NAME

fcntl − manipulate file descriptor

SYNOPSIS

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

DESCRIPTION

fcntl() performs one of the operations described below on the open file descriptor *fd*. The operation is determined by *cmd*.

fcntl() can take an optional third argument. Whether or not this argument is required is determined by *cmd*. The required argument type is indicated in parentheses after each *cmd* name (in most cases, the required type is *long*, and we identify the argument using the name *arg*), or *void* is specified if the argument is not required.

Duplicating a file descriptor

F_DUPFD (*long*)

Find the lowest numbered available file descriptor greater than or equal to *arg* and make it be a copy of *fd*. This is different from **dup2**(2), which uses exactly the descriptor specified.

On success, the new descriptor is returned.

See **dup**(2) for further details.

F_DUPFD_CLOEXEC (*long*; since Linux 2.6.24)

As for **F_DUPFD**, but additionally set the close-on-exec flag for the duplicate descriptor. Specifying this flag permits a program to avoid an additional **fcntl**() **F_SETFD** operation to set the **FD_CLOEXEC** flag. For an explanation of why this flag is useful, see the description of **O_CLOEXEC** in **open**(2).

File descriptor flags

The following commands manipulate the flags associated with a file descriptor. Currently, only one such flag is defined: **FD_CLOEXEC**, the close-on-exec flag. If the **FD_CLOEXEC** bit is 0, the file descriptor will remain open across an **execve**(2), otherwise it will be closed.

F_GETFD (*void*)

Read the file descriptor flags; *arg* is ignored.

F_SETFD (*long*)

Set the file descriptor flags to the value specified by *arg*.

File status flags

Each open file description has certain associated status flags, initialized by **open**(2) and possibly modified by **fcntl**(). Duplicated file descriptors (made with **dup**(2), **fcntl**(F_DUPFD), **fork**(2), etc.) refer to the same open file description, and thus share the same file status flags.

The file status flags and their semantics are described in **open**(2).

F_GETFL (*void*)

Read the file status flags; *arg* is ignored.

F_SETFL (*long*)

Set the file status flags to the value specified by *arg*. File access mode (**O_RDONLY**, **O_WRONLY**, **O_RDWR**) and file creation flags (i.e., **O_CREAT**, **O_EXCL**, **O_NOCTTY**, **O_TRUNC**) in *arg* are ignored. On Linux this command can only change the **O_APPEND**, **O_ASYNC**, **O_DIRECT**, **O_NOATIME**, and **O_NONBLOCK** flags.

