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 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 appropriately.

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**(2).

NAME

calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t *nmemb***, size_t** *size***);**
void *malloc(size_t *size***);**
void free(void **ptr***);*
void *realloc(void **ptr***, **size_t** *size***);**

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(***ptr***)** has already been called before, undefined behaviour occurs. If *ptr* is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free(***ptr***)**. Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory,

which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to *free*() is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign**(3)

NAME

open – open and possibly create a file

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***);**

DESCRIPTION

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, non-negative integer that is used in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

The full list of file creation flags and file status flags is as follows:

O_CREAT

If *pathname* does not exist, create it as a regular file.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** or **O_TMPFILE** is specified in *flags*; if neither **O_CREAT** nor **O_TMPFILE** is specified, then *mode* is ignored. The effective mode is modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created file is

(mode & ~umask). Note that this mode applies only to future accesses of the newly created file; the **open**() call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU S_IRUSR S_IWUSR S_IXUSR S_IRWXG
S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH
S_IXOTH

RETURN VALUE

open() return the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately).

ERRORS

open() can fail with the following errors:

EACCES

The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *pathname*, or the file did not exist yet and write access to the parent directory is not allowed. (See also **path_resolution**(7).)

...

NAME

read – read from a file descriptor

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

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

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

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 to indicate the error. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS

EAGAIN

The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block.

See **open**(2) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**
The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the read 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 reading.

EFAULT
buf is outside your accessible address space.

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

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

EINVAL
fd was created via a call to **timerfd_create**(2) and the wrong size buffer was given to **read**(); see **timerfd_create**(2) for further information.

EIO I/O error.

EISDIR
fd refers to a directory.

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

STANDARDS
POSIX.1-2008.

SEE ALSO
close(2), **fcntl**(2), **ioctl**(2), **lseek**(2), **open**(2), **pread**(2), **readdir**(2), **readlink**(2), **readv**(2), **select**(2), **write**(2), **fread**(3)

NAME

sigaction – POSIX signal handling functions.

SYNOPSIS

#include <signal.h>

int sigaction(int *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);

DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non–null, the new action for signal *signum* is installed from *act*. If *oldact* is non–null, the previous action is saved in *oldact*.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void      (*sa_handler)(int signal_number);
    sigset_t  sa_mask;
    int       sa_flags;
}
```

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** or **SA_NOMASK** flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP

If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when child processes receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**).

SA_RESTART

Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals. Without **SA_RESTART** the system calls return an error and set errno to EINTR when interrupted by a signal.

RETURN VALUES

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL

An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught.

SEE ALSO

kill(1), **kill**(2), **killpg**(2), **pause**(2), **sigsetops**(3),

NAME

time – get time in seconds

SYNOPSIS

#include <time.h>

time_t time(time_t *_tloc_);

DESCRIPTION

time() returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

If *tloc* is non-NULL, the return value is also stored in the memory pointed to by *tloc*.

When *tloc* is NULL, the call cannot fail.

RETURN VALUE

On success, the value of time in seconds since the Epoch is returned. On error, *((time_t) −1)* is returned, and *errno* is set appropriately.

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 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.

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_NONBLOCK** 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.

EFAULT
buf is outside your accessible address space.

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

EIO A low-level I/O error occurred while modifying the inode.

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

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*.

NOTES

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 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.

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.

SEE ALSO

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