Rocky Enterprise Linux 9.2 Manual Pages on command 'fcntl64.2'

$ man fcntl64.2

FCNTL(2)   Linux Programmer's Manual   FCNTL(2)

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 int, and we identify the argument using the name

   arg), or void is specified if the argument is not required.

   Certain of the operations below are supported only since a particular Linux kernel ver?

   sion. The preferred method of checking whether the host kernel supports a particular op?

   eration is to invoke fcntl() with the desired cmd value and then test whether the call

   failed with EINVAL, indicating that the kernel does not recognize this value.

 Duplicating a file descriptor

   F_DUPFD (int)

      Duplicate the file descriptor fd using the lowest-numbered available file descrip?

      tor greater than or equal to arg. This is different from dup2(2), which uses ex?

      actly the file descriptor specified.

On success, the new file descriptor is returned.

See dup(2) for further details.

F_DUPFD_CLOEXEC (int; since Linux 2.6.24)

As for F_DUPFD, but additionally set the close-on-exec flag for the duplicate file
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 set, the file descriptor will automatically be closed during a successful execve(2).
(If the execve(2) fails, the file descriptor is left open.) If the FD_CLOEXEC bit is not
set, the file descriptor will remain open across an execve(2).

F_GETFD (void)

Return (as the function result) the file descriptor flags; arg is ignored.

F_SETFD (int)

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

In multithreaded programs, using fcntl() F_SETFD to set the close-on-exec flag at the same
time as another thread performs a fork(2) plus execve(2) is vulnerable to a race condition
that may unintentionally leak the file descriptor to the program executed in the child
process. See the discussion of the O_CLOEXEC flag in open(2) for details and a remedy to
the problem.

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), fc?
ntl(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)

Return (as the function result) the file access mode and the file status flags; arg
is ignored.

F_SETFL (int)

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 change only the O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME, and O_NONBLOCK flags. It is not possible to change the O_DSYNC and O_SYNC flags; see BUGS, below.

Advisory record locking

Linux implements traditional ("process-associated") UNIX record locks, as standardized by POSIX. For a Linux-specific alternative with better semantics, see the discussion of open file description locks below.

F_SETLK, F_SETLKW, and F_GETLK are used to acquire, release, and test for the existence of record locks (also known as byte-range, file-segment, or file-region locks). The third argument, lock, is a pointer to a structure that has at least the following fields (in un‐ specified 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
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

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, shrink‐ing, 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. (The error returned in this case differs across implementations, so POSIX requires a portable application to check for both er‐ rors.)

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 inter‐ rupted 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 fc‐

ntl() returns details about one of those locks in the l_type, l_whence, l_start, and l_len fields of lock. If the conflicting lock is a traditional (process-asso? ciated) record lock, then the l_pid field is set to the PID of the process holding that lock. If the conflicting lock is an open file description lock, then l_pid is set to -1. Note that the returned information may already be out of date by the time the caller inspects it.

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.

When placing locks with F_SETLKW, the kernel detects deadlocks, whereby two or more pro? cesses have their lock requests mutually blocked by locks held by the other processes. For example, suppose process A holds a write lock on byte 100 of a file, and process B holds a write lock on byte 200. If each process then attempts to lock the byte already locked by the other process using F_SETLKW, then, without deadlock detection, both pro? cesses would remain blocked indefinitely. When the kernel detects such deadlocks, it causes one of the blocking lock requests to immediately fail with the error EDEADLK; an application that encounters such an error should release some of its locks to allow other applications to proceed before attempting regain the locks that it requires. Circular deadlocks involving more than two processes are also detected. Note, however, that there are limitations to the kernel's deadlock-detection algorithm; see BUGS.

As well as being removed by an explicit F_UNLCK, record locks are automatically released when the process terminates.

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.

The record locks described above are associated with the process (unlike the open file de? scription locks described below). This has some unfortunate consequences:

* If a process closes any file descriptor referring to a file, then all of the process's locks on that file are released, regardless of the file descriptor(s) on which the locks were obtained. This is bad: it means that a process can lose its locks on a file such as /etc/passwd or /etc/mtab when for some reason a library function decides to open, read, and close the same file.

