

**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, shmem, 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 provide that support 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 appropriately.

**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)**, **chmod(2)**, **link(2)**, **symlink(2)**, **unlink(2)**, **path\_resolution(7)**, **symlink(7)**

**COLOPHON**

This page is part of release 5.02 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/>.