**How does the open system call work**

**Introduction**

This is the fifth part of the chapter that describes [system calls](https://en.wikipedia.org/wiki/System_call) mechanism in the Linux kernel. Previous parts of this chapter described this mechanism in general. Now I will try to describe implementation of different system calls in the Linux kernel. Previous parts from this chapter and parts from other chapters of the books describe mostly deep parts of the Linux kernel that are faintly visible or fully invisible from the userspace. But the Linux kernel code is not only about itself. The vast of the Linux kernel code provides ability to our code. Due to the linux kernel our programs can read/write from/to files and don't know anything about sectors, tracks and other parts of a disk structures, we can send data over network and don't build encapsulated network packets by hand and etc.

I don't know how about you, but it is interesting to me not only how an operating system works, but how do my software interacts with it. As you may know, our programs interacts with the kernel through the special mechanism which is called [system call](https://en.wikipedia.org/wiki/System_call). So, I've decided to write series of parts which will describe implementation and behavior of system calls which we are using every day like read, write, open, close, dup and etc.

I have decided to start from the description of the [open](http://man7.org/linux/man-pages/man2/open.2.html) system call. if you have written at least one C program, you should know that before we are able to read/write or execute other manipulations with a file we need to open it with the open function:

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

int main(int argc, char \*argv) {

int fd = open("test", O\_RDONLY);

if fd < 0 {

perror("Opening of the file is failed\n");

}

else {

printf("file sucessfully opened\n");

}

close(fd);

return 0;

}

In this case, the open is the function from standard library, but not system call. The standard library will call related system call for us. The open call will return a [file descriptor](https://en.wikipedia.org/wiki/File_descriptor) which is just a unique number within our process which is associated with the opened file. Now as we opened a file and got file descriptor as result of open call, we may start to interact with this file. We can write into, read from it and etc. List of opened file by a process is available via [proc](https://en.wikipedia.org/wiki/Procfs) filesystem:

$ sudo ls /proc/1/fd/

**Definition of the open system call**

If you have read the [fourth part](https://github.com/0xAX/linux-insides/blob/master/SysCall/linux-syscall-4.md) of the [linux-insides](https://0xax.gitbooks.io/linux-insides/content/index.html" \t "_blank) book, you should know that system calls are defined with the help of SYSCALL\_DEFINE macro. So, the open system call is not exception.

Definition of the open system call is located in the [fs/open.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file and looks pretty small for the first view:

SYSCALL\_DEFINE3(open, const char \_\_user \*, filename, int, flags, umode\_t, mode)

{

if (force\_o\_largefile())

flags |= O\_LARGEFILE;

return do\_sys\_open(AT\_FDCWD, filename, flags, mode);

}

As you may guess, the do\_sys\_open function from the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file does the main job. But before this function will be called, let's consider the if clause from which the implementation of the open system call starts:

if (force\_o\_largefile())

flags |= O\_LARGEFILE;

Here we apply the O\_LARGEFILE flag to the flags which were passed to open system call in a case when the force\_o\_largefile() will return true. What is O\_LARGEFILE? We may read this in the [man page](http://man7.org/linux/man-pages/man2/open.2.html) for the open(2) system call:

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.

As we may read in the [GNU C Library Reference Manual](https://www.gnu.org/software/libc/manual/html_mono/libc.html#File-Position-Primitive):

off\_t

This is a signed integer type used to represent file sizes. In the GNU C Library, this type is no narrower than int. If the source is compiled with \_FILE\_OFFSET\_BITS == 64 this type is transparently replaced by off64\_t.

and

off64\_t

This type is used similar to off\_t. The difference is that even on 32 bit machines, where the off\_t type would have 32 bits, off64\_t has 64 bits and so is able to address files up to 2^63 bytes in length. When compiling with \_FILE\_OFFSET\_BITS == 64 this type is available under the name off\_t.

So it is not hard to guess that the off\_t, off64\_t and O\_LARGEFILE are about a file size. In the case of the Linux kernel, the O\_LARGEFILE is used to disallow opening large files on 32bit systems if the caller didn't specify O\_LARGEFILE flag during opening of a file. On 64bit systems we force on this flag in open system call. And the force\_o\_largefile macro from the [include/linux/fcntl.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fcntl.h#L7) linux kernel header file confirms this:

#ifndef force\_o\_largefile

#define force\_o\_largefile() (BITS\_PER\_LONG != 32)

#endif

This macro may be architecture-specific as for example for [IA-64](https://en.wikipedia.org/wiki/IA-64) architecture, but in our case the [x86\_64](https://en.wikipedia.org/wiki/X86-64) does not provide definition of the force\_o\_largefile and it will be used from [include/linux/fcntl.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fcntl.h#L7).

