Workshop D

Operating Systems Programming – COMP 3015

1 Introduction

In this workshop you will investigate file I/O and file copy operations.

2 Specification

Following on from the demonstration program `cp1.c` presented in the lecture, we will make a series of modifications to the program.

Firstly, what happens when the `cp1.c` program is asked to copy a file onto itself, i.e. `cp1 input input`? Is this what you expect? Modify the program to do something more sensible! As a hint, two files are the same if they are on the same device and have the same i-node number (which `stat()` can give you), simply comparing the names is not enough.

Secondly, a real copy program will assign the same file permissions to the destination as were on the source, modify your answer to the last part to do this.

Thirdly, real copy programs allow the second argument to be a directory, i.e. you specify a directory for the second part and a copy of the source is placed in that directory with the same name as the source. Modify the answer to the last part to include this functionality. You should allocate the space for the new name dynamically.

Fourthly, your program should rename the original version of the destination file to *filename*`.bak` (i.e. add an additional extension of `.bak`) if the destination file exists, or if the destination is a directory that a file with the same name exists in the directory.

3 Sample Code

3.1 cp1.c

```
/* cp1.c -- simple copy program example
 * Dr Evan Crawford (e.crawford@westernsydney.edu.au)
 * COMP 30015 Operating Systems Programming
 * Practical Case Study D
 * This sample file was adapted from:
 * Molay, B. (2003). cp1.c. In Understanding unix/linux programming.
 *           Source Code, Prentice Hall.
 */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFERSIZE 4096
#define COPYMODE 0644

void oops(char *, char *);

int main(int ac, char *av[])
{
        int in_fd;
        int out_fd;
        int n_chars;
        char buf[BUFFERSIZE];

        if (ac != 3)
        {
                fprintf(stderr, "usage: %s source destination\n", *av);
                exit(1);
        }

        if ((in_fd = open(av[1], O_RDONLY)) == -1)
        {
                oops("Cannot open ", av[1]);
        }

        if ((out_fd = creat(av[2], COPYMODE)) == -1)
        {
                oops("Cannot creat", av[2]);
        }

        while ((n_chars = read(in_fd, buf, BUFFERSIZE)) > 0)
        {
                if (write(out_fd, buf, n_chars) != n_chars)
                {
                        oops("Write error to ", av[2]);
                }
        }
```

```
        if (n_chars == -1)
        {
                oops("Read error from ", av[1]);
        }

        if (close(in_fd) == -1 || close(out_fd) == -1)
        {
                oops("Error closing files", "");
        }

        return 0;
}

void oops(char *s1, char *s2)
{
        fprintf(stderr, "Error: %s ", s1);
        perror(s2);
        exit(1);
}
```

4 Supplementary Materials

The material on the following pages is an extract of the linux system documentation and may prove useful in implementing this Workshop. These manual pages are taken from the Linux *man-pages* Project available at: `http://www.kernel.org/doc/man-pages/`.