Advisory locking

F_GETLK, **F_SETLK** and **F_SETLKW** are used to acquire, release, and test for the existence of record locks (also known as file-segment or file-region locks). The third argument, *lock*, is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (F_GETLK only) */
    ...
};
```

The *l_whence*, *l_start*, and *l_len* fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

l_start is the starting offset for the lock, and is interpreted relative to either: the start of the file (if *l_whence* is **SEEK_SET**); the current file offset (if *l_whence* is **SEEK_CUR**); or the end of the file (if *l_whence* is **SEEK_END**). In the final two cases, *l_start* can be a negative number provided the offset does not lie before the start of the file.

l_len specifies the number of bytes to be locked. If *l_len* is positive, then the range to be locked covers bytes *l_start* up to and including *l_start+l_len−1*. Specifying 0 for *l_len* has the special meaning: lock all bytes starting at the location specified by *l_whence* and *l_start* through to the end of file, no matter how large the file grows.

POSIX.1-2001 allows (but does not require) an implementation to support a negative *l_len* value; if *l_len* is negative, the interval described by *lock* covers bytes *l_start*+*l_len* up to and including *l_start*−1. This is supported by Linux since kernel versions 2.4.21 and 2.5.49.

The *l_type* field can be used to place a read (**F_RDLCK**) or a write (**F_WRLCK**) lock on a file. Any number of processes may hold a read lock (shared lock) on a file region, but only one process may hold a write lock (exclusive lock). An exclusive lock excludes all other locks, both shared and exclusive. A single process can hold only one type of lock on a file region; if a new lock is applied to an already-locked region, then the existing lock is converted to the new lock type. (Such conversions may involve splitting, shrinking, or coalescing with an existing lock if the byte range specified by the new lock does not precisely coincide with the range of the existing lock.)

F_SETLK (*struct flock **)

Acquire a lock (when *l_type* is **F_RDLCK** or **F_WRLCK**) or release a lock (when *l_type* is **F_UNLCK**) on the bytes specified by the *l_whence*, *l_start*, and *l_len* fields of *lock*. If a conflicting lock is held by another process, this call returns −1 and sets *errno* to **EACCES** or **EAGAIN**.

F_SETLKW (*struct flock **)

As for **F_SETLK**, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and (after the signal handler has returned) returns immediately (with return value −1 and *errno* set to **EINTR**; see **signal**(7)).

F_GETLK (*struct flock **)

On input to this call, *lock* describes a lock we would like to place on the file. If the lock could be placed, **fcntl**() does not actually place it, but returns **F_UNLCK** in the *l_type* field of *lock* and leaves the other fields of the structure unchanged. If one or more incompatible locks would prevent this lock being placed, then **fcntl**() returns details about one of these locks in the *l_type*, *l_whence*, *l_start*, and *l_len* fields of *lock* and sets *l_pid* to be the PID of the process holding that lock.

In order to place a read lock, *fd* must be open for reading. In order to place a write lock, *fd* must be open for writing. To place both types of lock, open a file read-write.

As well as being removed by an explicit **F_UNLCK**, record locks are automatically released when the process terminates or if it closes *any* file descriptor referring to a file on which locks are held. This is bad: it means that a process can lose the locks on a file like */etc/passwd* or */etc/mtab* when for some reason a library function decides to open, read and close it.

Record locks are not inherited by a child created via **fork**(2), but are preserved across an **execve**(2).

Because of the buffering performed by the **stdio**(3) library, the use of record locking with routines in that package should be avoided; use **read**(2) and **write**(2) instead.

Mandatory locking

(Non-POSIX.) The above record locks may be either advisory or mandatory, and are advisory by default.

Advisory locks are not enforced and are useful only between cooperating processes.

Mandatory locks are enforced for all processes. If a process tries to perform an incompatible access (e.g., **read**(2) or **write**(2)) on a file region that has an incompatible mandatory lock, then the result depends upon whether the **O_NONBLOCK** flag is enabled for its open file description. If the **O_NONBLOCK** flag is not enabled, then system call is blocked until the lock is removed or converted to a mode that is compatible with the access. If the **O_NONBLOCK** flag is enabled, then the system call fails with the error **EAGAIN .**

To make use of mandatory locks, mandatory locking must be enabled both on the file system that contains the file to be locked, and on the file itself. Mandatory locking is enabled on a file system using the "−o mand" option to **mount**(8), or the **MS_MANDLOCK** flag for **mount**(2). Mandatory locking is enabled on a file by disabling group execute permission on the file and enabling the set-group-ID permission bit (see **chmod**(1) and **chmod**(2)).

