 1.0

sync\_file\_range(2) 2.6.17

sync\_file\_range2(2) 2.6.22

syncfs(2) 2.6.39

sysfs(2) 1.2

sysinfo(2) 1.0

syslog(2) 1.0

tee(2) 2.6.17

tgkill(2) 2.6

time(2) 1.0

timer\_create(2) 2.6

timer\_delete(2) 2.6

timer\_getoverrun(2) 2.6

timer\_gettime(2) 2.6

timer\_settime(2) 2.6

timerfd\_create(2) 2.6.25

timerfd\_gettime(2) 2.6.25

timerfd\_settime(2) 2.6.25

times(2) 1.0

tkill(2) 2.6; 2.4.22

truncate(2) 1.0

truncate64(2) 2.4

ugetrlimit(2) 2.4

umask(2) 1.0

umount(2) 1.0

umount2(2) 2.2

uname(2) 1.0

unlink(2) 1.0

unlinkat(2) 2.6.16

unshare(2) 2.6.16

uselib(2) 1.0

ustat(2) 1.0

userfaultfd(2) 4.3

utime(2) 1.0

utimensat(2) 2.6.22

utimes(2) 2.2

utrap\_install(2) 2.2 Sparc only

vfork(2) 2.2

vhangup(2) 1.0

vm86old(2) 1.0 Was "vm86"; renamed in

2.0.28/2.2

vm86(2) 2.0.28; 2.2

vmsplice(2) 2.6.17

wait4(2) 1.0

waitid(2) 2.6.10

waitpid(2) 1.0

write(2) 1.0

writev(2) 2.0