File permissions and attributes

File systems use **permissions** and **attributes** to regulate the level of interaction that system processes can have with files and directories.

Related articles

Users and groups

umask

Access Control Lists

Capabilities

Warning: When used for security purposes, permissions and attributes only defend against attacks launched from the booted system. To protect the stored data from attackers with physical access to the machine, one must also implement **disk encryption**.

Contents

Viewing permissions

Use the **ls** command's `-l` option to view the permissions (or **file mode**) set for the contents of a directory, for example:

```
$ ls -l /path/to/directory

total 128
drwxr-xr-x 2 archie users  4096 Jul  5 21:03 Desktop
drwxr-xr-x 6 archie users  4096 Jul  5 17:37 Documents
drwxr-xr-x 2 archie users  4096 Jul  5 13:45 Downloads
-rw-rw-r-- 1 archie users  5120 Jun 27 08:28 customers.ods
-rw-r--r-- 1 archie users  3339 Jun 27 08:28 todo
-rwxr-xr-x 1 archie users  2048 Jul  6 12:56 myscript.sh
```

The first column is what we must focus on. Taking an example value of `drwxrwxrwx+`, the meaning of each character is explained in the following tables:

`d`	`rwx`	`rwx`	`rwx`	`+`
The file type, technically not part of its permissions. See `info ls -n "What information is listed"` for an explanation of the possible values.	The permissions that the owner has over the file, explained below.	The permissions that the group has over the file, explained below.	The permissions that all the other users have over the file, explained below.	A single character that specifies whether an alternate access method applies to the file. When this character is a space, there is no alternate access method. A `.` character indicates a file with a security context, but no other alternate access method. A file with any other combination of alternate access methods is marked with a `+` character, for example in the case of Access Control Lists.

Each of the three permission triads (`rwx` in the example above) can be made up of the following characters:

	Character	Effect on files	Effect on directories
Read permission (first character)	`-`	The file cannot be read.	The directory's contents cannot be shown.
	`r`	The file can be read.	The directory's contents can be shown.
Write permission (second character)	`-`	The file cannot be modified.	The directory's contents cannot be modified.
	`w`	The file can be modified.	The directory's contents can be modified (create new files or folders; rename or delete existing files or folders); requires the execute permission to be also set, otherwise this permission has no effect.
Execute permission (third character)	`-`	The file cannot be executed.	The directory cannot be accessed with cd.
	`x`	The file can be executed.	The directory can be accessed with cd; this is the only permission bit that in practice can be considered to be "inherited" from the ancestor directories, in fact if *any* folder in the path does not have the `x` bit set, the final file or folder cannot be accessed either, regardless of its permissions; see `path_resolution(7) (https://jlk.fjfi.cvut.cz/arch/manpages/man/path_resolution.7)` for more information.
	`s`	The setuid bit when found in the **u**ser triad; the **setgid** bit when found in the **g**roup triad; it is not found in the **o**thers triad; it also implies that `x` is set.	
	`S`	Same as `s`, but `x` is not set; rare on regular files, and useless on folders.	
	`t`	The sticky bit; it can only be found in the **o**thers triad; it also implies that `x` is set.	
	`T`	Same as `t`, but `x` is not set; rare on regular files, and useless on folders.	

See `info Coreutils -n "Mode Structure"` and chmod(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/chmod.1) for more details.

Examples

Let us see some examples to clarify:

```
drwx------ 6 archie users  4096 Jul  5 17:37 Documents
```

Archie has full access to the Documents directory. He can list, create files and rename, delete any file in Documents, regardless of file permissions. His ability to access a file depends on the file's permission.

```
dr-x------ 6 archie users  4096 Jul  5 17:37 Documents
```

Archie has full access except he can not create, rename, delete any file. He can list the files and (if file's permission empowers) may access an existing file in Documents.

```
d-wx------ 6 archie users  4096 Jul  5 17:37 Documents
```

Archie can not do 'ls' in Documents but if he knows the name of an existing file then he may list, rename, delete or (if file's permission empowers him) access it. Also, he is able to create new files.

```
d--x------ 6 archie users  4096 Jul  5 17:37 Documents
```

Archie is only capable of (if file's permission empowers him) access those files in Documents which he knows of. He can not list already existing files or create, rename, delete any of them.

You should keep in mind that we elaborate on directory permissions and it has nothing to do with the individual file permissions. When you create a new file it is the directory that changes. That is why you need write permission to the directory.