The Linux implementation of mandatory locking is unreliable. See BUGS below.

Managing signals

F_GETOWN, **F_SETOWN**, **F_GETSIG** and **F_SETSIG** are used to manage I/O availability signals:

F_GETOWN (*void*)

Return (as the function result) the process ID or process group currently receiving **SIGIO** and **SIGURG** signals for events on file descriptor *fd*. Process IDs are returned as positive values; process group IDs are returned as negative values (but see BUGS below). *arg* is ignored.

F_SETOWN (*long*)

Set the process ID or process group ID that will receive **SIGIO** and **SIGURG** signals for events on file descriptor *fd* to the ID given in *arg*. A process ID is specified as a positive value; a process group ID is specified as a negative value. Most commonly, the calling process specifies itself as the owner (that is, *arg* is specified as **getpid**(2)).

If you set the **O_ASYNC** status flag on a file descriptor by using the **F_SETFL** command of **fcntl**(), a **SIGIO** signal is sent whenever input or output becomes possible on that file descriptor. **F_SETSIG** can be used to obtain delivery of a signal other than **SIGIO**. If this permission check fails, then the signal is silently discarded.

Sending a signal to the owner process (group) specified by **F_SETOWN** is subject to the same permissions checks as are described for **kill**(2), where the sending process is the one that employs **F_SETOWN** (but see BUGS below).

If the file descriptor *fd* refers to a socket, **F_SETOWN** also selects the recipient of **SIGURG** signals that are delivered when out-of-band data arrives on that socket. (**SIGURG** is sent in any situation where **select**(2) would report the socket as having an "exceptional condition".)

If a non-zero value is given to **F_SETSIG** in a multithreaded process running with a threading library that supports thread groups (e.g., NPTL), then a positive value given to **F_SETOWN** has a different meaning: instead of being a process ID identifying a whole process, it is a thread ID identifying a specific thread within a process. Consequently, it may be necessary to pass **F_SETOWN** the result of **gettid**(2) instead of **getpid**(2) to get sensible results when **F_SETSIG** is used. (In current Linux threading implementations, a main thread's thread ID is the same as its process ID. This means that a single-threaded program can equally use **gettid**(2) or **getpid**(2) in this scenario.) Note, however, that the statements in this paragraph do not apply to the **SIGURG** signal generated for out-of-band data on a socket: this signal is always sent to either a process or a process group, depending on the value given to **F_SETOWN**. Note also that Linux imposes a limit on the number of real-time signals that may be queued to a process (see **getrlimit**(2) and **signal**(7)) and if this limit is reached, then the kernel reverts to delivering **SIGIO**, and this signal is delivered to the entire process rather than to a specific thread.

F_GETSIG (*void*)

Return (as the function result) the signal sent when input or output becomes possible. A value of zero means **SIGIO** is sent. Any other value (including **SIGIO**) is the signal sent instead, and in this case additional info is available to the signal handler if installed with **SA_SIGINFO**. *arg* is ignored.

F_SETSIG (*long*)

Set the signal sent when input or output becomes possible to the value given in *arg*. A value of zero means to send the default **SIGIO** signal. Any other value (including **SIGIO**) is the signal to send instead, and in this case additional info is available to the signal handler if installed with **SA_SIGINFO**.

Additionally, passing a non-zero value to **F_SETSIG** changes the signal recipient from a whole process to a specific thread within a process. See the description of **F_SETOWN** for more details.

By using **F_SETSIG** with a non-zero value, and setting **SA_SIGINFO** for the signal handler (see **sigaction**(2)), extra information about I/O events is passed to the handler in a *siginfo_t* structure. If the *si_code* field indicates the source is **SI_SIGIO**, the *si_fd* field gives the file descriptor associated with the event. Otherwise, there is no indication which file descriptors are pending, and you should use the usual mechanisms (**select**(2), **poll**(2), **read**(2) with **O_NONBLOCK** set etc.) to determine which file descriptors are available for I/O.

By selecting a real time signal (value >= **SIGRTMIN**), multiple I/O events may be queued using the same signal numbers. (Queuing is dependent on available memory). Extra information is available if **SA_SIGINFO** is set for the signal handler, as above.

Using these mechanisms, a program can implement fully asynchronous I/O without using **select**(2) or **poll**(2) most of the time.

The use of **O_ASYNC**, **F_GETOWN**, **F_SETOWN** is specific to BSD and Linux. **F_GETSIG** and **F_SETSIG** are Linux-specific. POSIX has asynchronous I/O and the *aio_sigevent* structure to achieve similar things; these are also available in Linux as part of the GNU C Library (Glibc).

Leases

