NAME

close − close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```

DESCRIPTION

close closes a file descriptor, so that it no longer refers to any file and may be reused. Any locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using **unlink**(2) the file is deleted.

RETURN VALUE

close returns zero on success, or −1 if an error occurred.

ERRORS

EBADF *fd* isn't a valid open file descriptor.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 documents an additional ENOLINK error condition.

NOTES

Not checking the return value of close is a common but nevertheless serious programming error. File system implementations which use techniques as ''write−behind'' to increase performance may lead to *write*(2) succeeding, although the data has not been written yet. The error status may be reported at a later write operation, but it is guaranteed to be reported on closing the file. Not checking the return value when closing the file may lead to silent loss of data. This can especially be observed with NFS and disk quotas.

SEE ALSO

open(2), **fcntl**(2), **shutdown**(2), **unlink**(2), **fclose**(3)

NAME

lseek − reposition read/write file offset

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

The **lseek** function repositions the offset of the file descriptor *fildes* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET
The offset is set to *offset* bytes.

SEEK_CUR
The offset is set to its current location plus *offset* bytes.

SEEK_END
The offset is set to the size of the file plus *offset* bytes.

The **lseek** function allows the file offset to be set beyond the end of the existing end−of−file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

RETURN VALUES

Upon successful completion, **lseek** returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of (off_t)−1 is returned and *errno* is set to indicate the error.

ERRORS

EBADF *Fildes* is not an open file descriptor.

ESPIPE *Fildes* is associated with a pipe, socket, or FIFO.

EINVAL *Whence* is not a proper value.

CONFORMING TO

SVr4, POSIX, BSD 4.3

RESTRICTIONS

Some devices are incapable of seeking and POSIX does not specify which devices must support it.

Linux specific restrictions: using **lseek** on a tty device returns **ESPIPE**. Other systems return the number of written characters, using SEEK_SET to set the counter. Some devices, e.g. **/dev/null** do not cause the error **ESPIPE**, but return a pointer which value is undefined.

SEE ALSO

dup(2), **open**(2), **fseek**(3)

NAME

open, creat − open and possibly create a file or device

SYNOPSIS

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

DESCRIPTION

The **open()** system call is used to convert a pathname into a file descriptor (a small, non−negative integer for use in subsequent I/O as with **read**, **write**, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the **fork**(2) system call.) The new file descriptor is set to remain open across exec functions (see **fcntl**(2)). The file offset is set to the beginning of the file.

flags is one of **O_RDONLY**, **O_WRONLY** or **O_RDWR** which request opening the file read−only, write−only or read/write, respectively.

flags may also be bitwise−*or*'d with one or more of the following:

O_CREAT If the file does not exist it will be created.

O_EXCL When used with **O_CREAT**, if the file already exists it is an error and the **open** will fail. **O_EXCL** is broken on NFS file systems, programs which rely on it for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same fs (e.g., incorporating hostname and pid), use **link**(2) to make a link to the lockfile. If **link()** returns 0, the lock is successful. Otherwise, use **stat**(2) on the unique file to check if its link count has increased to 2, in which case the lock is also successful.

O_TRUNC If the file already exists it will be truncated.

O_APPEND

The file is opened in append mode. Before each **write**, the file pointer is positioned at the end of the file, as if with **lseek**. **O_APPEND** may lead to corrupted files on NFS file systems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_NONBLOCK or **O_NDELAY**
The file is opened in non-blocking mode. Neither the **open** nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait. For the handling of FIFOs (named pipes), see also **fifo**(4).

Some of these optional flags can be altered using **fcntl** after the file has been opened.

mode specifies the permissions to use if a new file is created. It is modified by the process's **umask** in the usual way: the permissions of the created file are **(mode & ~umask)**.

The following symbolic constants are provided for *mode*:

S_IRWXU
00700 user (file owner) has read, write and execute permission

S_IRUSR (S_IREAD)
00400 user has read permission

S_IWUSR (S_IWRITE)
00200 user has write permission

S_IXUSR (S_IEXEC)
00100 user has execute permission

S_IRWXG
00070 group has read, write and execute permission

S_IRGRP
00040 group has read permission

S_IWGRP
00020 group has write permission

S_IXGRP
00010 group has execute permission

S_IRWXO
00007 others have read, write and execute permission

S_IROTH
00004 others have read permission

S_IWOTH
00002 others have write permisson

S_IXOTH
00001 others have execute permission

mode should always be specified when **O_CREAT** is in the *flags*, and is ignored otherwise.

creat is equivalent to **open** with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE

open and **creat** return the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately). Note that **open** can open device special files, but **creat** cannot create them − use **mknod**(2) instead.

