NAME fcntl – file control

SYNOPSIS

#include <sys/types.h>
#include <fcntl.h>

int fcntl(int *fildes*, **int** *cmd*, **/*** *arg* ***/ . . .);**

MT-LEVEL Async_Signal_Safe

DESCRIPTION

fcntl( ) provides for control over open files. *fildes* is an open file descriptor (see **intro**(2)).

fcntl( ) may take a third argument, *arg*, whose data type, value and use depend upon the value of *cmd*. *cmd* specifies the operation to be performed by **fcntl( )** and may be one of the following:

F_DUPFD Return a new file descriptor with the following characteristics:

Lowest numbered available file descriptor greater than or equal to the integer value given as the third argument.

Same open file (or pipe) as the original file.

Same file pointer as the original file (that is, both file descriptors share one file pointer).

Same access mode (read, write, or read/write) as the original file.

Shares any locks associated with the original file descriptor.

Same file status flags (that is, both file descriptors share the same file status flags) as the original file.

The close-on-exec flag (see **F_GETFD**) associated with the new file descriptor is set to remain open across **exec**(2) functions.

F_GETFD Get the close-on-exec flag associated with *fildes*. If the low-order bit is **0**, the file will remain open across **exec**. Otherwise, the file will be closed upon execution of **exec**.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of the integer value given as the third argument (**0** or **1** as above).

F_GETFL Get *fildes* status flags.

F_SETFL Set *fildes* status flags to the integer value given as the third argument. Only certain flags can be set (see **fcntl**(5)).

F_FREESP Free storage space associated with a section of the ordinary file *fildes*. The section is specified by a variable of data type **struct flock** pointed to by the third argument *arg*. The data type **struct flock** is defined in the **<fcntl.h>** header (see **fcntl**(5)) and contains the following members: **l_whence** is 0, 1, or 2 to indicate that the relative offset **l_start** will be measured from the start of the file, the current position, or the end of the file, respectively. **l_start** is the offset from the position specified in **l_whence**. **l_len** is the size of the section.

An **l_len** of 0 frees up to the end of the file; in this case, the end of file (that is, file size) is set to the beginning of the section freed. Any data previously written into this section is no longer accessible.

Note that all filesystems might not support all possible variations of F_FREESP arguments. In particular, many filesystems only allow space to be freed at the end of a file.

The following values for *cmd* are used for record-locking. Locks may be placed on an entire file or on segments of a file.

F_SETLK Set or clear a file segment lock according to the **flock** structure that *arg* points to (see **fcntl**(5)). The *cmd* **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as remove either type of lock (**F_UNLCK**). If a read or write lock cannot be set, **fcntl( )** will return immediately with an error value of −1.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, **fcntl( )** will block until the segment is free to be locked.

F_GETLK If the lock request described by the **flock** structure that *arg* points to could be created, then the structure is passed back unchanged except that the lock type is set to **F_UNLCK** and the **l_whence** field will be set to **SEEK_SET**.

If a lock is found that would prevent this lock from being created, then the structure is overwritten with a description of the first lock that is preventing such a lock from being created. The structure also contains the process ID and the system ID of the process holding the lock.

This command never creates a lock; it tests whether a particular lock could be created.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock and no read locks may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The record to be locked or unlocked is described by the flock structure defined in **<sys/fcntl.h>** (included in **<fcntl.h>**) as follows:

```
typedef struct flock {
        short      l_type;
        short      l_whence;
        off_t      l_start;
        off_t      l_len;       /* len == 0 means until end of file */
        long       l_sysid;
        pid_t      l_pid;
```

```
        long        pad[4];      /* reserve area */
} flock_t;
```

The **flock** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), process ID (**l_pid**), and system ID (**l_sysid**) of the segment of the file to be affected. The process ID and system ID fields are used only with the **F_GETLK** *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting **l_len** to **0**. If such a lock also has **l_whence** and **l_start** set to **0**, the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a **fork**(2) function.

