Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'fcntl.3p' command

$ man fcntl.3p

FCNTL(3P) POSIX Programmer's Manual FCNTL(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

fcntl ? file control

SYNOPSIS

#include <fcntl.h>

int fcntl(int fildes, int cmd, ...);

DESCRIPTION

The fcntl() function shall perform the operations described below on open files. The fildes argument is a file descriptor.

The available values for cmd are defined in <fcntl.h> and are as fol? lows:

F_DUPFD Return a new file descriptor which shall be allocated as described in Section 2.14, File Descriptor Allocation, except that it shall be the lowest numbered available file descriptor greater than or equal to the third argu? ment, arg, taken as an integer of type int. The new file descriptor shall refer to the same open file description as the original file descriptor, and shall share any

locks. The FD_CLOEXEC flag associated with the new file descriptor shall be cleared to keep the file open across calls to one of the exec functions.

F_DUPFD_CLOEXEC

Like F_DUPFD, but the FD_CLOEXEC flag associated with the new file descriptor shall be set.

F_GETFD Get the file descriptor flags defined in <fcntl.h> that are associated with the file descriptor fildes. File de? scriptor flags are associated with a single file descrip? tor and do not affect other file descriptors that refer to the same file.

F_SETFD Set the file descriptor flags defined in <fcntl.h>, that are associated with fildes, to the third argument, arg, taken as type int. If the FD_CLOEXEC flag in the third argument is 0, the file descriptor shall remain open across the exec functions; otherwise, the file descriptor shall be closed upon successful execution of one of the exec functions.

F_GETFL Get the file status flags and file access modes, defined in <fcntl.h>, for the file description associated with fildes. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h>. File status flags and file access modes are associated with the file description and do not af? fect other file descriptors that refer to the same file with different open file descriptions. The flags returned may include non-standard file status flags which the ap? plication did not set, provided that these additional flags do not alter the behavior of a conforming applica? tion.

F_SETFL Set the file status flags, defined in <fcntl.h>, for the file description associated with fildes from the corre? sponding bits in the third argument, arg, taken as type

int. Bits corresponding to the file access mode and the file creation flags, as defined in <fcntl.h>, that are set in arg shall be ignored. If any bits in arg other than those mentioned here are changed by the application, the result is unspecified. If fildes does not support non-blocking operations, it is unspecified whether the O_NONBLOCK flag will be ignored.

F_GETOWN If fildes refers to a socket, get the process ID or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values shall in? dicate a process ID; negative values, other than -1, shall indicate a process group ID; the value zero shall indicate that no SIGURG signals are to be sent. If fildes does not refer to a socket, the results are unspecified.

F_SETOWN If fildes refers to a socket, set the process ID or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, arg, taken as type int. Positive values shall indicate a process ID; negative values, other than -1, shall indicate a process group ID; the value zero shall indicate that no SIGURG signals are to be sent. Each time a SIGURG signal is sent to the specified process or process group, permission checks equivalent to those performed by kill() shall be performed, as if kill() were called by a process with the same real user ID, effective user ID, and privileges that the process calling fcntl() has at the time of the call; if the kill() call would fail, no signal shall be sent. These permission checks may also be performed by the fcntl() call. If the process specified by arg later terminates, or the process group specified by arg later becomes empty, while still being specified to receive SIGURG sig? nals when out-of-band data is available from fildes, then

no signals shall be sent to any subsequently created process that has the same process ID or process group ID, regardless of permission; it is unspecified whether this is achieved by the equivalent of a fcntl(fildes, F_SE? TOWN, 0) call at the time the process terminates or is waited for or the process group becomes empty, or by other means. If fildes does not refer to a socket, the results are unspecified.

The following values for cmd are available for advisory record locking. Record locking shall be supported for regular files, and may be sup? ported for other files.

F_GETLK Get any lock which blocks the lock description pointed to by the third argument, arg, taken as a pointer to type struct flock, defined in <fcntl.h>. The information re? trieved shall overwrite the information passed to fcntl() in the structure flock. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be 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 type struct flock, defined in <fcntl.h>. F_SETLK can establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to re? move either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in <fcntl.h>. If a shared or exclu? sive lock cannot be set, fcntl() shall return immediately with a return value of -1.

F_SETLKW This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while fc? ntl() is waiting for a region, fcntl() shall be inter?

rupted. Upon return from the signal handler, fcntl() shall return -1 with errno set to [EINTR], and the lock operation shall not be done.

Additional implementation-defined values for cmd may be defined in <fc?ntl.h>. Their names shall start with F_.

