NAME

fcntl - file control

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
fcntl(*int fd*, *int cmd*, ...);

DESCRIPTION

The **fcntl**() system call provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as described below. Depending on the value of *cmd*, **fcntl**() can take an additional third argument *arg*. Unless otherwise noted below for a specific operation, *arg* has type *int*.

F_DUPFD Return a new descriptor as follows:

- Lowest numbered available descriptor greater than or equal to *arg*.
- Same object references as the original descriptor.
- New descriptor shares the same file offset if the object was a file.
- Same access mode (read, write or read/write).
- Same file status flags (i.e., both file descriptors share the same file status flags).
- The close-on-exec flag FD_CLOEXEC associated with the new file descriptor is cleared, so the file descriptor is to remain open across execve(2) system calls.

F_DUPFD_CLOEXEC Like F_DUPFD, but the FD_CLOEXEC flag associated with the new file descriptor is set, so the file descriptor is closed when execve(2) system call executes.

F_DUP2FD It is functionally equivalent to

```
dup2(fd, arg)
```

F_DUP2FD_CLOEXEC Like F_DUP2FD, but the FD_CLOEXEC flag associated with the new file descriptor is set.

The F_DUP2FD and F_DUP2FD_CLOEXEC constants are not portable, so they should not be used if portability is needed. Use **dup2**() instead of F_DUP2FD.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fd* as FD_CLOEXEC. If the returned value ANDed with FD_CLOEXEC is 0, the file will remain open across **exec**(), otherwise the file will be closed upon execution of **exec**() (*arg* is ignored).

F_SETFD — Set the close-on-exec flag associated with *fd* to *arg*, where *arg* is either 0 or FD_CLOEXEC, as described above.

F_GETFL — Get descriptor status flags, as described below (*arg* is ignored).

F_SETFL — Set descriptor status flags to *arg*.

F_GETOWN — Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values (*arg* is ignored).

F_SETOWN — Set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

F_READAHEAD — Set or clear the read ahead amount for sequential access to the third argument, *arg*, which is rounded up to the nearest block size. A zero value in *arg* turns off read ahead, a negative value restores the system default.

F_RDAHEAD — Equivalent to Darwin counterpart which sets read ahead amount of 128KB when the third argument, *arg* is non-zero. A zero value in *arg* turns off read ahead.

F_ADD_SEALS — Add seals to the file as described below, if the underlying filesystem supports seals.

F_GET_SEALS — Get seals associated with the file, if the underlying filesystem supports seals.

F_ISUNIONSTACK — Check if the vnode is part of a union stack (either the "union" flag from mount(2) or unionfs). This is a hack not intended to be used outside of libc.

F_KINFO — Fills a *struct kinfo_file* for the file referenced by the specified file descriptor.

The *arg* argument should point to the storage for *struct kinfo_file*. The *kf_structsize* member of the passed structure must be initialized with the sizeof of *struct kinfo_file*, to allow for the interface versioning and evolution.

The flags for the F_GETFL and F_SETFL commands are as follows:

O_NONBLOCK Non-blocking I/O; if no data is available to a read(2) system call, or if a write(2) operation would block, the read or write call returns -1 with the error EAGAIN.

O_APPEND Force each write to append at the end of file; corresponds to the O_APPEND flag of open(2).

O_DIRECT Minimize or eliminate the cache effects of reading and writing. The system will attempt to avoid caching the data you read or write. If it cannot avoid caching the data, it will minimize the impact the data has on the cache. Use of this flag can drastically reduce performance if not used with care.

O_ASYNC Enable the SIGIO signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read.

O_SYNC Enable synchronous writes. Corresponds to the O_SYNC flag of open(2). O_FSYNC is an historical synonym for O_SYNC.

O_DSYNC Enable synchronous data writes. Corresponds to the O_DSYNC flag of open(2).

The seals that may be applied with F_ADD_SEALS are as follows:

F_SEAL_SEAL Prevent any further seals from being applied to the file.

F_SEAL_SHRINK Prevent the file from being shrunk with ftruncate(2).

F_SEAL_GROW Prevent the file from being enlarged with ftruncate(2).

F_SEAL_WRITE Prevent any further write(2) calls to the file. Any writes in progress will finish before **fcntl**() returns. If any writeable mappings exist, F_ADD_SEALS will fail and return EBUSY.

Seals are on a per-inode basis and require support by the underlying filesystem. If the underlying filesystem does not support seals, F_ADD_SEALS and F_GET_SEALS will fail and return EINVAL.

Several operations are available for doing advisory file locking; they all operate on the following structure:

```
struct flock {
        off_t    l_start;  /* starting offset */
        off_t    l_len;    /* len = 0 means until end of file */
        pid_t    l_pid;    /* lock owner */
        short    l_type;   /* lock type: read/write, etc. */
        short    l_whence; /* type of l_start */
        int      l_sysid;  /* remote system id or zero for local */
};
```

These advisory file locking operations take a pointer to *struct flock* as the third argument *arg*. The commands available for advisory record locking are as follows:

F_GETLK Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). The information retrieved overwrites the information passed to **fcntl**() in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged by this system call except for the lock type which is set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks, (F_WRLCK), as well as remove either type of lock (F_UNLCK). If a shared or exclusive lock cannot be set, **fcntl**() returns immediately with EAGAIN.

