

NAME

chmod – change file mode bits

SYNOPSIS

chmod [*OPTION*]... *MODE*[,*MODE*]... *FILE*...

chmod [*OPTION*]... *OCTAL-MODE* *FILE*...

chmod [*OPTION*]... *--reference=RFIL*E *FILE*...

DESCRIPTION

This manual page documents the GNU version of **chmod**. **chmod** changes the file mode bits of each given file according to *mode*, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

The format of a symbolic mode is [**ugo**a. . .][[**-+=**][*perms*. . .] . . .], where *perms* is either zero or more letters from the set **rwXst**, or a single letter from the set **ugo**. Multiple symbolic modes can be given, separated by commas.

A combination of the letters **ugo**a controls which users' access to the file will be changed: the user who owns it (**u**), other users in the file's group (**g**), other users not in the file's group (**o**), or all users (**a**). If none of these are given, the effect is as if (**a**) were given, but bits that are set in the umask are not affected.

The operator **+** causes the selected file mode bits to be added to the existing file mode bits of each file; **-** causes them to be removed; and **=** causes them to be added and causes unmentioned bits to be removed except that a directory's unmentioned set user and group ID bits are not affected.

The letters **rwXst** select file mode bits for the affected users: read (**r**), write (**w**), execute (or search for directories) (**x**), execute/search only if the file is a directory or already has execute permission for some user (**X**), set user or group ID on execution (**s**), restricted deletion flag or sticky bit (**t**). Instead of one or more of these letters, you can specify exactly one of the letters **ugo**: the permissions granted to the user who owns the file (**u**), the permissions granted to other users who are members of the file's group (**g**), and the permissions granted to users that are in neither of the two preceding categories (**o**).

A numeric mode is from one to four octal digits (0–7), derived by adding up the bits with values 4, 2, and 1. Omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and restricted deletion or sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

chmod never changes the permissions of symbolic links; the **chmod** system call cannot change their permissions. This is not a problem since the permissions of symbolic links are never used. However, for each symbolic link listed on the command line, **chmod** changes the permissions of the pointed-to file. In contrast, **chmod** ignores symbolic links encountered during recursive directory traversals.

SETUID AND SETGID BITS

chmod clears the set-group-ID bit of a regular file if the file's group ID does not match the user's effective group ID or one of the user's supplementary group IDs, unless the user has appropriate privileges. Additional restrictions may cause the set-user-ID and set-group-ID bits of *MODE* or *RFIL*E to be ignored. This behavior depends on the policy and functionality of the underlying **chmod** system call. When in doubt, check the underlying system behavior.

For directories **chmod** preserves set-user-ID and set-group-ID bits unless you explicitly specify otherwise. You can set or clear the bits with symbolic modes like **u+s** and **g-s**. To clear these bits for directories with a numeric mode requires an additional leading zero like **00755**, leading minus like **-6000**, or leading equals like **=755**.

RESTRICTED DELETION FLAG OR STICKY BIT

The restricted deletion flag or sticky bit is a single bit, whose interpretation depends on the file type. For directories, it prevents unprivileged users from removing or renaming a file in the directory unless they own the file or the directory; this is called the *restricted deletion flag* for the directory, and is commonly found on world-writable directories like **/tmp**. For regular files on some older systems, the bit saves the program's text image on the swap device so it will load more quickly when run; this is called the *sticky bit*.

OPTIONS

Change the mode of each *FILE* to *MODE*. With **--reference**, change the mode of each *FILE* to that of *RFILE*.

-c, --changes

like verbose but report only when a change is made

-f, --silent, --quiet

suppress most error messages

-v, --verbose

output a diagnostic for every file processed

--no-preserve-root

do not treat '/' specially (the default)

--preserve-root

fail to operate recursively on '/'

--reference=RFILE

use *RFILE*'s mode instead of specifying *MODE* values. *RFILE* is always dereferenced if a symbolic link.

-R, --recursive

change files and directories recursively

--help display this help and exit

--version

output version information and exit

Each *MODE* is of the form '[**ugo**a]*([**-+=**])([**rwXst**]*[**ugo**]))+([**-+=**])[0–7]+'.

AUTHOR

Written by David MacKenzie and Jim Meyering.

REPORTING BUGS

GNU coreutils online help: <<https://www.gnu.org/software/coreutils/>>

Report any translation bugs to <<https://translationproject.org/team/>>

COPYRIGHT

Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <<https://gnu.org/licenses/gpl.html>>.