NAME

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

SYNOPSIS

```
#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

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.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **open**() for a relative pathname).

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

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

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

RETURN VALUE

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

ERRORS

open(), **openat**(), and **creat**() can fail with the following errors:

EACCES
The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *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).

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.

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)).

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

EBADF
dirfd is not a valid file descriptor.

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

VERSIONS

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

CONFORMING TO

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

openat(): POSIX.1-2008.

openat2(2) is Linux-specific.

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

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

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

NOTES

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

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

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

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

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

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

Open file descriptions

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

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

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

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

Synchronized I/O

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

Linux implements **O_SYNC** and **O_DSYNC**, but not **O_RSYNC**. Somewhat incorrectly, glibc defines **O_RSYNC** to have the same value as **O_SYNC**. (**O_RSYNC** is defined in the Linux header file *<asm/fcntl.h>* on HP PA-RISC, but it is not used.)

O_SYNC provides synchronized I/O *file* integrity completion, meaning write operations will flush data and all associated metadata to the underlying hardware. **O_DSYNC** provides synchronized I/O *data* integrity

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

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

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

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

C library/kernel differences

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

NFS

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

On NFS filesystems with UID mapping enabled, **open**() may return a file descriptor but, for example, **read**(2) requests are denied with **EACCES**. This is because the client performs **open**() by checking the permissions, but UID mapping is performed by the server upon read and write requests.

FIFOs

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

File access mode

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

Linux reserves the special, nonstandard access mode 3 (binary 11) in *flags* to mean: check for read and write permission on the file and return a file descriptor that can't be used for reading or writing. This nonstandard access mode is used by some Linux drivers to return a file descriptor that is to be used only for device-specific **ioctl**(2) operations.

Rationale for openat() and other directory file descriptor APIs

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

First, **openat**() allows an application to avoid race conditions that could occur when using **open**() to open files in directories other than the current working directory. These race conditions result from the fact that some component of the directory prefix given to **open**() could be changed in parallel with the call to **open**(). Suppose, for example, that we wish to create the file *dir1/dir2/xxx.dep* if the file *dir1/dir2/xxx* exists. The problem is that between the existence check and the file-creation step, *dir1* or *dir2* (which might

be symbolic links) could be modified to point to a different location. Such races can be avoided by opening a file descriptor for the target directory, and then specifying that file descriptor as the *dirfd* argument of (say) **fstatat**(2) and **openat**(). The use of the *dirfd* file descriptor also has other benefits:

* the file descriptor is a stable reference to the directory, even if the directory is renamed; and
* the open file descriptor prevents the underlying filesystem from being dismounted, just as when a process has a current working directory on a filesystem.

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

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

When these APIs are given a *dirfd* argument of **AT_FDCWD** or the specified pathname is absolute, then they handle their pathname argument in the same way as the corresponding conventional APIs. However, in this case, several of the APIs have a *flags* argument that provides access to functionality that is not available with the corresponding conventional APIs.

O_DIRECT

The **O_DIRECT** flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. In Linux alignment restrictions vary by filesystem and kernel version and might be absent entirely. However there is currently no filesystem−independent interface for an application to discover these restrictions for a given file or filesystem. Some filesystems provide their own interfaces for doing so, for example the **XFS_IOC_DIOINFO** operation in **xfsctl**(3).

Under Linux 2.4, transfer sizes, the alignment of the user buffer, and the file offset must all be multiples of the logical block size of the filesystem. Since Linux 2.6.0, alignment to the logical block size of the underlying storage (typically 512 bytes) suffices. The logical block size can be determined using the **ioctl**(2) **BLKSSZGET** operation or from the shell using the command:

