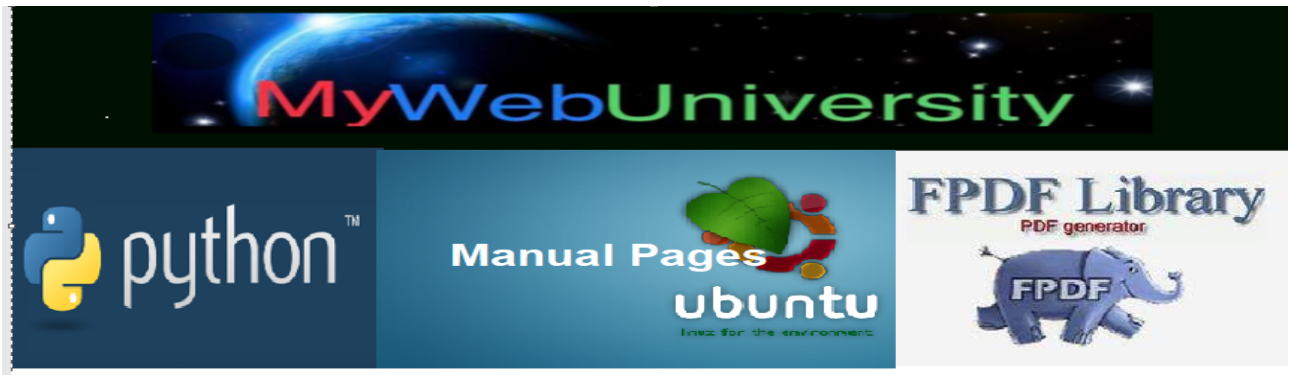

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'fcntl_locking.2'

$ man fcntl_locking.2

fcntl_locking(2) System Calls Manual fcntl_locking(2)

NAME

F_GETLK, F_SETLK, F_SETLKW, F_OFD_GETLK, F_OFD_SETLK, F_OFD_SETLKW - locking

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <fcntl.h>

int fcntl(int fd, F_GETLK, struct flock *lock);

int fcntl(int fd, F_SETLK, const struct flock *lock);

int fcntl(int fd, F_SETLKW, const struct flock *lock);

int fcntl(int fd, F_OFD_GETLK, struct flock *lock);

int fcntl(int fd, F_OFD_SETLK, const struct flock *lock);

int fcntl(int fd, F_OFD_SETLKW, const struct flock *lock);
```

DESCRIPTION

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 ar? gument, lock, is a pointer to a structure that has at least the following fields (in unspec? ified 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 since Linux 2.4.21 and Linux 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 re? gion; if a new lock is applied to an already-locked region, then the existing lock is con? verted to the new lock type. (Such conversions may involve splitting, shrinking, or coa? lescing 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

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 implementa? tions, so POSIX requires a portable application to check for both errors.)

F_SETLKW

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

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 those locks in the l_type, l_whence, l_start, and l_len fields of lock. If the conflicting lock is a traditional (process-associated) 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 in? spects 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 processes 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 processes would re? main 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 en? counters such an error should release some of its locks to allow other applications to pro?

ceed 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 ker? nel'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 in? clude this lock type in the next revision of POSIX.1.) For an explanation of open file de? scriptions, 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). Conse? quently (and unlike traditional advisory record locks), open file description locks are in? herited 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 con? flict 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), F_DUPFD(2const), 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 conversions 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 program 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 de? scriptor.

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 struc? ture must be set to zero when using the operations described below.

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

F_OFD_SETLK

Acquire an open file description lock (when l_type is F_RDLCK or F_WRLCK) or release 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 an? other process, this call returns -1 and sets errno to EAGAIN.

F_OFD_SETLKW

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 inter? rupted and (after the signal handler has returned) returns immediately (with return value -1 and errno set to EINTR; see signal(7)).

F_OFD_GETLK

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 described above for F_GETLK.

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

Mandatory locking

Warning: the Linux implementation of mandatory locking is unreliable. See BUGS below. Be? cause 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 configuration option (CONFIG_MANDATORY_FILE_LOCKING). This feature is no longer supported at all in Linux 5.15 and above.

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 re? gion 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 en? abled, 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 (including 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.

RETURN VALUE

Zero.

On error, -1 is returned, and errno is set to indicate the error.

ERRORS

See fcntl(2).

EBADF op is 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 operation would cause a deadlock.

EFAULT lock is outside your accessible address space.

EINTR op is F_SETLKW or F_OFD_SETLKW and the operation was interrupted by a signal; see signal(7).

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

EINVAL op is F_OFD_SETLK, F_OFD_SETLKW, or F_OFD_GETLK, and l_pid was not specified as zero.

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

STANDARDS

POSIX.1-2024.

HISTORY

F_GETLK

F_SETLK

F_SETLKW

4.3BSD, SVr4, POSIX.1-1988.

F_OFD_SETLK

F_OFD_SETLKW

F_OFD_GETLK

POSIX.1-2024.

NOTES

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 corresponding oper? ations, F_GETLK64, F_SETLK64, and F_SETLKW64. However, these details can be ignored by ap? plications using glibc, whose fcntl() wrapper function transparently employs the more recent system call where it is available.

Record locks

Since Linux 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 identify 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 corresponding oper? ations, F_GETLK64, F_SETLK64, and F_SETLKW64. However, these details can be ignored by ap? plications 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 default 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 reestab? lished with the server. Because of the attendant risk of data corruption, this parameter defaults to 0 (disabled).

BUGS

Deadlock detection

The deadlock-detection algorithm employed by the kernel when dealing with F_SETLKW requests 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 dead? lock). 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 detected. In addi? tion, the kernel may falsely indicate a deadlock when two or more processes 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 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

fcntl(2), flock(2), lockf(3), lslocks(8)

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