* The threads in a process share locks. In other words, a multithreaded program can't

use record locking to ensure that threads don't simultaneously access the same region of a file.

Open file description locks solve both of these problems.

Open file description locks (non-POSIX)

Open file description locks are advisory byte-range locks whose operation is in most re‐spects identical to the traditional record locks described above. This lock type is Linux-specific, and available since Linux 3.15. (There is a proposal with the Austin Group to include this lock type in the next revision of POSIX.1.) For an explanation of open file descriptions, see open(2).

The principal difference between the two lock types is that whereas traditional record locks are associated with a process, open file description locks are associated with the open file description on which they are acquired, much like locks acquired with flock(2). Consequently (and unlike traditional advisory record locks), open file description locks are inherited across fork(2) (and clone(2) with CLONE_FILES), and are only automatically released on the last close of the open file description, instead of being released on any close of the file.

Conflicting lock combinations (i.e., a read lock and a write lock or two write locks) where one lock is an open file description lock and the other is a traditional record lock conflict even when they are acquired by the same process on the same file descriptor.

Open file description locks placed via the same open file description (i.e., via the same file descriptor, or via a duplicate of the file descriptor created by fork(2), dup(2), fc‐ntl() F_DUPFD, and so on) are always compatible: if a new lock is placed on an already locked region, then the existing lock is converted to the new lock type. (Such conver‐sions may result in splitting, shrinking, or coalescing with an existing lock as discussed above.)

On the other hand, open file description locks may conflict with each other when they are acquired via different open file descriptions. Thus, the threads in a multithreaded pro‐gram can use open file description locks to synchronize access to a file region by having each thread perform its own open(2) on the file and applying locks via the resulting file descriptor.

As with traditional advisory locks, the third argument to fcntl(), lock, is a pointer to an flock structure. By contrast with traditional record locks, the l_pid field of that structure must be set to zero when using the commands described below.

The commands for working with open file description locks are analogous to those used with traditional locks:

F_OFD_SETLK (struct flock *)

Acquire an open file description lock (when l_type is F_RDLCK or F_WRLCK) or re?
lease an open file description 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 EAGAIN.

F_OFD_SETLKW (struct flock *)

As for F_OFD_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_OFD_GETLK (struct flock *)

On input to this call, lock describes an open file description 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 details about one of these locks are returned via lock, as de?
scribed above for F_GETLK.

In the current implementation, no deadlock detection is performed for open file descrip?
tion locks. (This contrasts with process-associated record locks, for which the kernel
does perform deadlock detection.)

Mandatory locking

Warning: the Linux implementation of mandatory locking is unreliable. See BUGS below.
Because of these bugs, and the fact that the feature is believed to be little used, since
Linux 4.5, mandatory locking has been made an optional feature, governed by a configura?
tion option (CONFIG_MANDATORY_FILE_LOCKING). This is an initial step toward removing this
feature completely.

By default, both traditional (process-associated) and open file description record locks
are advisory. Advisory locks are not enforced and are useful only between cooperating
processes.

Both lock types can also be mandatory. 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 the 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 filesystem
that contains the file to be locked, and on the file itself. Mandatory locking is enabled
on a filesystem 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)).

Mandatory locking is not specified by POSIX. Some other systems also support mandatory
locking, although the details of how to enable it vary across systems.

Lost locks

When an advisory lock is obtained on a networked filesystem such as NFS it is possible
that the lock might get lost. This may happen due to administrative action on the server,
or due to a network partition (i.e., loss of network connectivity with the server) which
lasts long enough for the server to assume that the client is no longer functioning.

When the filesystem determines that a lock has been lost, future read(2) or write(2) re?
quests may fail with the error EIO. This error will persist until the lock is removed or
the file descriptor is closed. Since Linux 3.12, this happens at least for NFSv4 (includ?
ing all minor versions).

Some versions of UNIX send a signal (SIGLOST) in this circumstance. Linux does not define
this signal, and does not provide any asynchronous notification of lost locks.