When a shared lock is set on a segment of a file, other processes shall be able to 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 shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A re?quest for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure flock describes the type (l_type), starting offset (l_whence), relative offset (l_start), size (l_len), and process ID (l_pid) of the segment of the file to be affected.

The value of l_whence is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset l_start bytes shall 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. The value of l_len may be negative (where the definition of off_t permits negative values of l_len). The l_pid field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the flock structure shall be as follows:

l_type Type of blocking lock found.

l_whence SEEK_SET.

l_start Start of the blocking lock.

l_len Length of the blocking lock.

l_pid Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall

be determined before the fcntl() function blocks. If the file size or file descriptor seek offset change while fcntl() is blocked, this shall not affect the range of bytes locked.

If l_len is positive, the area affected shall start at l_start and end at l_start+l_len-1. If l_len is negative, the area affected shall start at l_start+l_len and end at l_start-1. Locks may start and ex? tend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting l_len to 0. If such a lock also has l_start set to 0 and l_whence is set to SEEK_SET, the whole file shall be locked.

There shall be 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 re? gion specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As speci? fied above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified re? gion and the type of any of those locks conflicts with the type speci? fied in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inher? ited by a child process.

A potential for deadlock occurs if a process controlling a locked re? gion is put to sleep by attempting to lock the locked region of another process. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, fcntl() shall fail with an [EDEADLK] error.

An unlock (F_UNLCK) request in which l_len is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type off_t, when the process has an existing lock in which

l_len is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an l_len equal to 0. Otherwise, an unlock (F_UNLCK) re? quest shall attempt to unlock only the requested segment.

When the file descriptor fildes refers to a shared memory object, the behavior of fcntl() shall be the same as for a regular file except the effect of the following values for the argument cmd shall be unspeci? fied: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.

If fildes refers to a typed memory object, the result of the fcntl() function is unspecified.

RETURN VALUE

Upon successful completion, the value returned shall depend on cmd as follows:

F_DUPFD A new file descriptor.

F_DUPFD_CLOEXEC

A new file descriptor.

F_GETFD Value of flags defined in <fcntl.h>. The return value shall not be negative.

F_SETFD Value other than -1.

F_GETFL Value of file status flags and access modes. The return value is not negative.

F_SETFL Value other than -1.

F_GETLK Value other than -1.

F_SETLK Value other than -1.

F_SETLKW Value other than -1.

F_GETOWN Value of the socket owner process or process group; this will not be -1.

F_SETOWN Value other than -1.

Otherwise, -1 shall be returned and errno set to indicate the error.

ERRORS

The fcntl() function shall fail if:

EACCES or EAGAIN

The cmd argument is F_SETLK; the type of lock (l_type) is a

shared (F_RDLCK) or exclusive (F_WRLCK) lock 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 fildes argument is not a valid open file descriptor, or the argument cmd is F_SETLK or F_SETLKW, the type of lock, l_type, is a shared lock (F_RDLCK), and fildes is not a valid file de? scriptor open for reading, or the type of lock, l_type, is an exclusive lock (F_WRLCK), and fildes is not a valid file de? scriptor open for writing.

EINTR The cmd argument is F_SETLKW and the function was interrupted by a signal.

EINVAL The cmd argument is invalid, or the cmd argument is F_DUPFD or F_DUPFD_CLOEXEC and arg is negative or greater than or equal to {OPEN_MAX}, or the cmd argument is F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by arg is not valid, or fildes refers to a file that does not support locking.

EMFILE The argument cmd is F_DUPFD or F_DUPFD_CLOEXEC and all file de? scriptors available to the process are currently open, or no file descriptors greater than or equal to arg are available.

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.

EOVERFLOW

One of the values to be returned cannot be represented cor? rectly.

EOVERFLOW

The cmd argument is F_GETLK, F_SETLK, or F_SETLKW and the small? est or, if l_len is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an ob? ject of type off_t.

ESRCH The cmd argument is F_SETOWN and no process or process group can

be found corresponding to that specified by arg.

The fcntl() function may fail if:

EDEADLK

The cmd argument is F_SETLKW, the lock is blocked by a lock from another process, and putting the calling process to sleep to wait for that lock to become free would cause a deadlock.

EINVAL The cmd argument is F_SETOWN and the value of the argument is not valid as a process or process group identifier.

EPERM The cmd argument is F_SETOWN and the calling process does not have permission to send a SIGURG signal to any process specified by arg.

The following sections are informative.