Let us look at another example, this time of a file, not a directory:

```
-rw-r--r-- 1 archie users  5120 Jun 27 08:28 foobar
```

Here we can see the first letter is not `d` but `-`. So we know it is a file, not a directory. Next the owner's permissions are `rw-` so the owner has the ability to read and write but not execute. This may seem odd that the owner does not have all three permissions, but the `x` permission is not needed as it is a text/data file, to be read by a text editor such as Gedit, EMACS, or software like R, and not an executable in its own right (if it contained something like python programming code then it very well could be). The group's permssions are set to `r--`, so the group has the ability to read the file but not write/edit it in any way — it is essentially like setting something to read-only. We can see that the same permissions apply to everyone else as well.

Changing permissions

chmod is a command in Linux and other Unix-like operating systems that allows to *ch*ange the permissions (or access *mod*e) of a file or directory.

Text method

To change the permissions — or *access mode* — of a file, use the *chmod* command in a terminal. Below is the command's general structure:

```
chmod who=permissions filename
```

Where `who` is any from a range of letters, each signifying who is being given the permission. They are as follows:

- `u` : the **user** that owns the file.
- `g` : the **group** that the file belongs to.
- `o` : the **o**ther users, i.e. everyone else.
- `a` : **a**ll of the above; use this instead of typing `ugo` .

The permissions are the same as discussed in **#Viewing permissions** (`r` , `w` and `x`).

Now have a look at some examples using this command. Suppose you became very protective of the Documents directory and wanted to deny everybody but yourself, permissions to read, write, and execute (or in this case search/look) in it:

Before: `drwxr-xr-x 6 archie users 4096 Jul 5 17:37 Documents`

```
$ chmod g= Documents
$ chmod o= Documents
```

After: `drwx------ 6 archie users 4096 Jul 6 17:32 Documents`

Here, because you want to deny permissions, you do not put any letters after the `=` where permissions would be entered. Now you can see that only the owner's permissions are `rwx` and all other permissions are `-`.

This can be reverted with:

Before: `drwx------ 6 archie users 4096 Jul 6 17:32 Documents`

```
$ chmod g=rx Documents
$ chmod o=rx Documents
```

After: `drwxr-xr-x 6 archie users 4096 Jul 6 17:32 Documents`

In the next example, you want to grant read and execute permissions to the group, and other users, so you put the letters for the permissions (`r` and `x`) after the `=`, with no spaces.

You can simplify this to put more than one *`who`* letter in the same command, e.g:

```
$ chmod go=rx Documents
```

Note: It does not matter in which order you put the *`who`* letters or the permission letters in a `chmod` command: you could have `chmod go=rx file` or `chmod og=xr file`. It is all the same.

Now let us consider a second example, suppose you want to change a `foobar` file so that you have read and write permissions, and fellow users in the group `users` who may be colleagues working on `foobar`, can also read and write to it, but other users can only read it:

Before: `-rw-r--r-- 1 archie users 5120 Jun 27 08:28 foobar`

```
$ chmod g=rw foobar
```

After: `-rw-rw-r-- 1 archie users 5120 Jun 27 08:28 foobar`

This is exactly like the first example, but with a file, not a directory, and you grant write permission (just so as to give an example of granting every permission).

Text method shortcuts

The *chmod* command lets add and subtract permissions from an existing set using `+` or `-` instead of `=`. This is different from the above commands, which essentially re-write the permissions (e.g. to change a permission from `r--` to `rw-`, you still need to include `r` as well as `w` after the `=` in the *chmod* command invocation. If you missed out `r`, it would take away the `r` permission as they are being re-written with the `=`. Using `+` and `-` avoids this by adding or taking away from the *current* set of permissions).

Let us try this `+` and `-` method with the previous example of adding write permissions to the group:

Before: `-rw-r--r-- 1 archie users 5120 Jun 27 08:28 foobar`

```
$ chmod g+w foobar
```

After: `-rw-rw-r-- 1 archie users 5120 Jun 27 08:28 foobar`

Another example, denying write permissions to all (**a**):

Before: `-rw-rw-r-- 1 archie users 5120 Jun 27 08:28 foobar`

```
$ chmod a-w foobar
```

After: `-r--r--r-- 1 archie users 5120 Jun 27 08:28 foobar`

A different shortcut is the special `X` mode: this is not an actual file mode, but it is often used in conjunction with the `-R` option to set the executable bit only for directories, and leave it unchanged for regular files, for example:

```
$ chmod -R a+rX ./data/
```

