‹ (../03-file-transfer/)

Extra Unix Shell Material (../)

› (../05-directory-structure/)

Permissions

Overview

Teaching: 10 min
Exercises: 0 min
Questions

- Understanding file/directory permissions

Objectives

- What are file/directory permissions?
- How to view permissions?
- How to change permissions?
- File/directory permissions in Windows

Unix controls who can read, modify, and run files using *permissions*. We'll discuss how Windows handles permissions at the end of the section: the concepts are similar, but the rules are different.

Let's start with Nelle. She has a unique user name (../reference/#user-name), `nnemo`, and a user ID (../reference/#user-id), 1404.

Why Integer IDs?

Why integers for IDs? Again, the answer goes back to the early 1970s. Character strings like `alan.turing` are of varying length, and comparing one to another takes many instructions. Integers, on the other hand, use a fairly small amount of storage (typically four characters), and can be compared with a single instruction. To make operations fast and simple, programmers often keep track of things internally using integers, then use a lookup table of some kind to translate those integers into user-friendly text for presentation. Of course, programmers being programmers, they will often skip the user-friendly string part and just use the integers, in the same way that someone working in a lab might talk about Experiment 28 instead of "the chronotypical alpha-response trials on anacondas".

Users can belong to any number of groups (../reference/#user-group), each of which has a unique group name (../reference/#user-group-name) and numeric group ID (../reference/#user-group-id). The list of who's in what group is usually stored in the file `/etc/group`. (If you're in front of a Unix machine right now, try running `cat /etc/group` to look at that file.)

Now let's look at files and directories. Every file and directory on a Unix computer belongs to one owner and one group. Along with each file's content, the operating system stores the numeric IDs of the user and group that own it.

The user-and-group model means that for each file every user on the system falls into one of three categories: the owner of the file, someone in the file's group, and everyone else.

For each of these three categories, the computer keeps track of whether people in that category can read the file, write to the file, or execute the file (i.e., run it if it is a program).

For example, if a file had the following set of permissions:

	user	group	all
read	yes	yes	no
write	yes	no	no
execute	no	no	no

it would mean that:

- the file's owner can read and write it, but not run it;
- other people in the file's group can read it, but not modify it or run it; and
- everybody else can do nothing with it at all.

Let's look at this model in action. If we `cd` into the `labs` directory and run `ls -F`, it puts a `*` at the end of `setup`'s name. This is its way of telling us that `setup` is executable, i.e., that it's (probably) something the computer can run.

```
$ cd labs
$ ls -F
```

```
safety.txt    setup*     waiver.txt
```

📌 Necessary But Not Sufficient

The fact that something is marked as executable doesn't actually mean it contains a program of some kind. We could easily mark this HTML file as executable using the commands that are introduced below. Depending on the operating system we're using, trying to "run" it will either fail (because it doesn't contain instructions the computer recognizes) or cause the operating system to open the file with whatever application usually handles it (such as a web browser).

Now let's run the command `ls -l`:

```
$ ls -l
```

```
-rw-rw-r-- 1 vlad bio  1158  2010-07-11 08:22 safety.txt
-rwxr-xr-x 1 vlad bio 31988  2010-07-23 20:04 setup
-rw-rw-r-- 1 vlad bio  2312  2010-07-11 08:23 waiver.txt
```

The `-l` flag tells `ls` to give us a long-form listing. It's a lot of information, so let's go through the columns in turn.

On the right side, we have the files' names. Next to them, moving left, are the times and dates they were last modified. Backup systems and other tools use this information in a variety of ways, but you can use it to tell when you (or anyone else with permission) last changed a file.

Next to the modification time is the file's size in bytes and the names of the user and group that owns it (in this case, `vlad` and `bio` respectively). We'll skip over the second column for now (the one showing `1` for each file) because it's the first column that we care about most. This shows the file's permissions, i.e., who can read, write, or execute it.

Let's have a closer look at one of those permission strings: `-rwxr-xr-x` . The first character tells us what type of thing this is: '-' means it's a regular file, while 'd' means it's a directory, and other characters mean more esoteric things.

The next three characters tell us what permissions the file's owner has. Here, the owner can read, write, and execute the file: `rwx` . The middle triplet shows us the group's permissions. If the permission is turned off, we see a dash, so `r-x` means "read and execute, but not write". The final triplet shows us what everyone who isn't the file's owner, or in the file's group, can do. In this case, it's 'r-x' again, so everyone on the system can look at the file's contents and run it.

To change permissions, we use the `chmod` command (whose name stands for "change mode"). Here's a long-form listing showing the permissions on the final grades in the course Vlad is teaching:

```
$ ls -l final.grd
```

```
-rwxrwxrwx 1 vlad bio  4215  2010-08-29 22:30 final.grd
```

Whoops: everyone in the world can read it—and what's worse, modify it! (They could also try to run the grades file as a program, which would almost certainly not work.)

