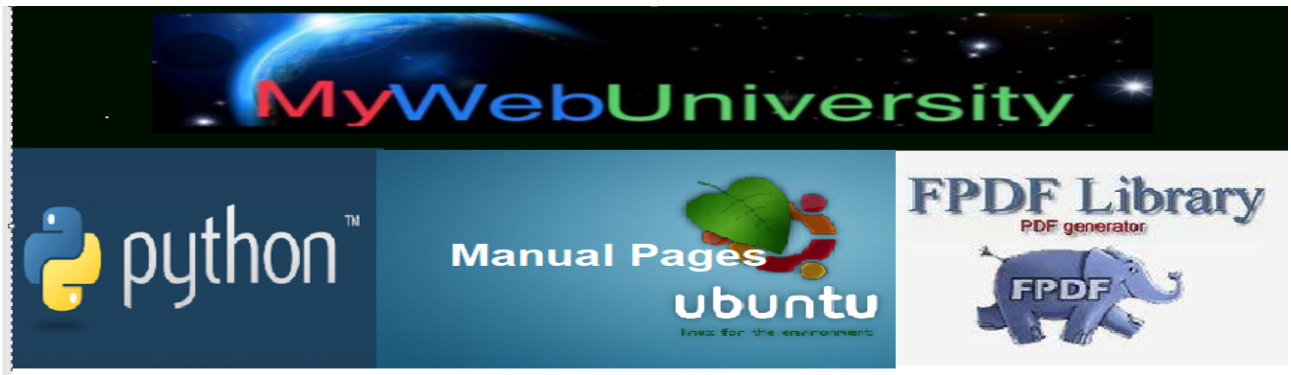

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'open.2'

$ man open.2

open(2) System Calls Manual open(2)

NAME

open, openat, creat - open and possibly create a file

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <fcntl.h>

int open(const char *path, int flags, ...
        /* mode_t mode */ );

int creat(const char *path, mode_t mode);

int openat(int dirfd, const char *path, int flags, ...
        /* mode_t mode */ );

/* Documented separately, in openat2(2): */

int openat2(int dirfd, const char *path,
        const struct open_how *how, size_t size);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

openat():

Since glibc 2.10:

_POSIX_C_SOURCE >= 200809L

Before glibc 2.10:

_ATFILE_SOURCE

DESCRIPTION

The open() system call opens the file specified by path. If the specified file does not ex?

ist, 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, nonnegative integer that is an in? dex to an entry in the process's table of open file descriptors. The file descriptor 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-num? bered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an execve(2) (i.e., the FD_CLOEXEC file descriptor flag described in fcntl(2) is initially disabled); the O_CLOEXEC flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see lseek(2)).

A call to open() creates a new open file description, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if path is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

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.

In addition, zero or more file creation flags and file status flags can be bitwise ORed in flags. The file creation flags are O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NO? FOLLOW, O_TMPFILE, and O_TRUNC. The file status flags are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see fcntl(2) for details.

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

O_APPEND

The file is opened in append mode. Before each write(2), the file offset is posi? tioned at the end of the file, as if with lseek(2). The modification of the file offset and the write operation are performed as a single atomic step.

O_APPEND may lead to corrupted files on NFS filesystems if more than one process ap? pends 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_ASYNC

Enable signal-driven I/O: generate a signal (SIGIO by default, but this can be changed via fcntl(2)) when input or output becomes possible on this file descriptor. This feature is available only for terminals, pseudoterminals, sockets, and (since Linux 2.6) pipes and FIFOs. See fcntl(2) for further details. See also BUGS, below.

O_CLOEXEC (since Linux 2.6.23)

Enable the close-on-exec flag for the new file descriptor. Specifying this flag per? mits a program to avoid additional fcntl(2) F_SETFD operations to set the FD_CLOEXEC flag.

Note that the use of this flag is essential in some multithreaded programs, because using a separate fcntl(2) F_SETFD operation to set the FD_CLOEXEC flag does not suf? fice to avoid race conditions where one thread opens a file descriptor and attempts to set its close-on-exec flag using fcntl(2) at the same time as another thread does a fork(2) plus execve(2). Depending on the order of execution, the race may lead to the file descriptor returned by open() being unintentionally leaked to the program executed by the child process created by fork(2). (This kind of race is in principle possible for any system call that creates a file descriptor whose close-on-exec flag should be set, and various other Linux system calls provide an equivalent of the O_CLOEXEC flag to deal with this problem.)