Copying permissions

It is possible to tell *chmod* to copy the permissions from one class, say the owner, and give those same permissions to group or even all. To do this, instead of putting `r`, `w`, or `x` after the `=`, put another *who* letter. e.g:

Before: `-rw-r--r-- 1 archie users 5120 Jun 27 08:28 foobar`

```
$ chmod g=u foobar
```

After: `-rw-rw-r-- 1 archie users 5120 Jun 27 08:28 foobar`

This command essentially translates to "change the permissions of group (`g=`), to be the same as the owning user (`=u`). Note that you cannot copy a set of permissions as well as grant new ones e.g.:

```
$ chmod g=wu foobar
```

In that case *chmod* throw an error.

Numeric method

chmod can also set permissions using numbers.

Using numbers is another method which allows you to edit the permissions for all three owner, group, and others at the same time, as well as the setuid, setgid, and sticky bits. This basic structure of the code is this:

```
$ chmod xxx filename
```

Where `xxx` is a 3-digit number where each digit can be anything from 0 to 7. The first digit applies to permissions for owner, the second digit applies to permissions for group, and the third digit applies to permissions for all others.

In this number notation, the values `r`, `w`, and `x` have their own number value:

```
r=4
w=2
x=1
```

To come up with a 3-digit number you need to consider what permissions you want owner, group, and user to have, and then total their values up. For example, if you want to grant the owner of a directory read write and execution permissions, and you want group and everyone else to have just read and execute permissions, you would come up with the numerical values like so:

- Owner: `rwx` =4+2+1=7
- Group: `r-x` =4+0+1=5
- Other: `r-x` =4+0+1=5

```
$ chmod 755 filename
```

This is the equivalent of using the following:

```
$ chmod u=rwx filename
$ chmod go=rx filename
```

Most folders and directories are set to `755` to allow reading, writing and execution to the owner, but deny writing to everyone else, and files are normally `644` to allow reading and writing for the owner but just reading for everyone else; refer to the last note on the lack of `x` permissions with non executable files: it is the same thing here.

To see this in action with examples consider the previous example that has been used but with this numerical method applied instead:

Before: `-rw-r--r-- 1 archie users 5120 Jun 27 08:28 foobar`

```
$ chmod 664 foobar
```

After: `-rw-rw-r-- 1 archie users 5120 Jun 27 08:28 foobar`

If this were an executable the number would be `774` if you wanted to grant executable permission to the owner and group. Alternatively if you wanted everyone to only have read permission the number would be `444`. Treating `r` as 4, `w` as 2, and `x` as 1 is probably the easiest way to work out the numerical values for using `chmod xxx filename`, but there is also a binary method, where each permission has a binary number, and then that is in turn converted to a number. It is a bit more convoluted, but here included for completeness.

Consider this permission set:

```
-rwxr-xr--
```

If you put a 1 under each permission granted, and a 0 for every one not granted, the result would be something like this:

```
-rwxrwxr-x
 111111101
```

You can then convert these binary numbers:

```
000=0       100=4
001=1       101=5
010=2       110=6
011=3       111=7
```

The value of the above would therefore be 775.

Consider we wanted to remove the writable permission from group:

```
-rwxr-xr-x
 111101101
```

The value would therefore be 755 and you would use `chmod 755 filename` to remove the writable permission. You will notice you get the same three digit number no matter which method you use. Whether you use text or numbers will depend on personal preference and typing speed. When you want to restore a directory or file to default permissions e.g. read and write (and execute) permission to the owner but deny write permission to everyone else, it may be faster to use `chmod 755/644 filename`. However if you are changing the permissions to something out of the norm, it may be simpler and quicker to use the text method as opposed to trying to convert it to numbers, which may lead to a mistake. It could be argued that there is not any real significant difference in the speed of either method for a user that only needs to use *chmod* on occasion.

You can also use the numeric method to set the `setuid`, `setgid`, and `sticky` bits by using four digits.

```
setuid=4
setgid=2
sticky=1
```

For example, `chmod 2777 filename` will set read/write/executable bits for everyone and also enable the `setgid` bit.

Bulk chmod

Generally directories and files should not have the same permissions. If it is necessary to bulk modify a directory tree, use **find** to selectively modify one or the other.

To *chmod* only directories to 755:

```
$ find directory -type d -exec chmod 755 {} +
```

To *chmod* only files to 644:

```
$ find directory -type f -exec chmod 644 {} +
```

Changing ownership

chown changes the owner of a file or directory, which is quicker and easier than altering the permissions in some cases.

