Summary of open, read, write, close

Open():

```
#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);

int openat(int dirfd, const char *pathname, int flags);

int openat(int dirfd, const char *pathname, int flags, mode_t mode);
```

opens file by 'pathname', if file doesn't exist, O_CREAT in 'flag' can create it

return value is file descriptor.

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 file creation flags are O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE, and O_TRUNC.

The file status flags are all of the remaining flags listed below.

O_APPEND

The file is opened in append mode. Before each write(2), the

file offset is positioned at the end of the file, as if with

lseek(2).

O_ASYNC

This feature is

available only for terminals, pseudoterminals, sockets, and (since Linux 2.6) pipes and FIFOs.

O_CLOEXEC

Enable the close-on-exec flag for the new file descriptor.

Specifying this flag permits a program to avoid additional

fcntl(2) F_SETFD operations to set the FD_CLOEXEC flag.

O_CREAT

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

The owner (user ID) of the new file is set to the effective user ID of the process.

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

S_IRUSR 00400 user has read permission

S_IWUSR 00200 user has write permission

S_IXUSR 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 permissio

S_IRWXO 00007 others have read, write, and execute permission

S_IROTH 00004 others have read permission

S_IWOTH 00002 others have write permission

S_IXOTH 00001 others have execute permission

S_ISUID 0004000 set-user-ID bit

S_ISGID 0002000 set-group-ID bit (see inode(7)).

O_DIRECT

try to minimize cache effects of the I/O to and from this file.

O_DIRECTORY

If pathname is not a directory, cause the open to fail.

O_DSYNC

Write operations on the file will complete according to the requirements of synchronized I/O data integrity completion.

O_EXCL Ensure that this call creates the file: if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() fails with the error EEXIST.

O_LARGEFILE

(LFS) Allow files whose sizes cannot be represented in an off_t (but can be represented in an off64_t) to be opened.

O_NOATIME

Do not update the file last access time when the file is read(2).

O_NOCTTY

If pathname refers to a terminal device—see tty(4)—it will not become the process's controlling terminal even if the process does not have one.

O_NOFOLLOW

If pathname is a symbolic link, then the open fails, with the error ELOOP.

O_NONBLOCK or O_NDELAY

When possible, the file is opened in nonblocking mode. Neither the open() nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

O_PATH

Obtain a file descriptor that can be used for two purposes: to indicate a location in the filesystem tree and to perform operations that act purely at the file descriptor level.

O_SYNC Write operations on the file will complete according to the

requirements of synchronized I/O file integrity completion

O_TMPFILE

Create an unnamed temporary regular file.

O_TRUNC

If the file already exists and is a regular file and the access

mode allows writing (i.e., is O_RDWR or O_WRONLY) it will be

truncated to length 0. If the file is a FIFO or terminal device

file, the O_TRUNC flag is ignored.

open(), openat(), and creat() return the new file descriptor, or -1 if error occurred

Read():

#include <unistd.h>

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

read 'count' size of characters starting from 'buf' at file 'fd'.

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.

On success, the number of bytes read is returned, 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.

On error, -1 is returned

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.

EAGAIN or EWOULDBLOCK

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

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. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, 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.

Write():

#include <unistd.h>

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

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 there is insufficient space on the underlying physical medium, or the RLIMIT_FSIZE resource limit is encountered, or the was interrupted by a signal handler after having written less than count bytes.

On success, the number of bytes written is returned. On error, -1 is returned.

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 will be returned without causing any other effect.

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.

EAGAIN or EWOULDBLOCK

The file descriptor fd refers to a socket and has been marked

nonblocking (O_NONBLOCK), and the write would block.

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 writ-

ten;

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(2), which may have been issued to a different file
descriptor on the same file.

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.)

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(2), 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.

Close():

#include <unistd.h>

int close(int fd);

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

If fd is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed

close() returns zero on success. On error, -1 is returned

ERRORS

EBADF fd isn't a valid open file descriptor.

EINTR The close() call was interrupted by a signal;

EIO An I/O error occurred.

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).