Managing signals

F_GETOWN, F_SETOWN, F_GETOWN_EX, F_SETOWN_EX, 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 ID currently re?
ceiving 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 (int)

Set the process ID or process group ID that will receive SIGIO and SIGURG signals

for events on the file descriptor fd. The target process or process group ID is specified 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)).

As well as setting the file descriptor owner, one must also enable generation of signals on the file descriptor. This is done by using the fcntl() F_SETFL command to set the O_ASYNC file status flag on the file descriptor. Subsequently, a SIGIO signal is sent whenever input or output becomes possible on the file descriptor. The fcntl() F_SETSIG command can be used to obtain delivery of a signal other than SIGIO.

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 this permission check fails, then the signal is silently discarded. Note: The F_SETOWN operation records the caller's credentials at the time of the fcntl() call, and it is these saved credentials that are used for the permission checks.

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".)

The following was true in 2.6.x kernels up to and including kernel 2.6.11:

> If a nonzero 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 spe?
> cific 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 re?
> sults 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 sce?
> nario.) 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.

The above behavior was accidentally dropped in Linux 2.6.12, and won't be restored.

From Linux 2.6.32 onward, use F_SETOWN_EX to target SIGIO and SIGURG signals at a particular thread.

F_GETOWN_EX (struct f_owner_ex *) (since Linux 2.6.32)

Return the current file descriptor owner settings as defined by a previous F_SE? TOWN_EX operation. The information is returned in the structure pointed to by arg, which has the following form:

```
struct f_owner_ex {

  int   type;

  pid_t pid;

};
```

The type field will have one of the values F_OWNER_TID, F_OWNER_PID, or F_OWNER_PGRP. The pid field is a positive integer representing a thread ID, process ID, or process group ID. See F_SETOWN_EX for more details.

F_SETOWN_EX (struct f_owner_ex *) (since Linux 2.6.32)

This operation performs a similar task to F_SETOWN. It allows the caller to direct I/O availability signals to a specific thread, process, or process group.   The caller specifies the target of signals via arg, which is a pointer to a f_owner_ex structure. The type field has one of the following values, which define how pid is interpreted:

F_OWNER_TID

Send the signal to the thread whose thread ID (the value returned by a call to clone(2) or gettid(2)) is specified in pid.

F_OWNER_PID

Send the signal to the process whose ID is specified in pid.

F_OWNER_PGRP

Send the signal to the process group whose ID is specified in pid.   (Note that, unlike with F_SETOWN, a process group ID is specified as a positive value here.)

F_GETSIG (void)

Return (as the function result) the signal sent when input or output becomes possi? ble. 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 sig?

nal handler if installed with SA_SIGINFO. arg is ignored.

F_SETSIG (int)

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 (in?

cluding SIGIO) is the signal to send instead, and in this case additional info is

available to the signal handler if installed with SA_SIGINFO.

By using F_SETSIG with a nonzero value, and setting SA_SIGINFO for the signal han?

dler (see sigaction(2)), extra information about I/O events is passed to the han?

dler in a siginfo_t structure. If the si_code field indicates the source is SI_SI?

GIO, the si_fd field gives the file descriptor associated with the event. Other?

wise, 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 de?

termine which file descriptors are available for I/O.

Note that the file descriptor provided in si_fd is the one that was specified dur?

ing the F_SETSIG operation. This can lead to an unusual corner case. If the file

descriptor is duplicated (dup(2) or similar), and the original file descriptor is

closed, then I/O events will continue to be generated, but the si_fd field will

contain the number of the now closed file descriptor.

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.

Note 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.

Using these mechanisms, a program can implement fully asynchronous I/O without using se?

lect(2) or poll(2) most of the time.

The use of O_ASYNC is specific to BSD and Linux. The only use of F_GETOWN and F_SETOWN

specified in POSIX.1 is in conjunction with the use of the SIGURG signal on sockets.

(POSIX does not specify the SIGIO signal.) F_GETOWN_EX, F_SETOWN_EX, F_GETSIG, and F_SET?