On NFS file systems with UID mapping enabled, **open** may return a file descriptor but e.g. **read**(2) requests are denied with **EACCES**. This is because the client performs **open** by checking the permissions, but UID mapping is performed by the server upon read and write requests.

ERRORS

EEXIST *pathname* already exists and **O_CREAT** and **O_EXCL** were used.

EISDIR *pathname* refers to a directory and the access requested involved writing.

EACCES The requested access to the file is not allowed, or one of the directories in *pathname* did not allow search (execute) permission, or the file did not exist yet and write access to the parent directory is not allowed.

ENAMETOOLONG
pathname was too long.

ENOENT A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOTDIR A component used as a directory in *pathname* is not, in fact, a directory, or **O_DIRECTORY** was specified and *pathname* was not a directory.

ENXIO O_NONBLOCK | O_WRONLY is set, the named file is a FIFO and no process has the file open for reading. Or, the file is a device special file and no corresponding device exists.

EROFS *pathname* refers to a file on a read−only filesystem and write access was requested.

ETXTBSY *pathname* refers to an executable image which is currently being executed and write access was requested.

EFAULT *pathname* points outside your accessible address space.

ELOOP Too many symbolic links were encountered in resolving *pathname*, or **O_NOFOLLOW** was specified but *pathname* was a symbolic link.

ENOSPC *pathname* was to be created but the device

containing *pathname* has no room for the new file.

ENOMEM Insufficient kernel memory was available.

EMFILE The process already has the maximum number of files open.

ENFILE The limit on the total number of files open on the system has been reached.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

RESTRICTIONS

There are many infelicities in the protocol underlying NFS, affecting amongst others **O_SYNC** and **O_NDELAY**.

POSIX provides for three different variants of synchronised I/O, corresponding to the flags **O_SYNC**, **O_DSYNC** and **O_RSYNC**. Currently (2.1.130) these are all synonymous under Linux.

SEE ALSO

read(2), **write**(2), **fcntl**(2), **close**(2), **link**(2), **mknod**(2), **mount**(2), **stat**(2), **umask**(2), **unlink**(2), **socket**(2), **fopen**(3), **fifo**(4)

NAME

read − read from a file descriptor

SYNOPSIS

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read()** returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end−of−file, or because we are reading from a pipe, or from a terminal), or because **read()** was interrupted by a signal. On error, −1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

ERRORS

EINTR The call was interrupted by a signal before any data was read.

EAGAIN Non−blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its process group is orphaned. It may also occur when there is a low−level I/O error while reading from a disk or tape.

EISDIR *fd* refers to a directory.

EBADF *fd* is not a valid file descriptor or is not open for reading.

EINVAL *fd* is attached to an object which is unsuitable for reading.

EFAULT *buf* is outside your accessible address space.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read** that is interrupted after reading some data to return −1 (with *errno* set to EINTR) or to return the number of bytes already read.

CONFORMING TO

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

RESTRICTIONS

On NFS file systems, reading small amounts of data will only update the time stamp the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave atime updates to the server and client side reads satisfied from the client's cache will not cause atime updates on the server as there are no server side reads. UNIX semantics can be obtained by disabling client side attribute caching, but in most situations this will substantially increase server load and decrease performance.

SEE ALSO

readdir(2), **write**(2), **write**(2), **fcntl**(2), **close**(2), **lseek**(2), **select**(2), **readlink**(2), **ioctl**(2), **fread**(3).

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd*, **const void ***buf*, **size_t** *count*);**

DESCRIPTION

write writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf.* POSIX requires that a **read()** which can be proved to occur after a **write()** has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, −1 is returned, and *errno* is set appropriately. If *count* is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.

ERRORS

EBADF *fd* is not a valid file descriptor or is not open for writing.

EINVAL *fd* is attached to an object which is unsuitable for writing.

EFAULT *buf* is outside your accessible address space.

EPIPE *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will receive a **SIGPIPE** signal; if it catches, blocks or ignores this the error **EPIPE** is returned.

EAGAIN Non−blocking I/O has been selected using **O_NONBLOCK** and there was no room in the pipe or socket connected to *fd* to write the data immediately.

EINTR The call was interrupted by a signal before any data was written.

ENOSPC The device containing the file referred to by *fd* has no room for the data.

Other errors may occur, depending on the object connected to *fd*.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, 4.3BSD. SVr4 documents additional error conditions EDEADLK, EFBIG, ENOLCK, ENOLNK, ENOSR, ENXIO, EPIPE, or ERANGE. Under SVr4 a write may be interrupted and return EINTR at any point, not just before any data is written.

SEE ALSO

open(2), **read**(2), **fcntl**(2), **close**(2), **lseek**(2), **select**(2), **ioctl**(2), **fsync**(2), **fwrite**(3).