So, as we may see the force\_o\_largefile is just a macro which expands to the true value in our case of [x86\_64](https://en.wikipedia.org/wiki/X86-64) architecture. As we are considering 64-bit architecture, the force\_o\_largefile will be expanded to true and the O\_LARGEFILE flag will be added to the set of flags which were passed to the open system call.

Now as we considered meaning of the O\_LARGEFILE flag and force\_o\_largefile macro, we can proceed to the consideration of the implementation of the do\_sys\_open function. As I wrote above, this function is defined in the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file and looks:

long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, umode\_t mode)

{

struct open\_flags op;

int fd = build\_open\_flags(flags, mode, &op);

struct filename \*tmp;

if (fd)

return fd;

tmp = getname(filename);

if (IS\_ERR(tmp))

return PTR\_ERR(tmp);

fd = get\_unused\_fd\_flags(flags);

if (fd >= 0) {

struct file \*f = do\_filp\_open(dfd, tmp, &op);

if (IS\_ERR(f)) {

put\_unused\_fd(fd);

fd = PTR\_ERR(f);

} else {

fsnotify\_open(f);

fd\_install(fd, f);

}

}

putname(tmp);

return fd;

}

Let's try to understand how the do\_sys\_open works step by step.

**open(2) flags**

As you know the open system call takes set of flags as second argument that control opening a file and mode as third argument that specifies permission the permissions of a file if it is created. The do\_sys\_open function starts from the call of the build\_open\_flags function which does some checks that set of the given flags is valid and handles different conditions of flags and mode.

Let's look at the implementation of the build\_open\_flags. This function is defined in the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) kernel file and takes three arguments:

* flags - flags that control opening of a file;
* mode - permissions for newly created file;

The last argument - op is represented with the open\_flags structure:

struct open\_flags {

int open\_flag;

umode\_t mode;

int acc\_mode;

int intent;

int lookup\_flags;

};

which is defined in the [fs/internal.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/internal.h#L99) header file and as we may see it holds information about flags and access mode for internal kernel purposes. As you already may guess the main goal of the build\_open\_flags function is to fill an instance of this structure.

Implementation of the build\_open\_flags function starts from the definition of local variables and one of them is:

int acc\_mode = ACC\_MODE(flags);

This local variable represents access mode and its initial value will be equal to the value of expanded ACC\_MODE macro. This macro is defined in the [include/linux/fs.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fs.h) and looks pretty interesting:

#define ACC\_MODE(x) ("\004\002\006\006"[(x)&O\_ACCMODE])

#define O\_ACCMODE 00000003

The "\004\002\006\006" is an array of four chars:

"\004\002\006\006" == {'\004', '\002', '\006', '\006'}

So, the ACC\_MODE macro just expands to the accession to this array by [(x) & O\_ACCMODE] index. As we just saw, the O\_ACCMODE is 00000003. By applying x & O\_ACCMODE we will take the two least significant bits which are represents read, write or read/write access modes:

#define O\_RDONLY 00000000

#define O\_WRONLY 00000001

#define O\_RDWR 00000002

After getting value from the array by the calculated index, the ACC\_MODE will be expanded to access mode mask of a file which will hold MAY\_WRITE, MAY\_READ and other information.

We may see following condition after we have calculated initial access mode:

if (flags & (O\_CREAT | \_\_O\_TMPFILE))

op->mode = (mode & S\_IALLUGO) | S\_IFREG;

else

op->mode = 0;

Here we reset permissions in open\_flags instance if a opened file wasn't temporary and wasn't open for creation. This is because:

if neither O\_CREAT nor O\_TMPFILE is specified, then mode is ignored.

In other case if O\_CREAT or O\_TMPFILE were passed we canonicalize it to a regular file because a directory should be created with the [opendir](http://man7.org/linux/man-pages/man3/opendir.3.html" \t "_blank) system call.