SIG 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 onward) are used to establish a new lease, and re? trieve 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 (int)

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 be placed only 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 dupli? cate 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. Fur? thermore, the lease is released by either an explicit F_UNLCK operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

Leases may be taken out only on regular files. An unprivileged process may take out a lease only on a file whose UID (owner) matches the filesystem 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 suf? ficient for the lease holder to downgrade the lease to a read lease. This is done by per? forming 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 a lease break has been initiated, F_GETLEASE returns the target lease type (either F_RDLCK or F_UNLCK, depending on what would be compatible with the lease breaker) until the lease holder voluntarily downgrades or removes the lease or the kernel forcibly does so after the lease break timer expires.

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 de? scribed 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 re? ceive a siginfo_t structure as its second argument, and the si_fd field of this argument will hold the file 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 (int)

(Linux 2.4 onward) 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 follow?ing bits:

DN_ACCESS

A file was accessed (read(2), pread(2), readv(2), and similar)

DN_MODIFY

A file was modified (write(2), pwrite(2), writev(2), truncate(2), ftrun?cate(2), and similar).

DN_CREATE

A file was created (open(2), creat(2), mknod(2), mkdir(2), link(2), sym?link(2), rename(2) into this directory).

DN_DELETE

A file was unlinked (unlink(2), rename(2) to another directory, rmdir(2)).

DN_RENAME

A file was renamed within this directory (rename(2)).

DN_ATTRIB

The attributes of a file were changed (chown(2), chmod(2), utime(2), utimen?sat(2), and similar).

(In order to obtain these definitions, the _GNU_SOURCE feature test macro must be defined before including any header files.)

Directory notifications are normally "one-shot", and the application must reregis?ter 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(). (Note that SIGIO is one of the nonqueuing standard signals; switching to the use of a real-time signal means that multiple notifications can be queued to the process.) 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 estab‐ lishing notification on multiple directories).

Especially when using DN_MULTISHOT, a real time signal should be used for notifica‐ tion, 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 filesystem events. See inotify(7).

Changing the capacity of a pipe

F_SETPIPE_SZ (int; since Linux 2.6.35)

Change the capacity of the pipe referred to by fd to be at least arg bytes. An un‐ privileged process can adjust the pipe capacity to any value between the system page size and the limit defined in /proc/sys/fs/pipe-max-size (see proc(5)). At‐ tempts to set the pipe capacity below the page size are silently rounded up to the page size. Attempts by an unprivileged process to set the pipe capacity above the limit in /proc/sys/fs/pipe-max-size yield the error EPERM; a privileged process (CAP_SYS_RESOURCE) can override the limit.

When allocating the buffer for the pipe, the kernel may use a capacity larger than arg, if that is convenient for the implementation. (In the current implementation, the allocation is the next higher power-of-two page-size multiple of the requested size.) The actual capacity (in bytes) that is set is returned as the function re‐ sult.

Attempting to set the pipe capacity smaller than the amount of buffer space cur‐ rently used to store data produces the error EBUSY.

Note that because of the way the pages of the pipe buffer are employed when data is written to the pipe, the number of bytes that can be written may be less than the nominal size, depending on the size of the writes.

F_GETPIPE_SZ (void; since Linux 2.6.35)

Return (as the function result) the capacity of the pipe referred to by fd.

File Sealing

File seals limit the set of allowed operations on a given file. For each seal that is set on a file, a specific set of operations will fail with EPERM on this file from now on.

The file is said to be sealed. The default set of seals depends on the type of the under?

lying file and filesystem. For an overview of file sealing, a discussion of its purpose,

and some code examples, see memfd_create(2).

Currently, file seals can be applied only to a file descriptor returned by memfd_create(2)

(if the MFD_ALLOW_SEALING was employed). On other filesystems, all fcntl() operations

that operate on seals will return EINVAL.

Seals are a property of an inode. Thus, all open file descriptors referring to the same

inode share the same set of seals. Furthermore, seals can never be removed, only added.

F_ADD_SEALS (int; since Linux 3.17)

> Add the seals given in the bit-mask argument arg to the set of seals of the inode
>
> referred to by the file descriptor fd. Seals cannot be removed again.   Once this
>
> call succeeds, the seals are enforced by the kernel immediately. If the current
>
> set of seals includes F_SEAL_SEAL (see below), then this call will be rejected with
>
> EPERM.   Adding a seal that is already set is a no-op, in case F_SEAL_SEAL is not
>
> set already. In order to place a seal, the file descriptor fd must be writable.