```
blockdev --getss
```

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

The **O_DIRECT** flag was introduced in SGI IRIX, where it has alignment restrictions similar to those of Linux 2.4. IRIX has also a **fcntl**(2) call to query appropriate alignments, and sizes. FreeBSD 4.x introduced a flag of the same name, but without alignment restrictions.

O_DIRECT support was added under Linux in kernel version 2.4.10. Older Linux kernels simply ignore this flag. Some filesystems may not implement the flag, in which case **open**() fails with the error **EINVAL** if it is used.

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

The behavior of **O_DIRECT** with NFS will differ from local filesystems. Older kernels, or kernels configured in certain ways, may not support this combination. The NFS protocol does not support passing the flag to the server, so **O_DIRECT** I/O will bypass the page cache only on the client; the server may still

cache the I/O. The client asks the server to make the I/O synchronous to preserve the synchronous semantics of **O_DIRECT**. Some servers will perform poorly under these circumstances, especially if the I/O size is small. Some servers may also be configured to lie to clients about the I/O having reached stable storage; this will avoid the performance penalty at some risk to data integrity in the event of server power failure. The Linux NFS client places no alignment restrictions on **O_DIRECT** I/O.

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

BUGS

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

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

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

SEE ALSO

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

COLOPHON

This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

close − close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see **open**(2)), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using **unlink**(2), the file is deleted.

RETURN VALUE

close() returns zero on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EBADF
fd isn't a valid open file descriptor.

EINTR
The **close**() call was interrupted by a signal; see **signal**(7).

EIO An I/O error occurred.

ENOSPC, **EDQUOT**
On NFS, these errors are not normally reported against the first write which exceeds the available storage space, but instead against a subsequent **write**(2), **fsync**(2), or **close**().

See NOTES for a discussion of why **close**() should not be retried after an error.

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use **fsync**(2). (It will depend on the disk hardware at this point.)

The close-on-exec file descriptor flag can be used to ensure that a file descriptor is automatically closed upon a successful **execve**(2); see **fcntl**(2) for details.

Multithreaded processes and close()

It is probably unwise to close file descriptors while they may be in use by system calls in other threads in the same process. Since a file descriptor may be reused, there are some obscure race conditions that may cause unintended side effects.

When dealing with sockets, you have to be sure that there is no **recv**(2) still blocking on it on another thread, otherwise it might block forever, since no more messages will be send via the socket. Be sure to use **shutdown**(2) to shut down all parts the connection before closing the socket.

Furthermore, consider the following scenario where two threads are performing operations on the same file descriptor:

1. One thread is blocked in an I/O system call on the file descriptor. For example, it is trying to **write**(2) to a pipe that is already full, or trying to **read**(2) from a stream socket which currently has no available data.
2. Another thread closes the file descriptor.

The behavior in this situation varies across systems. On some systems, when the file descriptor is closed,

the blocking system call returns immediately with an error.

On Linux (and possibly some other systems), the behavior is different: the blocking I/O system call holds a reference to the underlying open file description, and this reference keeps the description open until the I/O system call completes. (See **open**(2) for a discussion of open file descriptions.) Thus, the blocking system call in the first thread may successfully complete after the **close**() in the second thread.

Dealing with error returns from close()

A careful programmer will check the return value of **close**(), since it is quite possible that errors on a previous **write**(2) operation are reported only on the final **close**() that releases the open file description. Failing to check the return value when closing a file may lead to *silent* loss of data. This can especially be observed with NFS and with disk quota.

Note, however, that a failure return should be used only for diagnostic purposes (i.e., a warning to the application that there may still be I/O pending or there may have been failed I/O) or remedial purposes (e.g., writing the file once more or creating a backup).

Retrying the **close**() after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to be closed. This can occur because the Linux kernel *always* releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Many other implementations similarly always close the file descriptor (except in the case of **EBADF**, meaning that the file descriptor was invalid) even if they subsequently report an error on return from **close**(). POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

A careful programmer who wants to know about I/O errors may precede **close**() with a call to **fsync**(2).

The **EINTR** error is a somewhat special case. Regarding the **EINTR** error, POSIX.1-2008 says:

> If **close**() is interrupted by a signal that is to be caught, it shall return −1 with *errno* set to **EINTR** and the state of *fildes* is unspecified.

This permits the behavior that occurs on Linux and many other implementations, where, as with other errors that may be reported by **close**(), the file descriptor is guaranteed to be closed. However, it also permits another possibility: that the implementation returns an **EINTR** error and keeps the file descriptor open. (According to its documentation, HP-UX's **close**() does this.) The caller must then once more use **close**() to close the file descriptor, to avoid file descriptor leaks. This divergence in implementation behaviors provides a difficult hurdle for portable applications, since on many implementations, **close**() must not be called again after an **EINTR** error, and on at least one, **close**() must be called again. There are plans to address this conundrum for the next major release of the POSIX.1 standard.

SEE ALSO

close_range(2), **fcntl**(2), **fsync**(2), **open**(2), **shutdown**(2), **unlink**(2), **fclose**(3)

COLOPHON

This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

read − read from a file descriptor

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. See also NOTES.

On error, −1 is returned, and *errno* is set to indicate the error. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS

EAGAIN

The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. See **open**(2) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**

The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the read would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF

fd is not a valid file descriptor or is not open for reading.

EFAULT

buf is outside your accessible address space.

EINTR

