NAME
write − write to a file descriptor

LIBRARY
Standard C library (*libc*, *−lc*)

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see **setrlimit**(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also **pipe**(7).)

For a seekable file (i.e., one to which **lseek**(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was **open**(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a **read**(2) that can be proved to occur after a **write**() has returned will return the new data. Note that not all filesystems are POSIX conforming.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE
On success, the number of bytes written is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note that a successful **write**() may transfer fewer than *count* bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked **write**() to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes. In the event of a partial write, the caller can make another **write**() call to transfer the remaining bytes. The subsequent call will either transfer further bytes or may result in an error (e.g., if the disk is now full).

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if one of the errors below is detected. If no errors are detected, or error detection is not performed, 0 is returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

ERRORS
EAGAIN
> The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. See **open**(2) for further details on the **O_NON-BLOCK** flag.

EAGAIN or **EWOULDBLOCK**
> The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

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

EDESTADDRREQ
> *fd* refers to a datagram socket for which a peer address has not been set using **connect**(2).

EDQUOT

> The user's quota of disk blocks on the filesystem containing the file referred to by *fd* has been exhausted.

EFAULT

> *buf* is outside your accessible address space.

EFBIG

> An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process's file size limit, or to write at a position past the maximum allowed offset.

EINTR

> The call was interrupted by a signal before any data was written; see **signal**(7).

EINVAL

> *fd* is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EIO   A low-level I/O error occurred while modifying the inode. This error may relate to the write-back of data written by an earlier **write**(), which may have been issued to a different file descriptor on the same file. Since Linux 4.13, errors from write-back come with a promise that they *may* be reported by subsequent. **write**() requests, and *will* be reported by a subsequent **fsync**(2) (whether or not they were also reported by **write**()). An alternate cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost locks* section of **fcntl**(2) for further details.

ENOSPC

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

EPERM

> The operation was prevented by a file seal; see **fcntl**(2).

EPIPE   *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a **SIGPIPE** signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)

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

STANDARDS

SVr4, 4.3BSD, POSIX.1-2001.

Under SVr4 a write may be interrupted and return **EINTR** at any point, not just before any data is written.

NOTES

The types *size_t* and *ssize_t* are, respectively, unsigned and signed integer data types specified by POSIX.1.

A successful return from **write**() does not make any guarantee that data has been committed to disk. On some filesystems, including NFS, it does not even guarantee that space has successfully been reserved for the data. In this case, some errors might be delayed until a future **write**(), **fsync**(2), or even **close**(2). The only way to be sure is to call **fsync**(2) after you are done writing all your data.

If a **write**() is interrupted by a signal handler before any bytes are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.

On Linux, **write**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

An error return value while performing **write**() using direct I/O does not mean the entire write has failed. Partial data may be written and the data at the file offset on which the **write**() was attempted should be considered inconsistent.

BUGS

According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **write**() and **writev**(2). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, before Linux 3.14, this was not the case: if two processes that share an open file description (see **open**(2)) perform a **write**() (or **writev**(2)) at the same time, then the I/O operations were not atomic with respect to updating the file offset, with the result that the blocks of data output by the two processes might (incorrectly) overlap. This problem was fixed in Linux 3.14.

SEE ALSO

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