At the next step we check that a file is not tried to be opened via [fanotify](http://man7.org/linux/man-pages/man7/fanotify.7.html" \t "_blank) and without the O\_CLOEXEC flag:

flags &= ~FMODE\_NONOTIFY & ~O\_CLOEXEC;

We do this to not leak a [file descriptor](https://en.wikipedia.org/wiki/File_descriptor). By default, the new file descriptor is set to remain open across an execve system call, but the open system call supports O\_CLOEXEC flag that can be used to change this default behaviour. So we do this to prevent leaking of a file descriptor when one thread opens a file to set O\_CLOEXEC flag and in the same time the second process does a [fork](https://en.wikipedia.org/wiki/Fork_/(system_call/)) + [execve](https://en.wikipedia.org/wiki/Exec_/(system_call/" \t "_blank)) and as you may remember that child will have copies of the parent's set of open file descriptors.

At the next step we check that if our flags contains O\_SYNC flag, we apply O\_DSYNC flag too:

if (flags & \_\_O\_SYNC)

flags |= O\_DSYNC;

The O\_SYNC flag guarantees that the any write call will not return before all data has been transferred to the disk. The O\_DSYNC is like O\_SYNC except that there is no requirement to wait for any metadata (like atime, mtime and etc.) changes will be written. We apply O\_DSYNC in a case of \_\_O\_SYNC because it is implemented as \_\_O\_SYNC|O\_DSYNC in the Linux kernel.

After this we must be sure that if a user wants to create temporary file, the flags should contain O\_TMPFILE\_MASK or in other words it should contain or O\_CREAT or O\_TMPFILE or both and also it should be writeable:

if (flags & \_\_O\_TMPFILE) {

if ((flags & O\_TMPFILE\_MASK) != O\_TMPFILE)

return -EINVAL;

if (!(acc\_mode & MAY\_WRITE))

return -EINVAL;

} else if (flags & O\_PATH) {

flags &= O\_DIRECTORY | O\_NOFOLLOW | O\_PATH;

acc\_mode = 0;

}

as it is written in in the manual page:

O\_TMPFILE must be specified with one of O\_RDWR or O\_WRONLY

If we didn't pass O\_TMPFILE for creation of a temporary file, we check the O\_PATH flag at the next condition. The O\_PATH flag allows us to obtain a file descriptor that may be used for two following purposes:

* to indicate a location in the filesystem tree;
* to perform operations that act purely at the file descriptor level.

So, in this case the file itself is not opened, but operations like dup, fcntl and other can be used. So, if all file content related operations like read, write and other are not permitted, only O\_DIRECTORY | O\_NOFOLLOW | O\_PATH flags can be used. We have finished with flags for this moment in the build\_open\_flags for this moment and we may fill our open\_flags->open\_flag with them:

op->open\_flag = flags;

Now we have filled open\_flag field which represents flags that will control opening of a file and mode that will represent umask of a new file if we open file for creation. There are still to fill last flags in the our open\_flags structure. The next is op->acc\_mode which represents access mode to a opened file. We already filled the acc\_mode local variable with the initial value at the beginning of the build\_open\_flags and now we check last two flags related to access mode:

if (flags & O\_TRUNC)

acc\_mode |= MAY\_WRITE;

if (flags & O\_APPEND)

acc\_mode |= MAY\_APPEND;

op->acc\_mode = acc\_mode;

These flags are - O\_TRUNC that will truncate an opened file to length 0 if it existed before we open it and the O\_APPEND flag allows to open a file in append mode. So the opened file will be appended during write but not overwritten.

The next field of the open\_flags structure is - intent. It allows us to know about our intention or in other words what do we really want to do with file, open it, create, rename it or something else. So we set it to zero if our flags contains the O\_PATH flag as we can't do anything related to a file content with this flag:

op->intent = flags & O\_PATH ? 0 : LOOKUP\_OPEN;

or just to LOOKUP\_OPEN intention. Additionally we set LOOKUP\_CREATE intention if we want to create new file and to be sure that a file didn't exist before with O\_EXCL flag:

if (flags & O\_CREAT) {

op->intent |= LOOKUP\_CREATE;

if (flags & O\_EXCL)

op->intent |= LOOKUP\_EXCL;

}

The last flag of the open\_flags structure is the lookup\_flags:

if (flags & O\_DIRECTORY)

lookup\_flags |= LOOKUP\_DIRECTORY;

if (!(flags & O\_NOFOLLOW))

lookup\_flags |= LOOKUP\_FOLLOW;

op->lookup\_flags = lookup\_flags;

return 0;

We fill it with LOOKUP\_DIRECTORY if we want to open a directory and LOOKUP\_FOLLOW if we don't want to follow (open) [symlink](https://en.wikipedia.org/wiki/Symbolic_link" \t "_blank). That's all. It is the end of the build\_open\_flags function. The open\_flags structure is filled with modes and flags for a file opening and we can return back to the do\_sys\_open.