O_CREAT

If path 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.

The group ownership (group ID) of the new file is set either to the effective group ID of the process (System V semantics) or to the group ID of the parent directory (BSD semantics). On Linux, the behavior depends on whether the set-group-ID mode bit is set on the parent directory: if that bit is set, then BSD semantics apply; other? wise, System V semantics apply. For some filesystems, the behavior also depends on the bsdgroups and sysvgroups mount options described in mount(8).

The mode argument specifies the file mode bits to be applied when a new file is cre? ated. If neither O_CREAT nor O_TMPFILE is specified in flags, then mode is ignored (and can thus be specified as 0, or simply omitted). The mode argument must be sup? plied if O_CREAT or O_TMPFILE is specified in flags; if it is not supplied, some ar? bitrary bytes from the stack will be applied as the file mode.

The effective mode is modified by the process's umask in the usual way: in the ab?
sence of a default ACL, the mode of the created file is (mode & ~umask).
Note that 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 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 permission
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
According to POSIX, the effect when other bits are set in mode is unspecified. On
Linux, the following bits are also honored in mode:
S_ISUID 0004000 set-user-ID bit
S_ISGID 0002000 set-group-ID bit (see inode(7)).
S_ISVTX 0001000 sticky bit (see inode(7)).

O_DIRECT (since Linux 2.4.10)

Try to minimize cache effects of the I/O to and from this file. In general this will
degrade performance, but it is useful in special situations, such as when applica?
tions do their own caching. File I/O is done directly to/from user-space buffers.
The O_DIRECT flag on its own makes an effort to transfer data synchronously, but does
not give the guarantees of the O_SYNC flag that data and necessary metadata are
transferred. To guarantee synchronous I/O, O_SYNC must be used in addition to O_DI?
RECT. See NOTES below for further discussion.
A semantically similar (but deprecated) interface for block devices is described in
raw(8).

O_DIRECTORY

If path is not a directory, cause open() to fail. This flag was added in Linux 2.1.126, to avoid denial-of-service problems if opendir(3) is called on a FIFO or tape device.

O_DSYNC

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

By the time write(2) (and similar) return, the output data has been transferred to the underlying hardware, along with any file metadata that would be required to re? trieve that data (i.e., as though each write(2) was followed by a call to fdata? sync(2)). See VERSIONS.

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

When these two flags are specified, symbolic links are not followed: if path is a symbolic link, then open() fails regardless of where the symbolic link points.

In general, the behavior of O_EXCL is undefined if it is used without O_CREAT. There is one exception: on Linux 2.6 and later, O_EXCL can be used without O_CREAT if path refers to a block device. If the block device is in use by the system (e.g., mounted), open() fails with the error EBUSY.

On NFS, O_EXCL is supported only when using NFSv3 or later on kernel 2.6 or later. In NFS environments where O_EXCL support is not provided, programs that rely on it for performing locking tasks will contain a race condition. Portable programs that want to perform atomic file locking using a lockfile, and need to avoid reliance on NFS support for O_EXCL, can create a unique file on the same filesystem (e.g., incor? porating hostname and PID), and use link(2) to make a link to the lockfile. If link(2) 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 suc? cessful.

O_LARGEFILE

(LFS) Allow files whose sizes cannot be represented in an off_t (but can be repre? sented in an off64_t) to be opened. The _LARGEFILE64_SOURCE macro must be defined (before including any header files) in order to obtain this definition. Setting the _FILE_OFFSET_BITS feature test macro to 64 (rather than using O_LARGEFILE) is the preferred method of accessing large files on 32-bit systems (see fea?

ture_test_macros(7)).

O_NOATIME (since Linux 2.6.8)

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

This flag can be employed only if one of the following conditions is true:

? The effective UID of the process matches the owner UID of the file.

? The calling process has the CAP_FOWNER capability in its user namespace and the owner UID of the file has a mapping in the namespace.