F_SETLEASE and **F_GETLEASE** (Linux 2.4 onwards) are used (respectively) to establish a new lease, and retrieve the current lease, on the open file description referred to by the file descriptor *fd*. A file lease provides a mechanism whereby the process holding the lease (the "lease holder") is notified (via delivery of a signal) when a process (the "lease breaker") tries to **open**(2) or **truncate**(2) the file referred to by that file descriptor.

F_SETLEASE (*long*)

Set or remove a file lease according to which of the following values is specified in the integer *arg*:

F_RDLCK

Take out a read lease. This will cause the calling process to be notified when the file is opened for writing or is truncated. A read lease can only be placed on a file descriptor that is opened read-only.

F_WRLCK

Take out a write lease. This will cause the caller to be notified when the file is opened for reading or writing or is truncated. A write lease may be placed on a file only if there are no other open file descriptors for the file.

F_UNLCK

Remove our lease from the file.

Leases are associated with an open file description (see **open**(2)). This means that duplicate file descriptors (created by, for example, **fork**(2) or **dup**(2)) refer to the same lease, and this lease may be modified or released using any of these descriptors. Furthermore, the lease is released by either an explicit **F_UNLCK** operation on any of these duplicate descriptors, or when all such descriptors have been closed.

Leases may only be taken out on regular files. An unprivileged process may only take out a lease on a file whose UID (owner) matches the file system UID of the process. A process with the **CAP_LEASE** capability may take out leases on arbitrary files.

F_GETLEASE (*void*)

Indicates what type of lease is associated with the file descriptor *fd* by returning either **F_RDLCK**, **F_WRLCK**, or **F_UNLCK**, indicating, respectively, a read lease , a write lease, or no lease. *arg* is ignored.

When a process (the "lease breaker") performs an **open**(2) or **truncate**(2) that conflicts with a lease established via **F_SETLEASE**, the system call is blocked by the kernel and the kernel notifies the lease holder by sending it a signal (**SIGIO** by default). The lease holder should respond to receipt of this signal by doing whatever cleanup is required in preparation for the file to be accessed by another process (e.g., flushing cached buffers) and then either remove or downgrade its lease. A lease is removed by performing an **F_SETLEASE** command specifying *arg* as **F_UNLCK**. If the lease holder currently holds a write lease on the file, and the lease breaker is opening the file for reading, then it is sufficient for the lease holder to downgrade the lease to a read lease. This is done by performing an **F_SETLEASE** command specifying *arg* as **F_RDLCK**.

If the lease holder fails to downgrade or remove the lease within the number of seconds specified in */proc/sys/fs/lease-break-time* then the kernel forcibly removes or downgrades the lease holder's lease.

Once the lease has been voluntarily or forcibly removed or downgraded, and assuming the lease breaker has not unblocked its system call, the kernel permits the lease breaker's system call to proceed.

If the lease breaker's blocked **open**(2) or **truncate**(2) is interrupted by a signal handler, then the system call fails with the error **EINTR**, but the other steps still occur as described above. If the lease breaker is killed by a signal while blocked in **open**(2) or **truncate**(2), then the other steps still occur as described above. If the lease breaker specifies the **O_NONBLOCK** flag when calling **open**(2), then the call immediately fails with the error **EWOULDBLOCK**, but the other steps still occur as described above.

The default signal used to notify the lease holder is **SIGIO**, but this can be changed using the **F_SETSIG** command to **fcntl**(). If a **F_SETSIG** command is performed (even one specifying **SIGIO**), and the signal handler is established using **SA_SIGINFO**, then the handler will receive a *siginfo_t* structure as its second argument, and the *si_fd* field of this argument will hold the descriptor of the leased file that has been accessed by another process. (This is useful if the caller holds leases against multiple files).

File and directory change notification (dnotify)

F_NOTIFY (*long*)

(Linux 2.4 onwards) Provide notification when the directory referred to by *fd* or any of the files that it contains is changed. The events to be notified are specified in *arg*, which is a bit mask specified by ORing together zero or more of the following bits:

DN_ACCESS
A file was accessed (read, pread, readv)

DN_MODIFY
A file was modified (write, pwrite, writev, truncate, ftruncate).

DN_CREATE
A file was created (open, creat, mknod, mkdir, link, symlink, rename).

DN_DELETE
A file was unlinked (unlink, rename to another directory, rmdir).

DN_RENAME
A file was renamed within this directory (rename).

DN_ATTRIB
The attributes of a file were changed (chown, chmod, utime[s]).

(In order to obtain these definitions, the **_GNU_SOURCE** feature test macro must be defined.)

Directory notifications are normally "one-shot", and the application must re-register to receive further notifications. Alternatively, if **DN_MULTISHOT** is included in *arg*, then notification will remain in effect until explicitly removed.