F_GET_SEALS (void; since Linux 3.17)

> Return (as the function result) the current set of seals of the inode referred to
>
> by fd. If no seals are set, 0 is returned. If the file does not support sealing,
>
> -1 is returned and errno is set to EINVAL.

The following seals are available:

F_SEAL_SEAL

> If this seal is set, any further call to fcntl() with F_ADD_SEALS fails with the
>
> error EPERM.   Therefore, this seal prevents any modifications to the set of seals
>
> itself. If the initial set of seals of a file includes F_SEAL_SEAL, then this ef?
>
> fectively causes the set of seals to be constant and locked.

F_SEAL_SHRINK

> If this seal is set, the file in question cannot be reduced in size. This affects
>
> open(2) with the O_TRUNC flag as well as truncate(2) and ftruncate(2). Those calls
>
> fail with EPERM if you try to shrink the file in question. Increasing the file
>
> size is still possible.

F_SEAL_GROW

> If this seal is set, the size of the file in question cannot be increased.   This
>
> affects write(2) beyond the end of the file, truncate(2), ftruncate(2), and fallo?

cate(2). These calls fail with EPERM if you use them to increase the file size.

If you keep the size or shrink it, those calls still work as expected.

F_SEAL_WRITE

If this seal is set, you cannot modify the contents of the file. Note that shrink‐

ing or growing the size of the file is still possible and allowed. Thus, this seal

is normally used in combination with one of the other seals. This seal affects

write(2) and fallocate(2) (only in combination with the FALLOC_FL_PUNCH_HOLE flag).

Those calls fail with EPERM if this seal is set. Furthermore, trying to create new

shared, writable memory-mappings via mmap(2) will also fail with EPERM.

Using the F_ADD_SEALS operation to set the F_SEAL_WRITE seal fails with EBUSY if

any writable, shared mapping exists. Such mappings must be unmapped before you can

add this seal. Furthermore, if there are any asynchronous I/O operations (io_sub‐

mit(2)) pending on the file, all outstanding writes will be discarded.

F_SEAL_FUTURE_WRITE (since Linux 5.1)

The effect of this seal is similar to F_SEAL_WRITE, but the contents of the file

can still be modified via shared writable mappings that were created prior to the

seal being set. Any attempt to create a new writable mapping on the file via

mmap(2) will fail with EPERM. Likewise, an attempt to write to the file via

write(2) will fail with EPERM.

Using this seal, one process can create a memory buffer that it can continue to

modify while sharing that buffer on a "read-only" basis with other processes.

File read/write hints

Write lifetime hints can be used to inform the kernel about the relative expected lifetime

of writes on a given inode or via a particular open file description. (See open(2) for an

explanation of open file descriptions.) In this context, the term "write lifetime" means

the expected time the data will live on media, before being overwritten or erased.

An application may use the different hint values specified below to separate writes into

different write classes, so that multiple users or applications running on a single stor‐

age back-end can aggregate their I/O patterns in a consistent manner. However, there are

no functional semantics implied by these flags, and different I/O classes can use the

write lifetime hints in arbitrary ways, so long as the hints are used consistently.

The following operations can be applied to the file descriptor, fd:

F_GET_RW_HINT (uint64_t *; since Linux 4.13)

Returns the value of the read/write hint associated with the underlying inode re‐
ferred to by fd.

F_SET_RW_HINT (uint64_t *; since Linux 4.13)

Sets the read/write hint value associated with the underlying inode referred to by
fd. This hint persists until either it is explicitly modified or the underlying
filesystem is unmounted.

F_GET_FILE_RW_HINT (uint64_t *; since Linux 4.13)

Returns the value of the read/write hint associated with the open file description
referred to by fd.

F_SET_FILE_RW_HINT (uint64_t *; since Linux 4.13)

Sets the read/write hint value associated with the open file description referred
to by fd.

If an open file description has not been assigned a read/write hint, then it shall use the
value assigned to the inode, if any.