This flag is intended for use by indexing or backup programs, where its use can sig? nificantly reduce the amount of disk activity. This flag may not be effective on all filesystems. One example is NFS, where the server maintains the access time.

O_NOCTTY

If path refers to a terminal device?see tty(4)?it will not become the process's con? trolling terminal even if the process does not have one.

O_NOFOLLOW

If the trailing component (i.e., basename) of path is a symbolic link, then the open fails, with the error ELOOP. Symbolic links in earlier components of the pathname will still be followed. (Note that the ELOOP error that can occur in this case is indistinguishable from the case where an open fails because there are too many sym? bolic links found while resolving components in the path prefix of the pathname.)

This flag is a FreeBSD extension, which was added in Linux 2.1.126, and has subse? quently been standardized in POSIX.1-2008.

See also O_PATH below.

O_NONBLOCK or O_NDELAY

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

Note that the setting of this flag has no effect on the operation of poll(2), se? lect(2), epoll(7), and similar, since those interfaces merely inform the caller about whether a file descriptor is "ready", meaning that an I/O operation performed on the file descriptor with the O_NONBLOCK flag clear would not block.

Note that this flag has no effect for regular files and block devices; that is, I/O operations will (briefly) block when device activity is required, regardless of

whether O_NONBLOCK is set. Since O_NONBLOCK semantics might eventually be imple? mented, applications should not depend upon blocking behavior when specifying this flag for regular files and block devices.

For the handling of FIFOs (named pipes), see also fifo(7). For a discussion of the effect of O_NONBLOCK in conjunction with mandatory file locks and with file leases, see fcntl(2).

O_PATH (since Linux 2.6.39)

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. The file itself is not opened, and other file operations (e.g., read(2), write(2), fchmod(2), fchown(2), fgetxattr(2), ioctl(2), mmap(2)) fail with the error EBADF.

The following operations can be performed on the resulting file descriptor:

? close(2).

? fchdir(2), if the file descriptor refers to a directory (since Linux 3.5).

? fstat(2) (since Linux 3.6).

? fstatfs(2) (since Linux 3.12).

? Duplicating the file descriptor (dup(2), fcntl(2) F_DUPFD, etc.).

? Getting and setting file descriptor flags (fcntl(2) F_GETFD and F_SETFD).

? Retrieving open file status flags using the fcntl(2) F_GETFL operation: the re? turned flags will include the bit O_PATH.

? Passing the file descriptor as the dirfd argument of openat() and the other "*at()" system calls. This includes linkat(2) with AT_EMPTY_PATH (or via procfs using AT_SYMLINK_FOLLOW) even if the file is not a directory.

? Passing the file descriptor to another process via a UNIX domain socket (see SCM_RIGHTS in unix(7)).

When O_PATH is specified in flags, flag bits other than O_CLOEXEC, O_DIRECTORY, and O_NOFOLLOW are ignored.

Opening a file or directory with the O_PATH flag requires no permissions on the ob? ject itself (but does require execute permission on the directories in the path pre? fix). Depending on the subsequent operation, a check for suitable file permissions may be performed (e.g., fchdir(2) requires execute permission on the directory re? ferred to by its file descriptor argument). By contrast, obtaining a reference to a

filesystem object by opening it with the O_RDONLY flag requires that the caller have read permission on the object, even when the subsequent operation (e.g., fchdir(2), fstat(2)) does not require read permission on the object.

If path is a symbolic link and the O_NOFOLLOW flag is also specified, then the call returns a file descriptor referring to the symbolic link. This file descriptor can be used as the dirfd argument in calls to fchownat(2), fstatat(2), linkat(2), and readlinkat(2) with an empty pathname to have the calls operate on the symbolic link.

If path refers to an automount point that has not yet been triggered, so no other filesystem is mounted on it, then the call returns a file descriptor referring to the automount directory without triggering a mount. fstatfs(2) can then be used to de? termine if it is, in fact, an untriggered automount point (.f_type == AUTOFS_SU? PER_MAGIC).