**Actual opening of a file**

At the next step after build\_open\_flags function is finished and we have formed flags and modes for our file we should get the filename structure with the help of the getname function by name of a file which was passed to the open system call:

tmp = getname(filename);

if (IS\_ERR(tmp))

return PTR\_ERR(tmp);

The getname function is defined in the [fs/namei.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c) source code file and looks:

struct filename \*

getname(const char \_\_user \* filename)

{

return getname\_flags(filename, 0, NULL);

}

So, it just calls the getname\_flags function and returns its result. The main goal of the getname\_flags function is to copy a file path given from userland to kernel space. The filename structure is defined in the [include/linux/fs.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fs.h) linux kernel header file and contains following fields:

* name - pointer to a file path in kernel space;
* uptr - original pointer from userland;
* aname - filename from [audit](https://linux.die.net/man/8/auditd) context;
* refcnt - reference counter;
* iname - a filename in a case when it will be less than PATH\_MAX.

As I already wrote above, the main goal of the getname\_flags function is to copy name of a file which was passed to the open system call from user space to kernel space with the strncpy\_from\_user function. The next step after a filename will be copied to kernel space is getting of new non-busy file descriptor:

fd = get\_unused\_fd\_flags(flags);

The get\_unused\_fd\_flags function takes table of open files of the current process, minimum (0) and maximum (RLIMIT\_NOFILE) possible number of a file descriptor in the system and flags that we have passed to the open system call and allocates file descriptor and mark it busy in the file descriptor table of the current process. The get\_unused\_fd\_flags function sets or clears the O\_CLOEXEC flag depends on its state in the passed flags.

The last and main step in the do\_sys\_open is the do\_filp\_open function:

struct file \*f = do\_filp\_open(dfd, tmp, &op);

if (IS\_ERR(f)) {

put\_unused\_fd(fd);

fd = PTR\_ERR(f);

} else {

fsnotify\_open(f);

fd\_install(fd, f);

}

The main goal of this function is to resolve given path name into file structure which represents an opened file of a process. If something going wrong and execution of the do\_filp\_open function will be failed, we should free new file descriptor with the put\_unused\_fd or in other way the file structure returned by the do\_filp\_open will be stored in the file descriptor table of the current process.

Now let's take a short look at the implementation of the do\_filp\_open function. This function is defined in the [fs/namei.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c) linux kernel source code file and starts from initialization of the nameidata structure. This structure will provide a link to a file [inode](https://en.wikipedia.org/wiki/Inode" \t "_blank). Actually this is one of the main point of the do\_filp\_open function to acquire an inode by the filename given to open system call. After the nameidata structure will be initialized, the path\_openat function will be called:

filp = path\_openat(&nd, op, flags | LOOKUP\_RCU);

if (unlikely(filp == ERR\_PTR(-ECHILD)))

filp = path\_openat(&nd, op, flags);

if (unlikely(filp == ERR\_PTR(-ESTALE)))

filp = path\_openat(&nd, op, flags | LOOKUP\_REVAL);

Note that it is called three times. Actually, the Linux kernel will open the file in [RCU](https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt) mode. This is the most efficient way to open a file. If this try will be failed, the kernel enters the normal mode. The third call is relatively rare, only in the [nfs](https://en.wikipedia.org/wiki/Network_File_System" \t "_blank) file system is likely to be used. The path\_openat function executes path lookup or in other words it tries to find a dentry (what the Linux kernel uses to keep track of the hierarchy of files in directories) corresponding to a path.

The path\_openat function starts from the call of the get\_empty\_flip() function that allocates a new file structure with some additional checks like do we exceed amount of opened files in the system or not and etc. After we have got allocated new file structure we call the do\_tmpfile or do\_o\_path functions in a case if we have passed O\_TMPFILE | O\_CREATE or O\_PATH flags during call of the open system call. These both cases are quite specific, so let's consider quite usual case when we want to open already existed file and want to read/write from/to it.