The call was interrupted by a signal before any data was read; see **signal**(7).

EINVAL

fd is attached to an object which is unsuitable for reading; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EINVAL

fd was created via a call to **timerfd_create**(2) and the wrong size buffer was given to **read**(); see **timerfd_create**(2) for further information.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking **SIGTTIN** or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk

or tape. A further possible cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost locks* section of **fcntl**(2) for further details.

EISDIR
fd refers to a directory.

Other errors may occur, depending on the object connected to *fd*.

CONFORMING TO
SVr4, 4.3BSD, POSIX.1-2001.

NOTES
The types *size_t* and *ssize_t* are, respectively, unsigned and signed integer data types specified by POSIX.1.

On Linux, **read**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

On NFS filesystems, reading small amounts of data will update the timestamp only the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave *st_atime* (last file access time) updates to the server, and client side reads satisfied from the client's cache will not cause *st_atime* updates on the server as there are no server-side reads. UNIX semantics can be obtained by disabling client-side attribute caching, but in most situations this will substantially increase server load and decrease performance.

BUGS
According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

> All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **read**() and **readv**(2). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, on Linux before version 3.14, this was not the case: if two processes that share an open file description (see **open**(2)) perform a **read**() (or **readv**(2)) at the same time, then the I/O operations were not atomic with respect updating the file offset, with the result that the reads in the two processes might (incorrectly) overlap in the blocks of data that they obtained. This problem was fixed in Linux 3.14.

SEE ALSO
close(2), **fcntl**(2), **ioctl**(2), **lseek**(2), **open**(2), **pread**(2), **readdir**(2), **readlink**(2), **readv**(2), **select**(2), **write**(2), **fread**(3)

COLOPHON
This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see **setrlimit**(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also **pipe**(7).)

For a seekable file (i.e., one to which **lseek**(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was **open**(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a **read**(2) that can be proved to occur after a **write**() has returned will return the new data. Note that not all filesystems are POSIX conforming.

According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE

On success, the number of bytes written is returned. On error, −1 is returned, and *errno* is set to indicate the error.

Note that a successful **write**() may transfer fewer than *count* bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked **write**() to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes. In the event of a partial write, the caller can make another **write**() call to transfer the remaining bytes. The subsequent call will either transfer further bytes or may result in an error (e.g., if the disk is now full).

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if one of the errors below is detected. If no errors are detected, or error detection is not performed, 0 is returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

ERRORS

EAGAIN

The file descriptor *fd* refers to a file other than a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. See **open**(2) for further details on the **O_NONBLOCK** flag.

EAGAIN or **EWOULDBLOCK**

The file descriptor *fd* refers to a socket and has been marked nonblocking (**O_NONBLOCK**), and the write would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF

fd is not a valid file descriptor or is not open for writing.

EDESTADDRREQ

fd refers to a datagram socket for which a peer address has not been set using **connect**(2).

EDQUOT

The user's quota of disk blocks on the filesystem containing the file referred to by *fd* has been exhausted.

EFAULT
buf is outside your accessible address space.

EFBIG
An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process's file size limit, or to write at a position past the maximum allowed offset.

EINTR
The call was interrupted by a signal before any data was written; see **signal**(7).

EINVAL
fd is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EIO A low-level I/O error occurred while modifying the inode. This error may relate to the write-back of data written by an earlier **write**(), which may have been issued to a different file descriptor on the same file. Since Linux 4.13, errors from write-back come with a promise that they *may* be reported by subsequent. **write**() requests, and *will* be reported by a subsequent **fsync**(2) (whether or not they were also reported by **write**()). An alternate cause of **EIO** on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the *Lost locks* section of **fcntl**(2) for further details.

ENOSPC
The device containing the file referred to by *fd* has no room for the data.

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

EPIPE *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a **SIGPIPE** signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)

Other errors may occur, depending on the object connected to *fd*.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

Under SVr4 a write may be interrupted and return **EINTR** at any point, not just before any data is written.

NOTES

The types *size_t* and *ssize_t* are, respectively, unsigned and signed integer data types specified by POSIX.1.

A successful return from **write**() does not make any guarantee that data has been committed to disk. On some filesystems, including NFS, it does not even guarantee that space has successfully been reserved for the data. In this case, some errors might be delayed until a future **write**(), **fsync**(2), or even **close**(2). The only way to be sure is to call **fsync**(2) after you are done writing all your data.

If a **write**() is interrupted by a signal handler before any bytes are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.

On Linux, **write**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

An error return value while performing **write**() using direct I/O does not mean the entire write has failed. Partial data may be written and the data at the file offset on which the **write**() was attempted should be considered inconsistent.

BUGS

According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

> All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are **write**() and **writev**(2). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, on Linux before version 3.14, this was not the case: if two processes that share an open file description (see **open**(2)) perform a **write**() (or **writev**(2)) at the same time, then the I/O operations were not atomic with respect to updating the file offset, with the result that the blocks of data output by the two processes might (incorrectly) overlap. This problem was fixed in Linux 3.14.

SEE ALSO

close(2), **fcntl**(2), **fsync**(2), **ioctl**(2), **lseek**(2), **open**(2), **pwrite**(2), **read**(2), **select**(2), **writev**(2), **fwrite**(3)

COLOPHON

This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

stat, fstat, lstat, fstatat – get file status

SYNOPSIS

