NAME

close − close a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

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 record locks (see **fcntl**(2)) 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 file descriptor referring to the underlying open file description (see **open**(2)), the resources associated with the open file description are freed; if the file 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. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
fd isn't a valid open file descriptor.

EINTR
The **close**() call was interrupted by a signal; see **signal**(7).

EIO An I/O error occurred.

ENOSPC, **EDQUOT**
On NFS, these errors are not normally reported against the first write which exceeds the available storage space, but instead against a subsequent **write**(2), **fsync**(2), or **close**().

See NOTES for a discussion of why **close**() should not be retried after an error.

STANDARDS

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use **fsync**(2). (It will depend on the disk hardware at this point.)

The close-on-exec file descriptor flag can be used to ensure that a file descriptor is automatically closed upon a successful **execve**(2); see **fcntl**(2) for details.

Multithreaded processes and close()

It is probably unwise to close file descriptors while they may be in use by system calls in other threads in the same process. Since a file descriptor may be reused, there are some obscure race conditions that may cause unintended side effects.

Furthermore, consider the following scenario where two threads are performing operations on the same file descriptor:

(1) One thread is blocked in an I/O system call on the file descriptor. For example, it is trying to **write**(2) to a pipe that is already full, or trying to **read**(2) from a stream socket which currently has no available data.

(2) Another thread closes the file descriptor.

The behavior in this situation varies across systems. On some systems, when the file descriptor is closed, the blocking system call returns immediately with an error.

On Linux (and possibly some other systems), the behavior is different: the blocking I/O system call holds a reference to the underlying open file description, and this reference keeps the description open until the I/O system call completes. (See **open**(2) for a discussion of open file descriptions.) Thus, the blocking system call in the first thread may successfully complete after the **close**() in the second thread.

Dealing with error returns from close()

A careful programmer will check the return value of **close**(), since it is quite possible that errors on a previous **write**(2) operation are reported only on the final **close**() that releases the open file description. Failing to check the return value when closing a file may lead to *silent* loss of data. This can especially be observed with NFS and with disk quota.

Note, however, that a failure return should be used only for diagnostic purposes (i.e., a warning to the application that there may still be I/O pending or there may have been failed I/O) or remedial purposes (e.g., writing the file once more or creating a backup).

Retrying the **close**() after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to be closed. This can occur because the Linux kernel *always* releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Many other implementations similarly always close the file descriptor (except in the case of **EBADF**, meaning that the file descriptor was invalid) even if they subsequently report an error on return from **close**(). POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

A careful programmer who wants to know about I/O errors may precede **close**() with a call to **fsync**(2).

The **EINTR** error is a somewhat special case. Regarding the **EINTR** error, POSIX.1-2008 says:

> If **close**() is interrupted by a signal that is to be caught, it shall return −1 with *errno* set to **EINTR** and the state of *fildes* is unspecified.

This permits the behavior that occurs on Linux and many other implementations, where, as with other errors that may be reported by **close**(), the file descriptor is guaranteed to be closed. However, it also permits another possibility: that the implementation returns an **EINTR** error and keeps the file descriptor open. (According to its documentation, HP-UX's **close**() does this.) The caller must then once more use **close**() to close the file descriptor, to avoid file descriptor leaks. This divergence in implementation behaviors provides a difficult hurdle for portable applications, since on many implementations, **close**() must not be called again after an **EINTR** error, and on at least one, **close**() must be called again. There are plans to address this conundrum for the next major release of the POSIX.1 standard.

SEE ALSO

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