In this case the path\_init function will be called. This function performs some preporatory work before actual path lookup. This includes search of start position of path traversal and its metadata like inode of the path, dentry inode and etc. This can be root directory - / or current directory as in our case, because we use AT\_CWD as starting point (see call of the do\_sys\_open at the beginning of the post).

The next step after the path\_init is the [loop](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c#L3457) which executes the link\_path\_walk and do\_last. The first function executes name resolution or in other words this function starts process of walking along a given path. It handles everything step by step except the last component of a file path. This handling includes checking of a permissions and getting a file component. As a file component is gotten, it is passed to walk\_component that updates current directory entry from the dcache or asks underlying filesystem. This repeats before all path's components will not be handled in such way. After the link\_path\_walk will be executed, the do\_last function will populate a file structure based on the result of the link\_path\_walk. As we reached last component of the given file path the vfs\_open function from the do\_last will be called.

This function is defined in the [fs/open.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) linux kernel source code file and the main goal of this function is to call an open operation of underlying filesystem.

That's all for now. We didn't consider **full** implementation of the open system call. We skip some parts like handling case when we want to open a file from other filesystem with different mount point, resolving symlinks and etc., but it should be not so hard to follow this stuff. This stuff does not included in **generic** implementation of open system call and depends on underlying filesystem. If you are interested in, you may lookup the file\_operations.open callback function for a certain [filesystem](https://github.com/torvalds/linux/tree/master/fs).

**Conclusion**

This is the end of the fifth part of the implementation of different system calls in the Linux kernel. If you have questions or suggestions, ping me on twitter [0xAX](https://twitter.com/0xAX), drop me an [email](https://0xax.gitbooks.io/linux-insides/content/SysCall/anotherworldofworld@gmail.com), or just create an [issue](https://github.com/0xAX/linux-internals/issues/new). In the next part, we will continue to dive into system calls in the Linux kernel and see the implementation of the [read](http://man7.org/linux/man-pages/man2/read.2.html) system call.

**Please note that English is not my first language and I am really sorry for any inconvenience. If you find any mistakes please send me PR to [linux-insides](https://github.com/0xAX/linux-internals" \t "_blank).**

**OPEN(2) Linux Programmer's Manual OPEN(2)**

## NAME         [top](https://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

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

## SYNOPSIS         [top](https://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**#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***);**

/\* Documented separately, in [openat2(2)](https://man7.org/linux/man-pages/man2/openat2.2.html): \*/

**int openat2(int** *dirfd***, const char \****pathname***,**

**const struct open\_how \****how***, size\_t** *size***);**

Feature Test Macro Requirements for glibc (see

[feature\_test\_macros(7)](https://man7.org/linux/man-pages/man7/feature_test_macros.7.html)):

**openat**():

Since glibc 2.10:

\_POSIX\_C\_SOURCE >= 200809L

Before glibc 2.10:

\_ATFILE\_SOURCE

## DESCRIPTION         [top](https://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

## 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,

nonnegative integer that is an index to an entry in the process's

table of open file descriptors. The file descriptor is used in

subsequent system calls ([read(2)](https://man7.org/linux/man-pages/man2/read.2.html), [write(2)](https://man7.org/linux/man-pages/man2/write.2.html), [lseek(2)](https://man7.org/linux/man-pages/man2/lseek.2.html), [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html),

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.

By default, the new file descriptor is set to remain open across

an [execve(2)](https://man7.org/linux/man-pages/man2/execve.2.html) (i.e., the **FD\_CLOEXEC** file descriptor flag described

in [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html) 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)](https://man7.org/linux/man-pages/man2/lseek.2.html)).

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)](https://man7.org/linux/man-pages/man2/fcntl.2.html) 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)](https://man7.org/linux/man-pages/man2/write.2.html),

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

if with [lseek(2)](https://man7.org/linux/man-pages/man2/lseek.2.html). 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)](https://man7.org/linux/man-pages/man2/fcntl.2.html)) 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)](https://man7.org/linux/man-pages/man2/fcntl.2.html) 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)](https://man7.org/linux/man-pages/man2/fcntl.2.html) **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)](https://man7.org/linux/man-pages/man2/fcntl.2.html)

**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)](https://man7.org/linux/man-pages/man2/fcntl.2.html) at the same time as another thread does a

[fork(2)](https://man7.org/linux/man-pages/man2/fork.2.html) plus [execve(2)](https://man7.org/linux/man-pages/man2/execve.2.html). 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)](https://man7.org/linux/man-pages/man2/fork.2.html).