```
#include <sys/stat.h>

int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *restrict pathname,
         struct stat *restrict statbuf);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fstatat(int dirfd, const char *restrict pathname,
         struct stat *restrict statbuf, int flags);
```

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

lstat():

/* Since glibc 2.20 */ _DEFAULT_SOURCE
|| _XOPEN_SOURCE >= 500
|| /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L
|| /* Glibc 2.19 and earlier */ _BSD_SOURCE

fstatat():

Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *statbuf*. No permissions are required on the file itself, but—in the case of **stat**(), **fstatat**(), and **lstat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() and **fstatat**() retrieve information about the file pointed to by *pathname*; the differences for **fstatat**() are described below.

lstat() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

The stat structure

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
```

```
           precision for the following timestamp fields.
           For the details before Linux 2.6, see NOTES. */

        struct timespec st_atim;  /* Time of last access */
        struct timespec st_mtim;  /* Time of last modification */
        struct timespec st_ctim;  /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
    };
```

Note: the order of fields in the *stat* structure varies somewhat across architectures. In addition, the definition above does not show the padding bytes that may be present between some fields on various architectures. Consult the glibc and kernel source code if you need to know the details.

Note: for performance and simplicity reasons, different fields in the *stat* structure may contain state information from different moments during the execution of the system call. For example, if *st_mode* or *st_uid* is changed by another process by calling **chmod**(2) or **chown**(2), **stat**() might return the old *st_mode* together with the new *st_uid*, or the old *st_uid* together with the new *st_mode*.

The fields in the *stat* structure are as follows:

st_dev This field describes the device on which this file resides. (The **major**(3) and **minor**(3) macros may be useful to decompose the device ID in this field.)

st_ino This field contains the file's inode number.

st_mode
This field contains the file type and mode. See **inode**(7) for further information.

st_nlink
This field contains the number of hard links to the file.

st_uid This field contains the user ID of the owner of the file.

st_gid This field contains the ID of the group owner of the file.

st_rdev This field describes the device that this file (inode) represents.

st_size This field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

st_blksize
This field gives the "preferred" block size for efficient filesystem I/O.

st_blocks
This field indicates the number of blocks allocated to the file, in 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

st_atime
This is the time of the last access of file data.

st_mtime
This is the time of last modification of file data.

st_ctime
This is the file's last status change timestamp (time of last change to the inode).

For further information on the above fields, see **inode**(7).

fstatat()

The **fstatat**() system call is a more general interface for accessing file information which can still provide exactly the behavior of each of **stat**(), **lstat**(), and **fstat**().

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 **stat**() and **lstat**() for a relative pathname).

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

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

flags can either be 0, or include one or more of the following flags ORed:

AT_EMPTY_PATH (since Linux 2.6.39)
: If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the **open**(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory, and the behavior of **fstatat**() is similar to that of **fstat**(). If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_NO_AUTOMOUNT (since Linux 2.6.38)
: Don't automount the terminal ("basename") component of *pathname* if it is a directory that is an automount point. This allows the caller to gather attributes of an automount point (rather than the location it would mount). Since Linux 4.14, also don't instantiate a nonexistent name in an on-demand directory such as used for automounter indirect maps. This flag has no effect if the mount point has already been mounted over.

  Both **stat**() and **lstat**() act as though **AT_NO_AUTOMOUNT** was set.

  The **AT_NO_AUTOMOUNT** can be used in tools that scan directories to prevent mass-automounting of a directory of automount points.

  This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_SYMLINK_NOFOLLOW
: If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself, like **lstat**(). (By default, **fstatat**() dereferences symbolic links, like **stat**().)

See **openat**(2) for an explanation of the need for **fstatat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
: Search permission is denied for one of the directories in the path prefix of *pathname*. (See also **path_resolution**(7).)

EBADF
: *fd* is not a valid open file descriptor.

EFAULT
: Bad address.

ELOOP
: Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
: *pathname* is too long.

ENOENT
: A component of *pathname* does not exist or is a dangling symbolic link.

ENOENT
: *pathname* is an empty string and **AT_EMPTY_PATH** was not specified in *flags*.

ENOMEM
: Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path prefix of *pathname* is not a directory.

EOVERFLOW
pathname or *fd* refers to a file whose size, inode number, or number of blocks cannot be represented in, respectively, the types *off_t*, *ino_t*, or *blkcnt_t*. This error can occur when, for example, an application compiled on a 32-bit platform without *−D_FILE_OFFSET_BITS=64* calls **stat**() on a file whose size exceeds *(1<<31)−1* bytes.

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

EBADF
dirfd is not a valid file descriptor.

EINVAL
Invalid flag specified in *flags*.

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

VERSIONS

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

CONFORMING TO

stat(), **fstat**(), **lstat**(): SVr4, 4.3BSD, POSIX.1-2001, POSIX.1.2008.

fstatat(): POSIX.1-2008.

According to POSIX.1-2001, **lstat**() on a symbolic link need return valid information only in the *st_size* field and the file type of the *st_mode* field of the *stat* structure. POSIX.1-2008 tightens the specification, requiring **lstat**() to return valid information in all fields except the mode bits in *st_mode*.