One use of O_PATH for regular files is to provide the equivalent of POSIX.1's O_EXEC functionality. This permits us to open a file for which we have execute permission but not read permission, and then execute that file, with steps something like the following:

```
char buf[PATH_MAX];
fd = open("some_prog", O_PATH);
snprintf(buf, PATH_MAX, "/proc/self/fd/%d", fd);
execl(buf, "some_prog", (char *) NULL);
```

An O_PATH file descriptor can also be passed as the argument of fexecve(3).

O_SYNC Write operations on the file will complete according to the requirements of synchro? nized I/O file integrity completion (by contrast with the synchronized I/O data in? tegrity completion provided by O_DSYNC.)

By the time write(2) (or similar) returns, the output data and associated file meta? data have been transferred to the underlying hardware (i.e., as though each write(2) was followed by a call to fsync(2)). See VERSIONS.

O_TMPFILE (since Linux 3.11)

Create an unnamed temporary regular file. The path argument specifies a directory; an unnamed inode will be created in that directory's filesystem. Anything written to the resulting file will be lost when the last file descriptor is closed, unless the file is given a name.

O_TMPFILE must be specified with one of O_RDWR or O_WRONLY and, optionally, O_EXCL.

If O_EXCL is not specified, then linkat(2) can be used to link the temporary file into the filesystem, making it permanent, using code like the following:

```
char path[PATH_MAX];
fd = open("/path/to/dir", O_TMPFILE | O_RDWR,
                S_IRUSR | S_IWUSR);
/* File I/O on 'fd'... */
linkat(fd, "", AT_FDCWD, "/path/for/file", AT_EMPTY_PATH);
/* If the caller doesn't have the CAP_DAC_READ_SEARCH
   capability (needed to use AT_EMPTY_PATH with linkat(2)),
   and there is a proc(5) filesystem mounted, then the
   linkat(2) call above can be replaced with:
snprintf(path, PATH_MAX,  "/proc/self/fd/%d", fd);
linkat(AT_FDCWD, path, AT_FDCWD, "/path/for/file",
                AT_SYMLINK_FOLLOW);
*/
```

In this case, the open() mode argument determines the file permission mode, as with O_CREAT.

Specifying O_EXCL in conjunction with O_TMPFILE prevents a temporary file from being linked into the filesystem in the above manner. (Note that the meaning of O_EXCL in this case is different from the meaning of O_EXCL otherwise.)

There are two main use cases for O_TMPFILE:

? Improved tmpfile(3) functionality: race-free creation of temporary files that (1) are automatically deleted when closed; (2) can never be reached via any pathname; (3) are not subject to symlink attacks; and (4) do not require the caller to de? vise unique names.

? Creating a file that is initially invisible, which is then populated with data and adjusted to have appropriate filesystem attributes (fchown(2), fchmod(2), fsetx? attr(2), etc.) before being atomically linked into the filesystem in a fully formed state (using linkat(2) as described above).

O_TMPFILE requires support by the underlying filesystem; only a subset of Linux filesystems provide that support. In the initial implementation, support was pro? vided in the ext2, ext3, ext4, UDF, Minix, and tmpfs filesystems. Support for other filesystems has subsequently been added as follows: XFS (Linux 3.15); Btrfs (Linux

3.16); F2FS (Linux 3.16); and ubifs (Linux 4.9)

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. Otherwise, the effect of O_TRUNC is unspecified.

creat()

A call to creat() is equivalent to calling open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC.

openat()

The openat() system call operates in exactly the same way as open(), except for the differences described here.

The dirfd argument is used in conjunction with the path argument as follows:

? If the pathname given in path is absolute, then dirfd is ignored.

? If the pathname given in path is relative and dirfd is the special value AT_FDCWD, then path is interpreted relative to the current working directory of the calling process (like open()).

? If the pathname given in path is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by open() for a relative pathname). In this case, dirfd must be a directory that was opened for reading (O_RDONLY) or using the O_PATH flag.

If the pathname given in path is relative, and dirfd is not a valid file descriptor, an error (EBADF) results. (Specifying an invalid file descriptor number in dirfd can be used as a means to ensure that path is absolute.)

openat2(2)

The openat2(2) system call is an extension of openat(), and provides a superset of the features of openat(). It is documented separately, in openat2(2).