(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 *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.

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)](https://man7.org/linux/man-pages/man8/mount.8.html).

The *mode* argument specifies the file mode bits to be

applied when a new file is created. 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 supplied if **O\_CREAT** or **O\_TMPFILE** is

specified in *flags*; if it is not supplied, some arbitrary

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 absence 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)](https://man7.org/linux/man-pages/man7/inode.7.html)).

**S\_ISVTX** 0001000 sticky bit (see [inode(7)](https://man7.org/linux/man-pages/man7/inode.7.html)).

**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)](https://man7.org/linux/man-pages/man8/raw.8.html).

**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)](https://man7.org/linux/man-pages/man3/opendir.3.html) 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)](https://man7.org/linux/man-pages/man2/write.2.html) (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)](https://man7.org/linux/man-pages/man2/write.2.html) was followed by a

call to [fdatasync(2)](https://man7.org/linux/man-pages/man2/fdatasync.2.html)). *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**() fails with the error **EEXIST**.

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.

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)](https://man7.org/linux/man-pages/man2/link.2.html) to make a link to the lockfile. If [link(2)](https://man7.org/linux/man-pages/man2/link.2.html)

returns 0, the lock is successful. Otherwise, use [stat(2)](https://man7.org/linux/man-pages/man2/stat.2.html)

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)](https://man7.org/linux/man-pages/man7/feature_test_macros.7.html)).

**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)](https://man7.org/linux/man-pages/man2/read.2.html).

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)](https://man7.org/linux/man-pages/man4/tty.4.html)—it will

not become the process's controlling terminal even if the

process does not have one.

**O\_NOFOLLOW**

If the trailing component (i.e., basename) of *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 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)](https://man7.org/linux/man-pages/man2/poll.2.html), [select(2)](https://man7.org/linux/man-pages/man2/select.2.html), [epoll(7)](https://man7.org/linux/man-pages/man7/epoll.7.html), 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 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)](https://man7.org/linux/man-pages/man7/fifo.7.html).

For a discussion of the effect of **O\_NONBLOCK** in

conjunction with mandatory file locks and with file

leases, see [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html).

**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)](https://man7.org/linux/man-pages/man2/read.2.html), [write(2)](https://man7.org/linux/man-pages/man2/write.2.html), [fchmod(2)](https://man7.org/linux/man-pages/man2/fchmod.2.html),

[fchown(2)](https://man7.org/linux/man-pages/man2/fchown.2.html), [fgetxattr(2)](https://man7.org/linux/man-pages/man2/fgetxattr.2.html), [ioctl(2)](https://man7.org/linux/man-pages/man2/ioctl.2.html), [mmap(2)](https://man7.org/linux/man-pages/man2/mmap.2.html)) fail with the

error **EBADF**.

The following operations *can* be performed on the resulting

file descriptor:

\* [close(2)](https://man7.org/linux/man-pages/man2/close.2.html).

\* [fchdir(2)](https://man7.org/linux/man-pages/man2/fchdir.2.html), if the file descriptor refers to a directory

(since Linux 3.5).

\* [fstat(2)](https://man7.org/linux/man-pages/man2/fstat.2.html) (since Linux 3.6).

\* [fstatfs(2)](https://man7.org/linux/man-pages/man2/fstatfs.2.html) (since Linux 3.12).

\* Duplicating the file descriptor ([dup(2)](https://man7.org/linux/man-pages/man2/dup.2.html), [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html)

**F\_DUPFD**, etc.).

\* Getting and setting file descriptor flags ([fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html)

**F\_GETFD** and **F\_SETFD**).

\* Retrieving open file status flags using the [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html)

**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)](https://man7.org/linux/man-pages/man2/linkat.2.html) 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)](https://man7.org/linux/man-pages/man7/unix.7.html)).

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 object itself (but does require

execute permission on the directories in the path prefix).

Depending on the subsequent operation, a check for

suitable file permissions may be performed (e.g.,

[fchdir(2)](https://man7.org/linux/man-pages/man2/fchdir.2.html) requires execute permission on the directory

referred 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)](https://man7.org/linux/man-pages/man2/fchdir.2.html), [fstat(2)](https://man7.org/linux/man-pages/man2/fstat.2.html)) does not

require read permission on the object.