Consider the following example, making a new partition with **GParted** for backup data. Gparted does this all as root so everything belongs to root by default. This is all well and good but when it comes to writing data to the mounted partition, permission is denied for regular users.

```
brw-rw---- 1 root disk 8,    9 Jul  6 16:02 sda9
drwxr-xr-x 5 root root    4096 Jul  6 16:01 Backup
```

As you can see the device in `/dev` is owned by root, as is the mount location (`/media/Backup`). To change the owner of the mount location one can do the following:

Before: `drwxr-xr-x 5 root root 4096 Jul 6 16:01 Backup`

```
# chown archie /media/Backup
```

After: `drwxr-xr-x 5 archie root 4096 Jul 6 16:01 Backup`

Now the partition can have data written to it by the new owner, archie, without altering the permissions (as the owner triad already had `rwx` permissions).

Note:

- `chown` always clears the setuid and setgid bits.
- Non-root users cannot use `chown` to "give away" files they own to another user.

Access Control Lists

Access Control Lists provides an additional, more flexible permission mechanism for file systems by allowing to set permissions for any user or group to any file.

Umask

The **umask** utility is used to control the file-creation mode mask, which determines the initial value of file permission bits for newly created files.

File attributes

Apart from the file mode bits that control **user and group** read, write and execute permissions, several **file systems** support file attributes that enable further customization of allowable file operations. This section describes some of these attributes and how to work with them.

Warning: By default, file attributes are not preserved by **cp**, **rsync**, and other similar programs.

chattr and lsattr

For ext2 and ext3 file systems, the `e2fsprogs (https://www.archlinux.org/packages/?name=e2fsprogs)` package contains the programs **lsattr** and **chattr** that list and change a file's attributes, respectively. Though some are not honored by all file systems, the available attributes are:

- `a` : append only
- `c` : compressed
- `d` : no dump
- `e` : extent format
- `i` : immutable
- `j` : data journalling
- `s` : secure deletion
- `t` : no tail-merging
- `u` : undeletable
- `A` : no atime updates
- `C` : no copy on write
- `D` : synchronous directory updates
- `S` : synchronous updates
- `T` : top of directory hierarchy

For example, if you want to set the immutable bit on some file, use the following command:

```
# chattr +i /path/to/file
```

To remove an attribute on a file just change `+` to `-`.

Extended attributes

From `attr(5) (https://jlk.fjfi.cvut.cz/arch/manpages/man/attr.5)`: "Extended attributes are name:value pairs associated permanently with files and directories". There are four extended attribute classes: security, system, trusted and user.

Warning: By default, extended attributes are not preserved by **cp**, **rsync**, and other similar programs.

User extended attributes

User extended attributes can be used to store arbitrary information about a file. To create one:

```
$ setfattr -n user.checksum -v "3baf9ebce4c664ca8d9e5f6314fb47fb" foo.bar
```

Use getfattr to display extended attributes:

```
$ getfattr -d foo.bar
```

```
# file: foo.bar
user.checksum="3baf9ebce4c664ca8d9e5f6314fb47fb"
```

Capabilities

Extended attributes are also used to set **Capabilities**.

Tips and tricks

Preserve root

Use the `--preserve-root` flag to prevent `chmod` from acting recursively on `/`. This can, for example, prevent one from removing the executable bit systemwide and thus breaking the system. To use this flag every time, set it within an **alias**. See also **[1] (https://www.reddit.com/r/linux/comments/4ni3xe/tifu_sudo_chmod_644/)**.

See also

- **wikipedia:Chattr**
- **Linux File Permission Confusion (http://www.hackinglinuxexposed.com/articles/20030417.html)**
- **Linux File Permission Confusion part 2 (http://www.hackinglinuxexposed.com/articles/20030424.html)**
- **wikipedia:Extended file attributes#Linux**

- **Extended attributes: the good, the not so good, the bad. (http://www.lesbonscomptes.com/pages/extattrs.html)**
- **Backup and restore file permissions in Linux (http://www.concrete5.org/documentation/how-tos/designers/backup-and-restore-file-permissions-in-linux/)**
- **Why is “chmod -R 777 /” destructive? (https://serverfault.com/questions/364677/why-is-chmod-r-777-destructive)**

Retrieved from "https://wiki.archlinux.org/index.php?title=File_permissions_and_attributes&oldid=507325"