RETURN VALUE

On success, open(), openat(), and creat() return the new file descriptor (a nonnegative integer). On error, -1 is returned and errno is set to indicate the error.

ERRORS

open(), openat(), and creat() 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 path, or the file did not exist yet and write access to the parent directory is not allowed. (See also path_resolution(7).)

EACCES Where O_CREAT is specified, the protected_fifos or protected_regular sysctl is en? abled, the file already exists and is a FIFO or regular file, the owner of the file is neither the current user nor the owner of the containing directory, and the con? taining directory is both world- or group-writable and sticky. For details, see the descriptions of /proc/sys/fs/protected_fifos and /proc/sys/fs/protected_regular in proc_sys_fs(5).

EBADF (openat()) path is relative but dirfd is neither AT_FDCWD nor a valid file descrip? tor.

EBUSY O_EXCL was specified in flags and path refers to a block device that is in use by the system (e.g., it is mounted).

EDQUOT Where O_CREAT is specified, the file does not exist, and the user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST path already exists and O_CREAT and O_EXCL were used.

EFAULT path points outside your accessible address space.

EFBIG See EOVERFLOW.

EINTR While blocked waiting to complete an open of a slow device (e.g., a FIFO; see fifo(7)), the call was interrupted by a signal handler; see signal(7).

EINVAL The filesystem does not support the O_DIRECT flag. See NOTES for more information.

EINVAL Invalid value in flags.

EINVAL O_TMPFILE was specified in flags, but neither O_WRONLY nor O_RDWR was specified.

EINVAL O_CREAT and O_DIRECTORY were both specified in flags , and the Linux kernel version is 6.4 or later. (Earlier kernels were inconsistent in this area, and POSIX does not specify the behavior.)

EINVAL O_CREAT was specified in flags and the final component ("basename") of the new file's path is invalid (e.g., it contains characters not permitted by the underlying filesystem).

EINVAL The final component ("basename") of path is invalid (e.g., it contains characters not permitted by the underlying filesystem).

EISDIR path refers to a directory and the access requested involved writing (that is, O_WRONLY or O_RDWR is set).

EISDIR path refers to an existing directory, O_TMPFILE and one of O_WRONLY or O_RDWR were specified in flags, but this kernel version does not provide the O_TMPFILE function? ality.

ELOOP Too many symbolic links were encountered in resolving path.

ELOOP path was a symbolic link, and flags specified O_NOFOLLOW but not O_PATH.

EMFILE The per-process limit on the number of open file descriptors has been reached (see the description of RLIMIT_NOFILE in getrlimit(2)).

ENAMETOOLONG

path was too long.

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

ENODEV path refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation ENXIO must be returned.)

ENOENT O_CREAT is not set and the named file does not exist.

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

ENOENT path refers to a nonexistent directory, O_TMPFILE and one of O_WRONLY or O_RDWR were specified in flags, but this kernel version does not provide the O_TMPFILE function? ality.

ENOMEM The named file is a FIFO, but memory for the FIFO buffer can't be allocated because the per-user hard limit on memory allocation for pipes has been reached and the caller is not privileged; see pipe(7).

ENOMEM Insufficient kernel memory was available.

ENOSPC path was to be created but the device containing path has no room for the new file.

ENOTDIR

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

ENOTDIR

(openat()) path is a relative pathname and dirfd is a file descriptor referring to a file other than a directory.

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

ENXIO The file is a device special file and no corresponding device exists.

ENXIO The file is a UNIX domain socket.

EOPNOTSUPP

The filesystem containing path does not support O_TMPFILE.

EOVERFLOW

path refers to a regular file that is too large to be opened. The usual scenario here is that an application compiled on a 32-bit platform without -D_FILE_OFF? SET_BITS=64 tried to open a file whose size exceeds (1<<31)-1 bytes; see also O_LARGEFILE above. This is the error specified by POSIX.1; before Linux 2.6.24, Linux gave the error EFBIG for this case.

EPERM The O_NOATIME flag was specified, but the effective user ID of the caller did not match the owner of the file and the caller was not privileged.