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)](https://man7.org/linux/man-pages/man2/fchownat.2.html),

[fstatat(2)](https://man7.org/linux/man-pages/man2/fstatat.2.html), [linkat(2)](https://man7.org/linux/man-pages/man2/linkat.2.html), and [readlinkat(2)](https://man7.org/linux/man-pages/man2/readlinkat.2.html) with an empty

pathname to have the calls operate on the symbolic link.

If *pathname* 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)](https://man7.org/linux/man-pages/man2/fstatfs.2.html) can then be used to determine if it is, in

fact, an untriggered automount point (**.f\_type ==**

**AUTOFS\_SUPER\_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)](https://man7.org/linux/man-pages/man3/fexecve.3.html).

**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)](https://man7.org/linux/man-pages/man2/write.2.html) (or similar) returns, the output data

and associated file metadata have been transferred to the

underlying hardware (i.e., as though each [write(2)](https://man7.org/linux/man-pages/man2/write.2.html) was

followed by a call to [fsync(2)](https://man7.org/linux/man-pages/man2/fsync.2.html)). *See NOTES below*.

**O\_TMPFILE** (since Linux 3.11)

Create an unnamed temporary regular 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)](https://man7.org/linux/man-pages/man2/linkat.2.html) 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)](https://man7.org/linux/man-pages/man3/tmpfile.3.html) 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)](https://man7.org/linux/man-pages/man2/fchown.2.html),

[fchmod(2)](https://man7.org/linux/man-pages/man2/fchmod.2.html), [fsetxattr(2)](https://man7.org/linux/man-pages/man2/fsetxattr.2.html), etc.) before being atomically

linked into the filesystem in a fully formed state

(using [linkat(2)](https://man7.org/linux/man-pages/man2/linkat.2.html) 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 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 *pathname*

argument as follows:

\* If the pathname given in *pathname* is absolute, then *dirfd* is

ignored.

\* If the pathname given in *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 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). 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 *pathname* 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 *pathname* is absolute.)

**openat2(2)**

The [openat2(2)](https://man7.org/linux/man-pages/man2/openat2.2.html) system call is an extension of **openat**(), and

provides a superset of the features of **openat**(). It is

documented separately, in [openat2(2)](https://man7.org/linux/man-pages/man2/openat2.2.html).

## RETURN VALUE         [top](https://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

On success, **open**(), **openat**(), and **creat**() return the new file

descriptor (a nonnegative integer). On error, -1 is returned and

[*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate the error.

## ERRORS         [top](https://man7.org/linux/man-pages/man2/open.2.html#top_of_page)

**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)](https://man7.org/linux/man-pages/man7/path_resolution.7.html).)

**EACCES** Where **O\_CREAT** is specified, the *protected\_fifos* or

*protected\_regular* sysctl is enabled, 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 containing 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(5)](https://man7.org/linux/man-pages/man5/proc.5.html).

**EBADF** (**openat**()) *pathname* is relative but *dirfd* is neither

**AT\_FDCWD** nor a valid file descriptor.

**EBUSY O\_EXCL** was specified in *flags* and *pathname* 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** *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)](https://man7.org/linux/man-pages/man7/fifo.7.html)), the call was interrupted by a

signal handler; see [signal(7)](https://man7.org/linux/man-pages/man7/signal.7.html).

**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** was specified in *flags* and the final component

("basename") of the new file's *pathname* is invalid (e.g.,

it contains characters not permitted by the underlying

filesystem).

**EINVAL** The final component ("basename") of *pathname* is invalid

(e.g., it contains characters not permitted by the

underlying filesystem).

**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)](https://man7.org/linux/man-pages/man2/getrlimit.2.html)).

**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.

**ENOENT** 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)](https://man7.org/linux/man-pages/man7/pipe.7.html).

**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.

**ENOTDIR**

(**openat**()) *pathname* 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 *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)](https://man7.org/linux/man-pages/man2/fcntl.2.html).

**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.

**ETXTBSY**

*pathname* refers to a file that is currently in use as a

swap file, and the **O\_TRUNC** flag was specified.

**ETXTBSY**

*pathname* refers to a file that is currently being read by

the kernel (e.g., for module/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)](https://man7.org/linux/man-pages/man2/fcntl.2.html)).