EXAMPLES

Locking and Unlocking a File

The following example demonstrates how to place a lock on bytes 100 to 109 of a file and then later remove it. F_SETLK is used to perform a non-blocking lock request so that the process does not have to wait if an incompatible lock is held by another process; instead the process can take some other action.

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
int
main(int argc, char *argv[])
{
    int fd;
    struct flock fl;
    fd = open("testfile", O_RDWR);
    if (fd == -1)
        /* Handle error */;
    /* Make a non-blocking request to place a write lock
```

```
            on bytes 100-109 of testfile */
        fl.l_type = F_WRLCK;
        fl.l_whence = SEEK_SET;
        fl.l_start = 100;
        fl.l_len = 10;
        if (fcntl(fd, F_SETLK, &fl) == -1) {
            if (errno == EACCES || errno == EAGAIN) {
                printf("Already locked by another process\n");
                /* We cannot get the lock at the moment */
            } else {
                /* Handle unexpected error */;
            }
        } else { /* Lock was granted... */
            /* Perform I/O on bytes 100 to 109 of file */
            /* Unlock the locked bytes */
            fl.l_type = F_UNLCK;
            fl.l_whence = SEEK_SET;
            fl.l_start = 100;
            fl.l_len = 10;
            if (fcntl(fd, F_SETLK, &fl) == -1)
                /* Handle error */;
        }
        exit(EXIT_SUCCESS);
    } /* main */
```

Setting the Close-on-Exec Flag

The following example demonstrates how to set the close-on-exec flag for the file descriptor fd.

```
    #include <unistd.h>
    #include <fcntl.h>
    ...
        int flags;
        flags = fcntl(fd, F_GETFD);
        if (flags == -1)
```

```
    /* Handle error */;
flags |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, flags) == -1)
    /* Handle error */;"
```

APPLICATION USAGE

The arg values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow for future growth. Applications using these func? tions should do a read-modify-write operation on them, rather than as? suming that only the values defined by this volume of POSIX.1?2017 are valid. It is a common error to forget this, particularly in the case of F_SETFD. Some implementations set additional file status flags to ad? vise the application of default behavior, even though the application did not request these flags.

On systems which do not perform permission checks at the time of an fc? ntl() call with F_SETOWN, if the permission checks performed at the time the signal is sent disallow sending the signal to any process, the process that called fcntl() has no way of discovering that this has happened. A call to kill() with signal 0 can be used as a prior check of permissions, although this is no guarantee that permission will be granted at the time a signal is sent, since the target process(es) could change user IDs or privileges in the meantime.

RATIONALE

The ellipsis in the SYNOPSIS is the syntax specified by the ISO C stan? dard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

This volume of POSIX.1?2017 permits concurrent read and write access to file data using the fcntl() function; this is a change from the 1984 /usr/group standard and early proposals. Without concurrency controls, this feature may not be fully utilized without occasional loss of data.

Data losses occur in several ways. One case occurs when several pro? cesses try to update the same record, without sequencing controls; sev? eral updates may occur in parallel and the last writer ``wins''. An? other case is a bit-tree or other internal list-based database that is

undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While fcntl() is useful for many applications, it is not in? tended to be overly general and does not handle the bit-tree example well.

This facility is only required for regular files because it is not ap? propriate for many devices such as terminals and network connections. Since fcntl() works with ``any file descriptor associated with that file, however it is obtained'', the file descriptor may have been in? herited through a fork() or exec operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing an open file description, but there are too many implementations of the existing mechanism for this volume of POSIX.1?2017 to use different specifications.

Another consequence of this model is that closing any file descriptor for a given file (whether or not it is the same open file description that created the lock) causes the locks on that file to be relinquished for that process. Equivalently, any close for any file/process pair re? linquishes the locks owned on that file for that process. But note that while an open file description may be shared through fork(), locks are not inherited through fork(). Yet locks may be inherited through one of the exec functions.

The identification of a machine in a network environment is outside the scope of this volume of POSIX.1?2017. Thus, an l_sysid member, such as found in System V, is not included in the locking structure.

Changing of lock types can result in a previously locked region being split into smaller regions.

Mandatory locking was a major feature of the 1984 /usr/group standard. For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism be?

fore doing I/O on the file. Enforcement-mode record locking is impor? tant when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. En? forcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
3. Any publicly readable file could be locked by anyone. Many histori? cal implementations keep the password database in a publicly read? able file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, alarm() may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Since implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), close(), exec, kill(), open(), sigaction()

The Base Definitions volume of POSIX.1?2017, <fcntl.h>, <signal.h>

COPYRIGHT