A series of **F_NOTIFY** requests is cumulative, with the events in *arg* being added to the set already monitored. To disable notification of all events, make an **F_NOTIFY** call specifying *arg* as 0.

Notification occurs via delivery of a signal. The default signal is **SIGIO**, but this can be changed using the **F_SETSIG** command to **fcntl**(). In the latter case, the signal handler receives a *siginfo_t* structure as its second argument (if the handler was established using **SA_SIGINFO**) and the *si_fd* field of this structure contains the file descriptor which generated the notification (useful when establishing notification on multiple directories).

Especially when using **DN_MULTISHOT**, a real time signal should be used for notification, so that multiple notifications can be queued.

NOTE: New applications should use the *inotify* interface (available since kernel 2.6.13), which provides a much superior interface for obtaining notifications of file system events. See **inotify**(7).

RETURN VALUE

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

F_DUPFD The new descriptor.

F_GETFD Value of flags.

F_GETFL Value of flags.

F_GETLEASE
Type of lease held on file descriptor.

F_GETOWN Value of descriptor owner.

F_GETSIG Value of signal sent when read or write becomes possible, or zero for traditional **SIGIO** behavior.

All other commands
Zero.

On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EACCES or **EAGAIN**
Operation is prohibited by locks held by other processes.

EAGAIN
The operation is prohibited because the file has been memory-mapped by another process.

EBADF
fd is not an open file descriptor, or the command was **F_SETLK** or **F_SETLKW** and the file descriptor open mode doesn't match with the type of lock requested.

EDEADLK
It was detected that the specified **F_SETLKW** command would cause a deadlock.

EFAULT
lock is outside your accessible address space.

EINTR
For **F_SETLKW**, the command was interrupted by a signal; see **signal**(7). For **F_GETLK** and **F_SETLK**, the command was interrupted by a signal before the lock was checked or acquired. Most likely when locking a remote file (e.g., locking over NFS), but can sometimes happen locally.

EINVAL
For **F_DUPFD**, *arg* is negative or is greater than the maximum allowable value. For **F_SETSIG**, *arg* is not an allowable signal number.

EMFILE
For **F_DUPFD**, the process already has the maximum number of file descriptors open.

ENOLCK
Too many segment locks open, lock table is full, or a remote locking protocol failed (e.g., locking over NFS).

EPERM
Attempted to clear the **O_APPEND** flag on a file that has the append-only attribute set.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001. Only the operations **F_DUPFD**, **F_GETFD**, **F_SETFD**, **F_GETFL**, **F_SETFL**, **F_GETLK**, **F_SETLK**, **F_SETLKW**, **F_GETOWN**, and **F_SETOWN** are specified in POSIX.1-2001.

F_DUPFD_CLOEXEC is specified in POSIX.1-2008.

F_GETSIG, **F_SETSIG**, **F_NOTIFY**, **F_GETLEASE**, and **F_SETLEASE** are Linux-specific. (Define the **_GNU_SOURCE** macro to obtain these definitions.)

NOTES

The errors returned by **dup2**(2) are different from those returned by **F_DUPFD**.

Since kernel 2.0, there is no interaction between the types of lock placed by **flock**(2) and **fcntl**().

Several systems have more fields in *struct flock* such as, for example, *l_sysid*. Clearly, *l_pid* alone is not going to be very useful if the process holding the lock may live on a different machine.

BUGS

A limitation of the Linux system call conventions on some architectures (notably i386) means that if a (negative) process group ID to be returned by **F_GETOWN** falls in the range −1 to −4095, then the return value is wrongly interpreted by glibc as an error in the system call; that is, the return value of **fcntl**() will be −1, and *errno* will contain the (positive) process group ID.

In Linux 2.4 and earlier, there is bug that can occur when an unprivileged process uses **F_SETOWN** to specify the owner of a socket file descriptor as a process (group) other than the caller. In this case, **fcntl**() can return −1 with *errno* set to **EPERM**, even when the owner process (group) is one that the caller has permission to send signals to. Despite this error return, the file descriptor owner is set, and signals will be sent to the owner.

The implementation of mandatory locking in all known versions of Linux is subject to race conditions which render it unreliable: a **write**(2) call that overlaps with a lock may modify data after the mandatory lock is acquired; a **read**(2) call that overlaps with a lock may detect changes to data that were made only after a write lock was acquired. Similar races exist between mandatory locks and **mmap**(2). It is therefore inadvisable to rely on mandatory locking.

SEE ALSO

dup2(2), **flock**(2), **open**(2), **socket**(2), **lockf**(3), **capabilities**(7), **feature_test_macros**(7)

See also *locks.txt*, *mandatory-locking.txt*, and *dnotify.txt* in the kernel source directory *Documentation/filesystems/*. (On older kernels, these files are directly under the *Documentation/* directory, and *mandatory-locking.txt* is called *mandatory.txt*.)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.