F_SETLKW This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied. If a signal that is to be caught is received while **fcntl**() is waiting for a region, the **fcntl**() will be interrupted if the signal handler has not specified the SA_RESTART (see sigaction(2)).

When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file was not opened with write access.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, *l_start* means end edge of the region. The *l_pid* and *l_sysid* fields are only used with F_GETLK to return the process ID of the process holding a blocking lock and the system ID of the system that owns that process. Locks created by the local system will have a system ID of zero. After a successful F_GETLK request, the value of *l_whence* is SEEK_SET.

Locks may start and extend beyond the current end of a file, but may not start or extend before the beginning of the file. A lock is set to extend to the largest possible value of the file offset for that file if *l_len* is set to zero. If *l_whence* and *l_start* point to the beginning of the file, and *l_len* is zero, the entire file is locked. If an application wishes only to do entire file locking, the flock(2) system call is much more efficient.

There is at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request fails or blocks respectively when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

The queuing for F_SETLKW requests on local files is fair; that is, while the thread is blocked, subsequent requests conflicting with its requests will not be granted, even if these requests do not conflict with existing locks.

This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks associated with a file for a given process are removed when *any* file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls getpwnam(3) to retrieve a record, the lock will be lost because getpwnam(3) opens, reads, and closes the password database. The database close will release all locks that the process has associated with the database, even if the library routine never requested a lock on the database. Another minor semantic problem with this interface is that locks are not inherited by a child process created using the fork(2) system call. The flock(2) interface has much more rational last close semantics and allows locks to be inherited by child processes. The flock(2) system call is recommended for applications that want to ensure the integrity of their locks when using library routines or wish to pass locks to their children.

The **fcntl**(), flock(2), and lockf(3) locks are compatible. Processes using different locking interfaces can

cooperate over the same file safely. However, only one of such interfaces should be used within the same process. If a file is locked by a process through flock(2), any record within the file will be seen as locked from the viewpoint of another process using **fcntl**() or lockf(3), and vice versa. Note that **fcntl**(*F_GETLK*) returns -1 in *l_pid* if the process holding a blocking lock previously locked the file descriptor by flock(2).

All locks associated with a file for a given process are removed when the process terminates.

All locks obtained before a call to execve(2) remain in effect until the new program releases them. If the new program does not know about the locks, they will not be released until the program exits.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an EDEADLK error.

RETURN VALUES

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_DUP2FD	A file descriptor equal to *arg*.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags.
F_GETOWN	Value of file descriptor owner.
other	Value other than -1.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **fcntl**() system call will fail if:

[EAGAIN]	The argument *cmd* is F_SETLK, the type of lock (*l_type*) is a shared lock (F_RDLCK) or exclusive lock (F_WRLCK), and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF] The *fd* argument is not a valid open file descriptor.

The argument *cmd* is F_DUP2FD, and *arg* is not a valid file descriptor.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a shared lock (F_RDLCK), and *fd* is not a valid file descriptor open for reading.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is an exclusive lock (F_WRLCK), and *fd* is not a valid file descriptor open for writing.

[EBUSY] The argument *cmd* is F_ADD_SEALS, attempting to set F_SEAL_WRITE, and writeable mappings of the file exist.

[EDEADLK] The argument *cmd* is F_SETLKW, and a deadlock condition was detected.

[EINTR] The argument *cmd* is F_SETLKW, and the system call was interrupted by a signal.

[EINVAL] The *cmd* argument is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see getdtablesize(2)).

The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and the data to which *arg* points is not valid.

The argument *cmd* is F_ADD_SEALS or F_GET_SEALS, and the underlying filesystem does not support sealing.

The argument *cmd* is invalid.

[EMFILE] The argument *cmd* is F_DUPFD and the maximum number of file descriptors permitted for the process are already in use, or no file descriptors greater than or equal to *arg* are available.

[ENOTTY] The *fd* argument is not a valid file descriptor for the requested operation. This may be the case if *fd* is a device node, or a descriptor returned by kqueue(2).

[ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

[EOPNOTSUPP] The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and *fd* refers to a file for which locking is not supported.

[EOVERFLOW] The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and an *off_t* calculation overflowed.

[EPERM] The *cmd* argument is F_SETOWN and the process ID or process group given as an argument is in a different session than the caller.

The *cmd* argument is F_ADD_SEALS and the F_SEAL_SEAL seal has already been set.

[ESRCH] The *cmd* argument is F_SETOWN and the process ID given as argument is not in use.

In addition, if *fd* refers to a descriptor open on a terminal device (as opposed to a descriptor open on a socket), a *cmd* of F_SETOWN can fail for the same reasons as in tcsetpgrp(3), and a *cmd* of F_GETOWN for the reasons as stated in tcgetpgrp(3).

SEE ALSO

close(2), dup2(2), execve(2), flock(2), getdtablesize(2), open(2), sigaction(2), lockf(3), tcgetpgrp(3), tcsetpgrp(3)

STANDARDS

The F_DUP2FD constant is non portable. It is provided for compatibility with AIX and Solaris.

Per Version 4 of the Single UNIX Specification ("SUSv4"), a call with F_SETLKW should fail with [EINTR] after any caught signal and should continue waiting during thread suspension such as a stop signal. However, in this implementation a call with F_SETLKW is restarted after catching a signal with a SA_RESTART handler or a thread suspension such as a stop signal.

HISTORY

The **fcntl**() system call appeared in 4.2BSD.

The F_DUP2FD constant first appeared in FreeBSD 7.1.