Use of the *st_blocks* and *st_blksize* fields may be less portable. (They were introduced in BSD. The interpretation differs between systems, and possibly on a single system when NFS mounts are involved.)

NOTES

Timestamp fields

Older kernels and older standards did not support nanosecond timestamp fields. Instead, there were three timestamp fields—*st_atime*, *st_mtime*, and *st_ctime*—typed as *time_t* that recorded timestamps with one-second precision.

Since kernel 2.5.48, the *stat* structure supports nanosecond resolution for the three file timestamp fields. The nanosecond components of each timestamp are available via names of the form *st_atim.tv_nsec*, if suitable feature test macros are defined. Nanosecond timestamps were standardized in POSIX.1-2008, and, starting with version 2.12, glibc exposes the nanosecond component names if **_POSIX_C_SOURCE** is defined with the value 200809L or greater, or **_XOPEN_SOURCE** is defined with the value 700 or greater. Up to and including glibc 2.19, the definitions of the nanoseconds components are also defined if **_BSD_SOURCE** or **_SVID_SOURCE** is defined. If none of the aforementioned macros are defined, then the nanosecond values are exposed with names of the form *st_atimensec*.

C library/kernel differences

Over time, increases in the size of the *stat* structure have led to three successive versions of **stat**(): *sys_stat*() (slot *__NR_oldstat*), *sys_newstat*() (slot *__NR_stat*), and *sys_stat64()* (slot *__NR_stat64*) on 32-bit platforms such as i386. The first two versions were already present in Linux 1.0 (albeit with different names); the last was added in Linux 2.4. Similar remarks apply for **fstat**() and **lstat**().

The kernel-internal versions of the *stat* structure dealt with by the different versions are, respectively:

__old_kernel_stat
The original structure, with rather narrow fields, and no padding.

stat Larger *st_ino* field and padding added to various parts of the structure to allow for future expansion.

stat64 Even larger *st_ino* field, larger *st_uid* and *st_gid* fields to accommodate the Linux-2.4 expansion of UIDs and GIDs to 32 bits, and various other enlarged fields and further padding in the structure. (Various padding bytes were eventually consumed in Linux 2.6, with the advent of 32-bit device IDs and nanosecond components for the timestamp fields.)

The glibc **stat**() wrapper function hides these details from applications, invoking the most recent version of the system call provided by the kernel, and repacking the returned information if required for old binaries.

On modern 64-bit systems, life is simpler: there is a single **stat**() system call and the kernel deals with a *stat* structure that contains fields of a sufficient size.

The underlying system call employed by the glibc **fstatat**() wrapper function is actually called **fstatat64**() or, on some architectures, **newfstatat**().

EXAMPLES

The following program calls **lstat**() and displays selected fields in the returned *stat* structure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdint.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysmacros.h>

int
main(int argc, char *argv[])
{
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        exit(EXIT_FAILURE);
    }

    printf("ID of containing device:  [%jx,%jx]\n",
            (uintmax_t) major(sb.st_dev),
            (uintmax_t) minor(sb.st_dev));

    printf("File type:                ");

    switch (sb.st_mode & S_IFMT) {
    case S_IFBLK:  printf("block device\n");            break;
    case S_IFCHR:  printf("character device\n");        break;
    case S_IFDIR:  printf("directory\n");               break;
    case S_IFIFO:  printf("FIFO/pipe\n");               break;
    case S_IFLNK:  printf("symlink\n");                 break;
    case S_IFREG:  printf("regular file\n");            break;
    case S_IFSOCK: printf("socket\n");                  break;
    default:       printf("unknown?\n");                break;
    }
```

```
        printf("I-node number:            %ju\n", (uintmax_t) sb.st_ino);

        printf("Mode:                     %jo (octal)\n",
                (uintmax_t) sb.st_mode);

        printf("Link count:               %ju\n", (uintmax_t) sb.st_nlink);
        printf("Ownership:                UID=%ju   GID=%ju\n",
                (uintmax_t) sb.st_uid, (uintmax_t) sb.st_gid);

        printf("Preferred I/O block size: %jd bytes\n",
                (intmax_t) sb.st_blksize);
        printf("File size:                %jd bytes\n",
                (intmax_t) sb.st_size);
        printf("Blocks allocated:         %jd\n",
                (intmax_t) sb.st_blocks);

        printf("Last status change:       %s", ctime(&sb.st_ctime));
        printf("Last file access:         %s", ctime(&sb.st_atime));
        printf("Last file modification:   %s", ctime(&sb.st_mtime));