The following read/write hints are valid since Linux 4.13:

RWH_WRITE_LIFE_NOT_SET

No specific hint has been set. This is the default value.

RWH_WRITE_LIFE_NONE

No specific write lifetime is associated with this file or inode.

RWH_WRITE_LIFE_SHORT

Data written to this inode or via this open file description is expected to have a
short lifetime.

RWH_WRITE_LIFE_MEDIUM

Data written to this inode or via this open file description is expected to have a
lifetime longer than data written with RWH_WRITE_LIFE_SHORT.

RWH_WRITE_LIFE_LONG

Data written to this inode or via this open file description is expected to have a
lifetime longer than data written with RWH_WRITE_LIFE_MEDIUM.

RWH_WRITE_LIFE_EXTREME

Data written to this inode or via this open file description is expected to have a
lifetime longer than data written with RWH_WRITE_LIFE_LONG.

All the write-specific hints are relative to each other, and no individual absolute mean‐
ing should be attributed to them.

RETURN VALUE

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

F_DUPFD

The new file descriptor.

F_GETFD

Value of file descriptor flags.

F_GETFL

Value of file status flags.

F_GETLEASE

Type of lease held on file descriptor.

F_GETOWN

Value of file descriptor owner.

F_GETSIG

Value of signal sent when read or write becomes possible, or zero for traditional

SIGIO behavior.

F_GETPIPE_SZ, F_SETPIPE_SZ

The pipe capacity.

F_GET_SEALS

A bit mask identifying the seals that have been set for the inode referred to by

fd.

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

EBADF cmd is F_SETLK or F_SETLKW and the file descriptor open mode doesn't match with the

type of lock requested.

EBUSY cmd is F_SETPIPE_SZ and the new pipe capacity specified in arg is smaller than the

amount of buffer space currently used to store data in the pipe.

EBUSY cmd is F_ADD_SEALS, arg includes F_SEAL_WRITE, and there exists a writable, shared

    mapping on the file referred to by fd.

EDEADLK

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

EFAULT lock is outside your accessible address space.

EINTR cmd is F_SETLKW or F_OFD_SETLKW and the operation was interrupted by a signal; see

    signal(7).

EINTR cmd is F_GETLK, F_SETLK, F_OFD_GETLK, or F_OFD_SETLK, and the operation was inter?

    rupted by a signal before the lock was checked or acquired. Most likely when lock?

    ing a remote file (e.g., locking over NFS), but can sometimes happen locally.

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

EINVAL cmd is F_ADD_SEALS and arg includes an unrecognized sealing bit.

EINVAL cmd is F_ADD_SEALS or F_GET_SEALS and the filesystem containing the inode referred

    to by fd does not support sealing.

EINVAL cmd is F_DUPFD and arg is negative or is greater than the maximum allowable value

    (see the discussion of RLIMIT_NOFILE in getrlimit(2)).

EINVAL cmd is F_SETSIG and arg is not an allowable signal number.

EINVAL cmd is F_OFD_SETLK, F_OFD_SETLKW, or F_OFD_GETLK, and l_pid was not specified as

    zero.

EMFILE cmd is F_DUPFD and the per-process limit on the number of open file descriptors has

    been reached.

ENOLCK Too many segment locks open, lock table is full, or a remote locking protocol

    failed (e.g., locking over NFS).

ENOTDIR

    F_NOTIFY was specified in cmd, but fd does not refer to a directory.

EPERM cmd is F_SETPIPE_SZ and the soft or hard user pipe limit has been reached; see

    pipe(7).

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

    set.

EPERM cmd was F_ADD_SEALS, but fd was not open for writing or the current set of seals on

    the file already includes F_SEAL_SEAL.

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, and F_SETLKW are specified in POSIX.1-2001.

F_GETOWN and F_SETOWN are specified in POSIX.1-2001. (To get their definitions, define either _XOPEN_SOURCE with the value 500 or greater, or _POSIX_C_SOURCE with the value 200809L or greater.)

F_DUPFD_CLOEXEC is specified in POSIX.1-2008. (To get this definition, define _POSIX_C_SOURCE with the value 200809L or greater, or _XOPEN_SOURCE with the value 700 or greater.)