When mandatory file and record locking is active on a file (see **chmod**(2)), **creat**(2), **open**(2), **read**(2) and **write**(2) functions issued on the file will be affected by the record locks in effect. When mandatory file and record locking is active on a file, it cannot be memory mapped.

RETURN VALUES

On success, **fcntl( )** returns a value that depends on *cmd*:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined). The return value will not be negative.
F_SETFD	Value other than –1.
F_FREESP	Value of 0.
F_GETFL	Value of file status flags. The return value will not be 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.

On failure, **fcntl( )** returns –1 and sets **errno** to indicate the error.

ERRORS

fcntl( ) will fail if one or more of the following are true:

EAGAIN	*cmd* is **F_SETLK**, the type of lock (**l_type**) is a read lock (**F_RDLCK**) and the segment of a file to be locked is already write locked by another process, or the type is a write lock (**F_WRLCK**) and the segment of a file to be locked is already read or write locked by another process. Note that in the past this function was returned as **EACCES**.
EAGAIN	*cmd* is **F_FREESP**, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file.

EAGAIN *cmd* is **F_SETLK** or **F_SETLKW**, mandatory file/record locking is set, and the file is currently being mapped to virtual memory using **mmap**(2).

EBADF *fildes* is not a valid open file descriptor.

EBADF *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a read lock (**F_RDLCK**), and *fildes* is not a valid file descriptor open for reading.

EBADF *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a write lock (**F_WRLCK**), and *fildes* is not a valid file descriptor open for writing.

EBADF *cmd* is **F_FREESP**, and *fildes* is not a valid file descriptor open for writing.

EDEADLK *cmd* is **F_SETLKW**, the lock is blocked by some lock from another process, and if **fcntl( )** blocked the calling process waiting for that lock to become free, a deadlock would occur.

EDEADLK *cmd* is **F_FREESP**, mandatory record locking is enabled, **O_NDELAY** and **O_NONBLOCK** are clear and a deadlock condition was detected.

EFAULT *cmd* is **F_FREESP** and the third argument *arg* points to an illegal address.

EFAULT *cmd* is **F_GETLK**, **F_SETLK** or **F_SETLKW** and the third argument points to an illegal address.

EINTR A signal was caught during execution of the **fcntl( )** function.

EINVAL *cmd* is **F_DUPFD** and the third argument is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.

EINVAL *cmd* is not a valid value.

EINVAL *cmd* is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the third argument or the data it points to is not valid, or *fildes* refers to a file that does not support locking.

EIO An I/O error occurred while reading from or writing to the file system.

EMFILE *cmd* is **F_DUPFD** and the number of file descriptors currently open in the calling process is the configured value for the maximum number of open file descriptors allowed each user.

ENOLCK *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

ENOLINK *cmd* is **F_FREESP**, the file is on a remote machine, and the link to that machine is no longer active.

EOVERFLOW *cmd* is **F_GETLK** and the process ID of the process holding the requested lock is too large to be stored in the *l_pid* field.

SEE ALSO

lockd(1M), **chmod**(2), **close**(2), **creat**(2), **dup**(2), **exec**(2), **fork**(2), **open**(2), **pipe**(2), **read**(2), **write**(2), **fcntl**(5)

System Interfaces Guide

WARNINGS

Mandatory record locks are dangerous. If a runaway or otherwise out-of-control process should hold a mandatory lock on a file critical to the system and fail to release that lock, the entire system could hang or crash. For this reason, mandatory record locks may be removed in a future SunOS release. Use advisory record locking whenever possible.

NOTES

In the past, the variable **errno** was set to **EACCES** rather than **EAGAIN** when a section of a file is already locked by another process. Therefore, portable application programs should expect and test for either value.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access. Files can be accessed without advisory locks, but inconsistencies may result.

read(2) and **write**(2) system calls on files are affected by mandatory file and record locks (see **chmod**(2)).