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ls(1), **stat**(1), **access**(2), **chmod**(2), **chown**(2), **readlink**(2), **statx**(2), **utime**(2), **capabilities**(7), **inode**(7), **symlink**(7)

COLOPHON

This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

rename, renameat, renameat2 − change the name or location of a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <stdio.h>

int renameat(int olddirfd, const char *oldpath,
             int newdirfd, const char *newpath);
int renameat2(int olddirfd, const char *oldpath,
             int newdirfd, const char *newpath, unsigned int flags);
```

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

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

renameat2():
_GNU_SOURCE

DESCRIPTION

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using **link**(2)) are unaffected. Open file descriptors for *oldpath* are also unaffected.

Various restrictions determine whether or not the rename operation succeeds: see ERRORS below.

If *newpath* already exists, it will be atomically replaced, so that there is no point at which another process attempting to access *newpath* will find it missing. However, there will probably be a window in which both *oldpath* and *newpath* refer to the file being renamed.

If *oldpath* and *newpath* are existing hard links referring to the same file, then **rename**() does nothing, and returns a success status.

If *newpath* exists but the operation fails for some reason, **rename**() guarantees to leave an instance of *newpath* in place.

oldpath can specify a directory. In this case, *newpath* must either not exist, or it must specify an empty directory.

If *oldpath* refers to a symbolic link, the link is renamed; if *newpath* refers to a symbolic link, the link will be overwritten.

renameat()

The **renameat**() system call operates in exactly the same way as **rename**(), except for the differences described here.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by **rename**() for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value **AT_FDCWD**, then *oldpath* is interpreted relative to the current working directory of the calling process (like **rename**()).

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor *newdirfd*.

See **openat**(2) for an explanation of the need for **renameat**().

renameat2()

renameat2() has an additional *flags* argument. A **renameat2**() call with a zero *flags* argument is equivalent to **renameat**().

The *flags* argument is a bit mask consisting of zero or more of the following flags:

RENAME_EXCHANGE

Atomically exchange *oldpath* and *newpath*. Both pathnames must exist but may be of different types (e.g., one could be a non-empty directory and the other a symbolic link).

RENAME_NOREPLACE

Don't overwrite *newpath* of the rename. Return an error if *newpath* already exists.

RENAME_NOREPLACE can't be employed together with **RENAME_EXCHANGE**.

RENAME_NOREPLACE requires support from the underlying filesystem. Support for various filesystems was added as follows:

* ext4 (Linux 3.15);
* btrfs, tmpfs, and cifs (Linux 3.17);
* xfs (Linux 4.0);
* Support for many other filesystems was added in Linux 4.9, including ext2, minix, reiserfs, jfs, vfat, and bpf.

RENAME_WHITEOUT (since Linux 3.18)

This operation makes sense only for overlay/union filesystem implementations.

Specifying **RENAME_WHITEOUT** creates a "whiteout" object at the source of the rename at the same time as performing the rename. The whole operation is atomic, so that if the rename succeeds then the whiteout will also have been created.

A "whiteout" is an object that has special meaning in union/overlay filesystem constructs. In these constructs, multiple layers exist and only the top one is ever modified. A whiteout on an upper layer will effectively hide a matching file in the lower layer, making it appear as if the file didn't exist.

When a file that exists on the lower layer is renamed, the file is first copied up (if not already on the upper layer) and then renamed on the upper, read-write layer. At the same time, the source file needs to be "whiteouted" (so that the version of the source file in the lower layer is rendered invisible). The whole operation needs to be done atomically.

When not part of a union/overlay, the whiteout appears as a character device with a {0,0} device number. (Note that other union/overlay implementations may employ different methods for storing whiteout entries; specifically, BSD union mount employs a separate inode type, **DT_WHT**, which, while supported by some filesystems available in Linux, such as CODA and XFS, is ignored by the kernel's whiteout support code, as of Linux 4.19, at least.)

RENAME_WHITEOUT requires the same privileges as creating a device node (i.e., the **CAP_MKNOD** capability).

RENAME_WHITEOUT can't be employed together with **RENAME_EXCHANGE**.