This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

SEE ALSO

chmod(2)

Full documentation <<https://www.gnu.org/software/coreutils/chmod>> or available locally via: info '(coreutils) chmod invocation'

NAME

clearerr, feof, ferror, fileno – check and reset stream status

SYNOPSIS

```
#include <stdio.h>
```

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fileno(FILE *stream);
```

DESCRIPTION

The function **clearerr()** clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof()** tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr()**.

The function **ferror()** tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the **clearerr()** function.

The function **fileno()** examines the argument *stream* and returns its integer descriptor.

For non-locking counterparts, see **unlocked_stdio(3)**.

ERRORS

These functions should not fail and do not set the external variable *errno*. (However, in case **fileno()** detects that its argument is not a valid stream, it must return -1 and set *errno* to **EBADF**.)

CONFORMING TO

The functions **clearerr()**, **feof()**, and **ferror()** conform to C89 and C99.

SEE ALSO

open(2), **fdopen(3)**, **stdio(3)**, **unlocked_stdio(3)**

NAME

fgetc, getchar, ungetc – FILE get character

LIBRARY

Standard C library (*libc*, $-lc$)

SYNOPSIS

```
#include <stdio.h>
```

```
int fgetc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getchar() is equivalent to *fgetc(stdin)*.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the *stdio* library for the same input stream.

For nonlocking counterparts, see **unlocked_stdio(3)**.

RETURN VALUE

fgetc() and **getchar()** return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

ungetc() returns *c* on success, or **EOF** on error.

On error, *errno* is set to indicate the error.

ERRORS

See **read(2)**.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes(7)**.

Interface	Attribute	Value
fgetc() , getchar() , ungetc()	Thread safety	MT-Safe

STANDARDS

C11, POSIX.1-2008.

HISTORY

POSIX.1-2001, C89.

NOTES

It is not advisable to mix calls to input functions from the *stdio* library with low-level calls to **read(2)** for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), **write(2)**, **ferror(3)**, **fgetwc(3)**, **fgets(3)**, **fopen(3)**, **fread(3)**, **fseek(3)**, **getline(3)**, **gets(3)**, **getwchar(3)**, **puts(3)**, **scanf(3)**, **ungetwc(3)**, **unlocked_stdio(3)**, **feature_test_macros(7)**

NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

```
#include <stdio.h>
```

```
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
int fileno(FILE *stream);
int fclose(FILE *stream);
```

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

- r** Open text file for reading. The stream is positioned at the beginning of the file.
- r+** Open for reading and writing. The stream is positioned at the beginning of the file.
- w** Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- w+** Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- a** Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- a+** Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(const char *name);
int closedir(DIR *dirp);
struct dirent *readdir(DIR *dir);
```

DESCRIPTION opendir

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, **NULL** is returned, and *errno* is set appropriately.

DESCRIPTION closedir

The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.

RETURN VALUE

The **closedir**() function returns 0 on success. On error, -1 is returned, and *errno* is set appropriately.

DESCRIPTION readdir

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dir*. It returns **NULL** on reaching the end-of-file or if an error occurred. It is safe to use **readdir**() inside threads if the pointers passed as *dir* are created by distinct calls to **opendir**().

The data returned by **readdir**() is overwritten by subsequent calls to **readdir**() for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long    d_ino;        /* inode number */
    char    d_name[256];  /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, **NULL** is returned and *errno* is not changed. If an error occurs, **NULL** is returned and *errno* is set appropriately. To distinguish end of stream and from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if **NULL** is returned.

ERRORS

EACCES

Permission denied.

ENOENT

Directory does not exist, or *name* is an empty string.

ENOTDIR

name is not a directory.

NAME

pthread_create – create a new thread / pthread_exit – terminate the calling thread

SYNOPSIS

```
#include <pthread.h>
```

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);
```

```
void pthread_exit(void *retval);
```

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling **pthread_exit**(3), or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling **pthread_exit**(3) with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread. See **pthread_attr_init**(3) for a complete list of thread attributes. The *attr* argument can also be **NULL**, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

pthread_exit terminates the execution of the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push**(3) are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non- **NULL** values associated with them in the calling thread (see **pthread_key_create**(3)). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be consulted from another thread using **pthread_join**(3).

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed to by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

The **pthread_exit** function never returns.

ERRORS

EAGAIN

not enough system resources to create a process for the new thread.

EAGAIN

more than **PTHREAD_THREADS_MAX** threads are already active.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_join(3), **pthread_detach**(3), **pthread_attr_init**(3).

NAME

pthread_join – join with a terminated thread

SYNOPSIS

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Compile and link with *-pthread*.

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not **NULL**, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to **pthread_exit**(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK

A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL

thread is not a joinable thread.

EINVAL

Another thread is already waiting to join with this thread.

ESRCH

No thread with the ID *thread* could be found.

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of *waitpid(-1, &status, 0)*, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLE

See **pthread_create**(3).

SEE ALSO

pthread_cancel(3), **pthread_create**(3), **pthread_detach**(3), **pthread_exit**(3), **pthread_t**(7)