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

EROFS path refers to a file on a read-only filesystem and write access was requested.

ETXTBSY

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

ETXTBSY

path refers to a file that is currently in use as a swap file, and the O_TRUNC flag was specified.

ETXTBSY

path refers to a file that is currently being read by the kernel (e.g., for mod? ule/firmware loading), and write access was requested.

EWOULDBLOCK

The O_NONBLOCK flag was specified, and an incompatible lease was held on the file (see fcntl(2)).

VERSIONS

The (undefined) effect of O_RDONLY | O_TRUNC varies among implementations. On many systems the file is actually truncated.

Synchronized I/O

The POSIX.1-2008 "synchronized I/O" option specifies different variants of synchronized I/O, and specifies the open() flags O_SYNC, O_DSYNC, and O_RSYNC for controlling the behavior. Regardless of whether an implementation supports this option, it must at least support the use of O_SYNC for regular files.

Linux implements O_SYNC and O_DSYNC, but not O_RSYNC. Somewhat incorrectly, glibc defines O_RSYNC to have the same value as O_SYNC. (O_RSYNC is defined in the Linux header file

<asm/fcntl.h> on HP PA-RISC, but it is not used.)

O_SYNC provides synchronized I/O file integrity completion, meaning write operations will flush data and all associated metadata to the underlying hardware. O_DSYNC provides synchronized I/O data integrity completion, meaning write operations will flush data to the underlying hardware, but will only flush metadata updates that are required to allow a subsequent read operation to complete successfully. Data integrity completion can reduce the number of disk operations that are required for applications that don't need the guarantees of file integrity completion.

To understand the difference between the two types of completion, consider two pieces of file metadata: the file last modification timestamp (st_mtime) and the file length. All write operations will update the last file modification timestamp, but only writes that add data to the end of the file will change the file length. The last modification timestamp is not needed to ensure that a read completes successfully, but the file length is. Thus, O_DSYNC would only guarantee to flush updates to the file length metadata (whereas O_SYNC would also always flush the last modification timestamp metadata).

Before Linux 2.6.33, Linux implemented only the O_SYNC flag for open(). However, when that flag was specified, most filesystems actually provided the equivalent of synchronized I/O data integrity completion (i.e., O_SYNC was actually implemented as the equivalent of O_DSYNC).

Since Linux 2.6.33, proper O_SYNC support is provided. However, to ensure backward binary compatibility, O_DSYNC was defined with the same value as the historical O_SYNC, and O_SYNC was defined as a new (two-bit) flag value that includes the O_DSYNC flag value. This ensures that applications compiled against new headers get at least O_DSYNC semantics before Linux 2.6.33.

C library/kernel differences

Since glibc 2.26, the glibc wrapper function for open() employs the openat() system call, rather than the kernel's open() system call. For certain architectures, this is also true before glibc 2.26.

POSIX

POSIX.1-2024 specifies O_CLOFORK, but Linux doesn't support it.

STANDARDS

open()

creat()

openat()

POSIX.1-2024.

openat2(2)

Linux.

O_DIRECT

O_NOATIME

O_PATH

O_TMPFILE

Linux.

HISTORY

open()

creat()

SVr4, 4.3BSD, POSIX.1-2001.

openat()

POSIX.1-2008. Linux 2.6.16, glibc 2.4.

O_CLOEXEC

O_DIRECTORY

O_NOFOLLOW

POSIX.1-2008.

NOTES

Under Linux, the O_NONBLOCK flag is sometimes used in cases where one wants to open but does not necessarily have the intention to read or write. For example, this may be used to open a device in order to get a file descriptor for use with ioctl(2).

Note that open() can open device special files, but creat() cannot create them; use mknod(2) instead.

If the file is newly created, its st_atime, st_ctime, st_mtime fields (respectively, time of last access, time of last status change, and time of last modification; see stat(2)) are set to the current time, and so are the st_ctime and st_mtime fields of the parent directory.

Otherwise, if the file is modified because of the O_TRUNC flag, its st_ctime and st_mtime fields are set to the current time.

The files in the /proc/pid/fd directory show the open file descriptors of the process with the PID pid. The files in the /proc/pid/fdinfo directory show even more information about these file descriptors. See proc(5) for further details of both of these directories.