RENAME_WHITEOUT requires support from the underlying filesystem. Among the filesystems that support it are tmpfs (since Linux 3.18), ext4 (since Linux 3.18), XFS (since Linux 4.1), f2fs (since Linux 4.2), btrfs (since Linux 4.7), and ubifs (since Linux 4.9).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES

Write permission is denied for the directory containing *oldpath* or *newpath*, or, search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*, or *oldpath* is a

directory and does not allow write permission (needed to update the .. entry). (See also **path_resolution**(7).)

EBUSY
The rename fails because *oldpath* or *newpath* is a directory that is in use by some process (perhaps as current working directory, or as root directory, or because it was open for reading) or is in use by the system (for example as mount point), while the system considers this an error. (Note that there is no requirement to return **EBUSY** in such cases—there is nothing wrong with doing the rename anyway—but it is allowed to return **EBUSY** if the system cannot otherwise handle such situations.)

EDQUOT
The user's quota of disk blocks on the filesystem has been exhausted.

EFAULT
oldpath or *newpath* points outside your accessible address space.

EINVAL
The new pathname contained a path prefix of the old, or, more generally, an attempt was made to make a directory a subdirectory of itself.

EISDIR
newpath is an existing directory, but *oldpath* is not a directory.

ELOOP
Too many symbolic links were encountered in resolving *oldpath* or *newpath*.

EMLINK
oldpath already has the maximum number of links to it, or it was a directory and the directory containing *newpath* has the maximum number of links.

ENAMETOOLONG
oldpath or *newpath* was too long.

ENOENT
The link named by *oldpath* does not exist; or, a directory component in *newpath* does not exist; or, *oldpath* or *newpath* is an empty string.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
The device containing the file has no room for the new directory entry.

ENOTDIR
A component used as a directory in *oldpath* or *newpath* is not, in fact, a directory. Or, *oldpath* is a directory, and *newpath* exists but is not a directory.

ENOTEMPTY or **EEXIST**
newpath is a nonempty directory, that is, contains entries other than "." and "..".

EPERM or **EACCES**
The directory containing *oldpath* has the sticky bit (**S_ISVTX**) set and the process's effective user ID is neither the user ID of the file to be deleted nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or *newpath* is an existing file and the directory containing it has the sticky bit set and the process's effective user ID is neither the user ID of the file to be replaced nor that of the directory containing it, and the process is not privileged (Linux: does not have the **CAP_FOWNER** capability); or the filesystem containing *pathname* does not support renaming of the type requested.

EROFS
The file is on a read-only filesystem.

EXDEV
oldpath and *newpath* are not on the same mounted filesystem. (Linux permits a filesystem to be mounted at multiple points, but **rename**() does not work across different mount points, even if the same filesystem is mounted on both.)

The following additional errors can occur for **renameat**() and **renameat2**():

EBADF
olddirfd or *newdirfd* is not a valid file descriptor.

ENOTDIR
oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

The following additional errors can occur for **renameat2**():

EEXIST
flags contains **RENAME_NOREPLACE** and *newpath* already exists.

EINVAL
An invalid flag was specified in *flags*.

EINVAL
Both **RENAME_NOREPLACE** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL
Both **RENAME_WHITEOUT** and **RENAME_EXCHANGE** were specified in *flags*.

EINVAL
The filesystem does not support one of the flags in *flags*.

ENOENT
flags contains **RENAME_EXCHANGE** and *newpath* does not exist.

EPERM
RENAME_WHITEOUT was specified in *flags*, but the caller does not have the **CAP_MKNOD** capability.

VERSIONS

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

renameat2() was added to Linux in kernel 3.15; library support was added in glibc 2.28.

CONFORMING TO

rename(): 4.3BSD, C89, C99, POSIX.1-2001, POSIX.1-2008.

renameat(): POSIX.1-2008.

renameat2() is Linux-specific.

NOTES

Glibc notes

On older kernels where **renameat**() is unavailable, the glibc wrapper function falls back to the use of **rename**(). When *oldpath* and *newpath* are relative pathnames, glibc constructs pathnames based on the symbolic links in */proc/self/fd* that correspond to the *olddirfd* and *newdirfd* arguments.

BUGS

On NFS filesystems, you can not assume that if the operation failed, the file was not renamed. If the server does the rename operation and then crashes, the retransmitted RPC which will be processed when the server is up again causes a failure. The application is expected to deal with this. See **link**(2) for a similar problem.

SEE ALSO

mv(1), **rename**(1), **chmod**(2), **link**(2), **symlink**(2), **unlink**(2), **path_resolution**(7), **symlink**(7)

COLOPHON

This page is part of release 5.12 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.