NAME

open, openat, creat − 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);

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);
```

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

Given a *pathname* for a file, **open**() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered 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 *pathname* 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-*or*'d in *flags*. 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. 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 positioned 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 appends 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 permits 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 suffice 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 the file does not exist, it will be created.

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; otherwise, 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 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
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 applications 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_DIRECT**. See NOTES below for further discussion.

A semantically similar (but deprecated) interface for block devices is described in **raw**(8).

O_DIRECTORY
If *pathname* is not a directory, cause the open to fail. This flag was added in kernel version 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 synchronized 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 retrieve that data (i.e., as though each **write**(2) was followed by a call to **fdatasync**(2)). *See NOTES below*.

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**() will fail.

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

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 *pathname* 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., incorporating 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 successful.

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. 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 **feature_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 significantly 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 *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**. 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 symbolic links found while resolving components in the prefix part of the pathname.)

This flag is a FreeBSD extension, which was added to Linux in version 2.1.126, and has subsequently 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 operations on the file descriptor which is returned will cause the calling process to wait.

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 implemented, 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),

mmap(2)) fail with the error **EBADF**.

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

* **close**(2); **fchdir**(2) (since Linux 3.5); **fstat**(2) (since Linux 3.6).
* 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 returned 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.

If *pathname* 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.

O_SYNC

Write operations on the file will complete according to the requirements of synchronized I/O *file* integrity completion (by contrast with the synchronized I/O *data* integrity completion provided by **O_DSYNC**.)

By the time **write**(2) (and similar) return, the output data and associated file metadata have been transferred to the underlying hardware (i.e., as though each **write**(2) was followed by a call to **fsync**(2)). *See NOTES below*.

O_TMPFILE (since Linux 3.11)

Create an unnamed temporary file. The *pathname* 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'... */

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 devise 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), **fsetxattr**(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 provided in the ext2, ext3, ext4, UDF, Minix, and shmem 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.

If the pathname given in *pathname* 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).

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

If *pathname* is absolute, then *dirfd* is ignored.

RETURN VALUE

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

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 *pathname*, or the file did not exist yet and write access to the parent directory is not allowed. (See also **path_resolution**(7).)

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
pathname already exists and **O_CREAT** and **O_EXCL** were used.

EFAULT
pathname 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.

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

EISDIR
pathname 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** functionality.

ELOOP
Too many symbolic links were encountered in resolving *pathname*.

ELOOP
pathname 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
pathname was too long.

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

ENODEV
pathname 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. Or, a directory component in *pathname* does not exist or is a dangling symbolic link.

ENOENT
pathname 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** functionality.

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
pathname was to be created but the device containing *pathname* has no room for the new file.

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

EOPNOTSUPP
The filesystem containing *pathname* does not support **O_TMPFILE**.

EOVERFLOW
pathname 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_OFFSET_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; in kernels before 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
pathname refers to a file on a read-only filesystem and write access was requested.

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

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

The following additional errors can occur for **openat**():

EBADF
dirfd is not a valid file descriptor.

ENOTDIR
pathname is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

VERSIONS

openat() was added to Linux in kernel 2.6.16; library support was added to glibc in version 2.4.

CONFORMING TO

open(), **creat**() SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.

openat(): POSIX.1-2008.

The **O_DIRECT**, **O_NOATIME**, **O_PATH**, and **O_TMPFILE** flags are Linux-specific. One must define **_GNU_SOURCE** to obtain their definitions.

The **O_CLOEXEC**, **O_DIRECTORY**, and **O_NOFOLLOW** flags are not specified in POSIX.1-2001, but are specified in POSIX.1-2008. Since glibc 2.12, one can obtain their definitions by defining either **_POSIX_C_SOURCE** with a value greater than or equal to 200809L or **_XOPEN_SOURCE** with a value greater than or equal to 700. In glibc 2.11 and earlier, one obtains the definitions by defining **_GNU_SOURCE**.

As noted in **feature_test_macros**(7), feature test macros such as **_POSIX_C_SOURCE**, **_XOPEN_SOURCE**, and **_GNU_SOURCE** must be defined before including *any* header files.

NOTES

Under Linux, the **O_NONBLOCK** flag indicates that one wants to open but does not necessarily have the intention to read or write. This is typically used to open devices in order to get a file descriptor for use with **ioctl**(2).

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

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 files descriptors. See **proc**(5) for further details of both of these directories.

Open file descriptions

The term open file description is the one used by POSIX to refer to the entries in the system-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 parlance—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 between 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.

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_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 on pre-2.6.33 kernels.

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 another 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 return 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 descriptor argument (i.e., **execveat**(2), **faccessat**(2), **fanotify_mark**(2), **fchmodat**(2), **fchownat**(2), **fstatat**(2), **futimesat**(2), **linkat**(2), **mkdirat**(2), **mknodat**(2), **name_to_handle_at**(2), **readlinkat**(2), **renameat**(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 *path/to/xxx.dep* if the file *path/to/xxx* exists. The problem is that between the existence check and the file creation step, *path* or *to* (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 descriptor also has other benefits:

* the file descriptor is a stable reference to the directory, even if the directory is renamed; 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.)

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. However there is currently no filesystem−independent interface for an application to discover these restrictions for a given file or filesystem. Some filesystems provide their own interfaces for doing so, for example the **XFS_IOC_DIOINFO** operation in **xfsctl**(3).

Under Linux 2.4, transfer sizes, and the alignment of the user buffer and the file offset must all be multiples of the logical block size of the filesystem. Since Linux 2.6.0, alignment to the logical block size of the underlying storage (typically 512 bytes) suffices. The 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 under Linux in kernel version 2.4.10. Older Linux kernels simply ignore this flag. Some filesystems may not implement the flag and **open**() will fail with **EINVAL** 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 perform 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 disabled by default.

> "The thing that has always disturbed me about O_DIRECT is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances."—Linus

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.

When both **O_CREAT** and **O_DIRECTORY** are specified in *flags* and the file specified by *pathname* does not exist, **open**() will create a regular file (i.e., **O_DIRECTORY** is ignored).

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), **read**(2), **socket**(2), **stat**(2), **umask**(2), **unlink**(2), **write**(2), **fopen**(3), **acl**(5), **fifo**(7), **inode**(7), **path_resolution**(7), **symlink**(7)

COLOPHON

This page is part of release 4.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.