The Linux header file <asm/fcntl.h> doesn't define O_ASYNC; the (BSD-derived) FASYNC synonym is defined instead.

Open file descriptions

The term open file description is the one used by POSIX to refer to the entries in the sys? tem-wide table of open files. In other contexts, this object is variously also called an "open file object", a "file handle", an "open file table entry", or?in kernel-developer par? lance?a struct file.

When a file descriptor is duplicated (using dup(2) or similar), the duplicate refers to the same open file description as the original file descriptor, and the two file descriptors consequently share the file offset and file status flags. Such sharing can also occur be? tween processes: a child process created via fork(2) inherits duplicates of its parent's file descriptors, and those duplicates refer to the same open file descriptions.

Each open() of a file creates a new open file description; thus, there may be multiple open file descriptions corresponding to a file inode.

On Linux, one can use the kcmp(2) KCMP_FILE operation to test whether two file descriptors (in the same process or in two different processes) refer to the same open file description.

NFS

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

On NFS filesystems with UID mapping enabled, open() may return a file descriptor but, for example, 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.

FIFOs

Opening the read or write end of a FIFO blocks until the other end is also opened (by an? other process or thread). See fifo(7) for further details.

File access mode

Unlike the other values that can be specified in flags, the access mode values O_RDONLY, O_WRONLY, and O_RDWR do not specify individual bits. Rather, they define the low order two bits of flags, and are defined respectively as 0, 1, and 2. In other words, the combination O_RDONLY | O_WRONLY is a logical error, and certainly does not have the same meaning as O_RDWR.

Linux reserves the special, nonstandard access mode 3 (binary 11) in flags to mean: check

for read and write permission on the file and return a file descriptor that can't be used for reading or writing. This nonstandard access mode is used by some Linux drivers to re? turn a file descriptor that is to be used only for device-specific ioctl(2) operations.

Rationale for openat() and other directory file descriptor APIs

openat() and the other system calls and library functions that take a directory file de? scriptor argument (i.e., execveat(2), faccessat(2), fanotify_mark(2), fchmodat(2), fchow? nat(2), fspick(2), fstatat(2), futimesat(2), linkat(2), mkdirat(2), mknodat(2), mount_se? tattr(2), move_mount(2), name_to_handle_at(2), open_tree(2), openat2(2), readlinkat(2), re? nameat(2), renameat2(2), statx(2), symlinkat(2), unlinkat(2), utimensat(2), mkfifoat(3), and scandirat(3)) address two problems with the older interfaces that preceded them. Here, the explanation is in terms of the openat() call, but the rationale is analogous for the other interfaces.

First, openat() allows an application to avoid race conditions that could occur when using open() to open files in directories other than the current working directory. These race conditions result from the fact that some component of the directory prefix given to open() could be changed in parallel with the call to open(). Suppose, for example, that we wish to create the file dir1/dir2/xxx.dep if the file dir1/dir2/xxx exists. The problem is that be? tween the existence check and the file-creation step, dir1 or dir2 (which might be symbolic links) could be modified to point to a different location. Such races can be avoided by opening a file descriptor for the target directory, and then specifying that file descriptor as the dirfd argument of (say) fstatat(2) and openat(). The use of the dirfd file descrip? tor also has other benefits:

? the file descriptor is a stable reference to the directory, even if the directory is re? named; and

? the open file descriptor prevents the underlying filesystem from being dismounted, just as when a process has a current working directory on a filesystem.

Second, openat() allows the implementation of a per-thread "current working directory", via file descriptor(s) maintained by the application. (This functionality can also be obtained by tricks based on the use of /proc/self/fd/dirfd, but less efficiently.)

The dirfd argument for these APIs can be obtained by using open() or openat() to open a di? rectory (with either the O_RDONLY or the O_PATH flag). Alternatively, such a file descrip? tor can be obtained by applying dirfd(3) to a directory stream created using opendir(3).

When these APIs are given a dirfd argument of AT_FDCWD or the specified pathname is ab?

solute, then they handle their pathname argument in the same way as the corresponding con? ventional APIs. However, in this case, several of the APIs have a flags argument that pro? vides access to functionality that is not available with the corresponding conventional APIs.