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

F_OFD_SETLK, F_OFD_SETLKW, and F_OFD_GETLK are Linux-specific (and one must define _GNU_SOURCE to obtain their definitions), but work is being done to have them included in the next version of POSIX.1.

F_ADD_SEALS and F_GET_SEALS are Linux-specific.

NOTES

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

  File locking

The original Linux fcntl() system call was not designed to handle large file offsets (in the flock structure). Consequently, an fcntl64() system call was added in Linux 2.4. The newer system call employs a different structure for file locking, flock64, and correspond? ing commands, F_GETLK64, F_SETLK64, and F_SETLKW64. However, these details can be ignored by applications using glibc, whose fcntl() wrapper function transparently employs the more recent system call where it is available.

  Record locks

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 (to iden? tify the machine where the lock is held). Clearly, l_pid alone is not going to be very useful if the process holding the lock may live on a different machine; on Linux, while present on some architectures (such as MIPS32), this field is not used.

The original Linux fcntl() system call was not designed to handle large file offsets (in the flock structure). Consequently, an fcntl64() system call was added in Linux 2.4. The newer system call employs a different structure for file locking, flock64, and correspond?

ing commands, F_GETLK64, F_SETLK64, and F_SETLKW64. However, these details can be ignored

by applications using glibc, whose fcntl() wrapper function transparently employs the more

recent system call where it is available.

Record locking and NFS

Before Linux 3.12, if an NFSv4 client loses contact with the server for a period of time

(defined as more than 90 seconds with no communication), it might lose and regain a lock

without ever being aware of the fact. (The period of time after which contact is assumed

lost is known as the NFSv4 leasetime. On a Linux NFS server, this can be determined by

looking at /proc/fs/nfsd/nfsv4leasetime, which expresses the period in seconds.   The de?

fault value for this file is 90.) This scenario potentially risks data corruption, since

another process might acquire a lock in the intervening period and perform file I/O.

Since Linux 3.12, if an NFSv4 client loses contact with the server, any I/O to the file by

a process which "thinks" it holds a lock will fail until that process closes and reopens

the file. A kernel parameter, nfs.recover_lost_locks, can be set to 1 to obtain the

pre-3.12 behavior, whereby the client will attempt to recover lost locks when contact is

reestablished with the server. Because of the attendant risk of data corruption, this pa?

rameter defaults to 0 (disabled).

BUGS

F_SETFL

It is not possible to use F_SETFL to change the state of the O_DSYNC and O_SYNC flags.

Attempts to change the state of these flags are silently ignored.

F_GETOWN

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 sys?

tem call; that is, the return value of fcntl() will be -1, and errno will contain the

(positive) process group ID. The Linux-specific F_GETOWN_EX operation avoids this prob?

lem.   Since glibc version 2.11, glibc makes the kernel F_GETOWN problem invisible by im?

plementing F_GETOWN using F_GETOWN_EX.

F_SETOWN

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.

  Deadlock detection

The deadlock-detection algorithm employed by the kernel when dealing with F_SETLKW re‐
quests can yield both false negatives (failures to detect deadlocks, leaving a set of
deadlocked processes blocked indefinitely) and false positives (EDEADLK errors when there
is no deadlock). For example, the kernel limits the lock depth of its dependency search
to 10 steps, meaning that circular deadlock chains that exceed that size will not be de‐
tected. In addition, the kernel may falsely indicate a deadlock when two or more pro‐
cesses created using the clone(2) CLONE_FILES flag place locks that appear (to the kernel)
to conflict.

  Mandatory locking

The Linux implementation of mandatory locking is subject to race conditions which render
it unreliable: a write(2) call that overlaps with a lock may modify data after the manda‐
tory 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 manda‐
tory 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),
lslocks(8)

locks.txt, mandatory-locking.txt, and dnotify.txt in the Linux kernel source directory
Documentation/filesystems/ (on older kernels, these files are directly under the Documen‐
tation/ directory, and mandatory-locking.txt is called mandatory.txt)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the
project, information about reporting bugs, and the latest version of this page, can be
found at https://www.kernel.org/doc/man-pages/.

Linux   2020-12-21   FCNTL(2)