The command to change the owner's permissions to `rw-` is:

```
$ chmod u=rw final.grd
```

The 'u' signals that we're changing the privileges of the user (i.e., the file's owner), and `rw` is the new set of permissions. A quick `ls -l` shows us that it worked, because the owner's permissions are now set to read and write:

```
$ ls -l final.grd
```

```
-rw-rwxrwx 1 vlad bio  4215  2010-08-30 08:19 final.grd
```

Let's run `chmod` again to give the group read-only permission:

```
$ chmod g=r final.grd
$ ls -l final.grd
```

```
-rw-r--rw- 1 vlad bio  4215  2010-08-30 08:19 final.grd
```

And finally, let's give "all" (everyone on the system who isn't the file's owner or in its group) no permissions at all:

```
$ chmod a= final.grd
$ ls -l final.grd
```

```
-rw-r----- 1 vlad bio  4215  2010-08-30 08:20 final.grd
```

Here, the 'a' signals that we're changing permissions for "all", and since there's nothing on the right of the "=", "all"'s new permissions are empty.

We can search by permissions, too. Here, for example, we can use `-type f -perm -u=x` to find files that the user can execute:

```
$ find . -type f -perm -u=x
```

```
./tools/format
./tools/stats
```

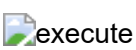
Before we go any further, let's run `ls -a -l` to get a long-form listing that includes directory entries that are normally hidden:

```
$ ls -a -l
```

```
drwxr-xr-x 1 vlad bio     0  2010-08-14 09:55 .
drwxr-xr-x 1 vlad bio  8192  2010-08-27 23:11 ..
-rw-rw-r-- 1 vlad bio  1158  2010-07-11 08:22 safety.txt
-rwxr-xr-x 1 vlad bio 31988  2010-07-23 20:04 setup
-rw-rw-r-- 1 vlad bio  2312  2010-07-11 08:23 waiver.txt
```

The permissions for `.` and `..` (this directory and its parent) start with a 'd'. But look at the rest of their permissions: the 'x' means that "execute" is turned on. What does that mean? A directory isn't a program—how can we "run" it?

In fact, 'x' means something different for directories. It gives someone the right to *traverse* the directory, but not to look at its contents. The distinction is subtle, so let's have a look at an example. Vlad's home directory has three subdirectories called `venus`, `mars`, and `pluto`:



Each of these has a subdirectory in turn called `notes`, and those sub-subdirectories contain various files. If a user's permissions on `venus` are 'r-x', then if she tries to see the contents of `venus` and `venus/notes` using `ls`, the computer lets her see both. If her permissions on `mars` are just 'r–', then she is allowed to read the contents of both `mars` and `mars/notes`. But if her permissions on `pluto` are only '–x', she cannot see what's in the `pluto` directory: `ls pluto` will tell her she doesn't have permission to view its contents. If she tries to look in `pluto/notes`, though, the computer will let her do that. She's allowed to go through `pluto`, but not to look at what's there. This trick gives people a way to make some of their directories visible to the world as a whole without opening up everything else.

What about Windows?

Those are the basics of permissions on Unix. As we said at the outset, though, things work differently on Windows. There, permissions are defined by access control lists (../reference/#access-control-list), or ACLs. An ACL is a list of pairs, each of which combines a “who” with a “what”. For example, you could give the Mummy permission to append data to a file without giving him permission to read or delete it, and give Frankenstein permission to delete a file without being able to see what it contains.

This is more flexible that the Unix model, but it’s also more complex to administer and understand on small systems. (If you have a large computer system, *nothing* is easy to administer or understand.) Some modern variants of Unix support ACLs as well as the older read-write-execute permissions, but hardly anyone uses them.

Challenge

If `ls -l myfile.php` returns the following details:

```
-rwxr-xr-- 1 caro zoo  2312  2014-10-25 18:30 myfile.php
```

Which of the following statements is true?

1. caro (the owner) can read, write, and execute myfile.php
2. caro (the owner) cannot write to myfile.php
3. members of caro (a group) can read, write, and execute myfile.php
4. members of zoo (a group) cannot execute myfile.php

Key Points

- Correct permissions are critical for the security of a system.
- File permissions describe who and what can read, write, modify, and access a file.
- Use `ls -l` to view the permissions for a specific file.
- Use `chmod` to change permissions on a file or directory.

< (../03-file-transfer/)

> (../05-directory-structure/)

Edit on GitHub (https://github.com/swcarpentry/shell-extras/edit/gh-pages/_episodes/04-permissions.md) / Source (https://github.com/swcarpentry/shell-extras/) / Contributing (../contributing/) / Cite (../citation/) / Contact (mailto:lessons@software-carpentry.org)