How does the open system call work

Introduction

This is the fifth part of the chapter that describes system calls 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. 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 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 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 filesystem:

```
$ sudo ls /proc/1/fd/
```

Definition of the open system call

If you have read the fourth part of the linux-insides 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 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 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 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:

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 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 architecture, but in our case the x86_64 does not provide definition of the force_o_largefile and it will be used from include/linux/fcntl.h.

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 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 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 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 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 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 system call.
At the next step we check that a file is not tried to be opened via fanotify and without the O_CLOEXEC flag:

```
flags &= ~FMODE_NONOTIFY & ~O_CLOEXEC;
```

We do this to not leak a 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) + execve) 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. 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 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 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 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 linux kernel source code file and starts from initialization of the nameidata structure. This structure will provide a link to a file inode. 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 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 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 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 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.

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, drop me an email, or just create an issue. In the next part, we will continue to dive into system calls in the Linux kernel and see the implementation of the read 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.

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

NAME top

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

SYNOPSIS top

```
#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): */
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)):

openat():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _ATFILE_SOURCE

DESCRIPTION   top

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), write(2), lseek(2), fcntl(2),
etc.) to refer to the open file. The file descriptor returned by
a successful call will be the lowest-numbered file descriptor not
currently open for the process.

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 *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).

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)).

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**() 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) 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 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), select(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 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), 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 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.
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) 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), fstat(2)) 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), fstatat(2), linkat(2), and readlinkat(2) 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) 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).

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) (or similar) returns, 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 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) 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 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 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) 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   top

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   top

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).)

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).

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)), 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** 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)).

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).

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).

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)).