O_DIRECT

The O_DIRECT flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. In Linux alignment restrictions vary by filesystem and kernel version and might be absent entirely. The handling of misaligned O_DIRECT I/Os also varies; they can either fail with EINVAL or fall back to buffered I/O.

Since Linux 6.1, O_DIRECT support and alignment restrictions for a file can be queried using statx(2), using the STATX_DIOALIGN flag. Support for STATX_DIOALIGN varies by filesystem; see statx(2).

Some filesystems provide their own interfaces for querying O_DIRECT alignment restrictions, for example the XFS_IOC_DIOINFO operation in xfsctl(3). STATX_DIOALIGN should be used in? stead when it is available.

If none of the above is available, then direct I/O support and alignment restrictions can only be assumed from known characteristics of the filesystem, the individual file, the un? derlying storage device(s), and the kernel version. In Linux 2.4, most filesystems based on block devices require that the file offset and the length and memory address of all I/O seg? ments be multiples of the filesystem block size (typically 4096 bytes). In Linux 2.6.0, this was relaxed to the logical block size of the block device (typically 512 bytes). A block device's logical block size can be determined using the ioctl(2) BLKSSZGET operation or from the shell using the command:

```
blockdev --getss
```

O_DIRECT I/Os should never be run concurrently with the fork(2) system call, if the memory buffer is a private mapping (i.e., any mapping created with the mmap(2) MAP_PRIVATE flag; this includes memory allocated on the heap and statically allocated buffers). Any such I/Os, whether submitted via an asynchronous I/O interface or from another thread in the process, should be completed before fork(2) is called. Failure to do so can result in data corruption and undefined behavior in parent and child processes. This restriction does not apply when the memory buffer for the O_DIRECT I/Os was created using shmat(2) or mmap(2) with the MAP_SHARED flag. Nor does this restriction apply when the memory buffer has been advised as MADV_DONTFORK with madvise(2), ensuring that it will not be available to the

child after fork(2).

The O_DIRECT flag was introduced in SGI IRIX, where it has alignment restrictions similar to those of Linux 2.4. IRIX has also a fcntl(2) call to query appropriate alignments, and sizes. FreeBSD 4.x introduced a flag of the same name, but without alignment restrictions. O_DIRECT support was added in Linux 2.4.10. Older Linux kernels simply ignore this flag. Some filesystems may not implement the flag, in which case open() fails with the error EIN? VAL if it is used.

Applications should avoid mixing O_DIRECT and normal I/O to the same file, and especially to overlapping byte regions in the same file. Even when the filesystem correctly handles the coherency issues in this situation, overall I/O throughput is likely to be slower than using either mode alone. Likewise, applications should avoid mixing mmap(2) of files with direct I/O to the same files.

The behavior of O_DIRECT with NFS will differ from local filesystems. Older kernels, or kernels configured in certain ways, may not support this combination. The NFS protocol does not support passing the flag to the server, so O_DIRECT I/O will bypass the page cache only on the client; the server may still cache the I/O. The client asks the server to make the I/O synchronous to preserve the synchronous semantics of O_DIRECT. Some servers will per? form poorly under these circumstances, especially if the I/O size is small. Some servers may also be configured to lie to clients about the I/O having reached stable storage; this will avoid the performance penalty at some risk to data integrity in the event of server power failure. The Linux NFS client places no alignment restrictions on O_DIRECT I/O.

In summary, O_DIRECT is a potentially powerful tool that should be used with caution. It is recommended that applications treat use of O_DIRECT as a performance option which is dis? abled by default.

BUGS

Currently, it is not possible to enable signal-driven I/O by specifying O_ASYNC when calling open(); use fcntl(2) to enable this flag.

One must check for two different error codes, EISDIR and ENOENT, when trying to determine whether the kernel supports O_TMPFILE functionality.

SEE ALSO

chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), open_by_handle_at(2), openat2(2), read(2), socket(2), stat(2), umask(2), un? link(2), write(2), fopen(3), acl(5), fifo(7), inode(7), path_resolution(7), symlink(7)