Linux Refresher

Khaja Ibrahim

Linux Refresher

Table of Contents

Commands, Arguments, and Options

To run a Linux command at the command line, type the command and press Enter. To kill a command in progress, press Ctrl-C.

A simple Linux command consists of a single word, which is usually the name of a program, followed by additional strings called *arguments*. For example, the following command consists of a program name, `ls`, and two arguments:

```
$ ls -l
```

Arguments that begin with a dash, such as `-l`, are called *options* because they change the command's behavior. Other arguments might be filenames, directory names, usernames, hostnames, or any other strings that the program needs. Options usually (but not always) precede the rest of the arguments.

Command options come in various forms, depending on which program you run:

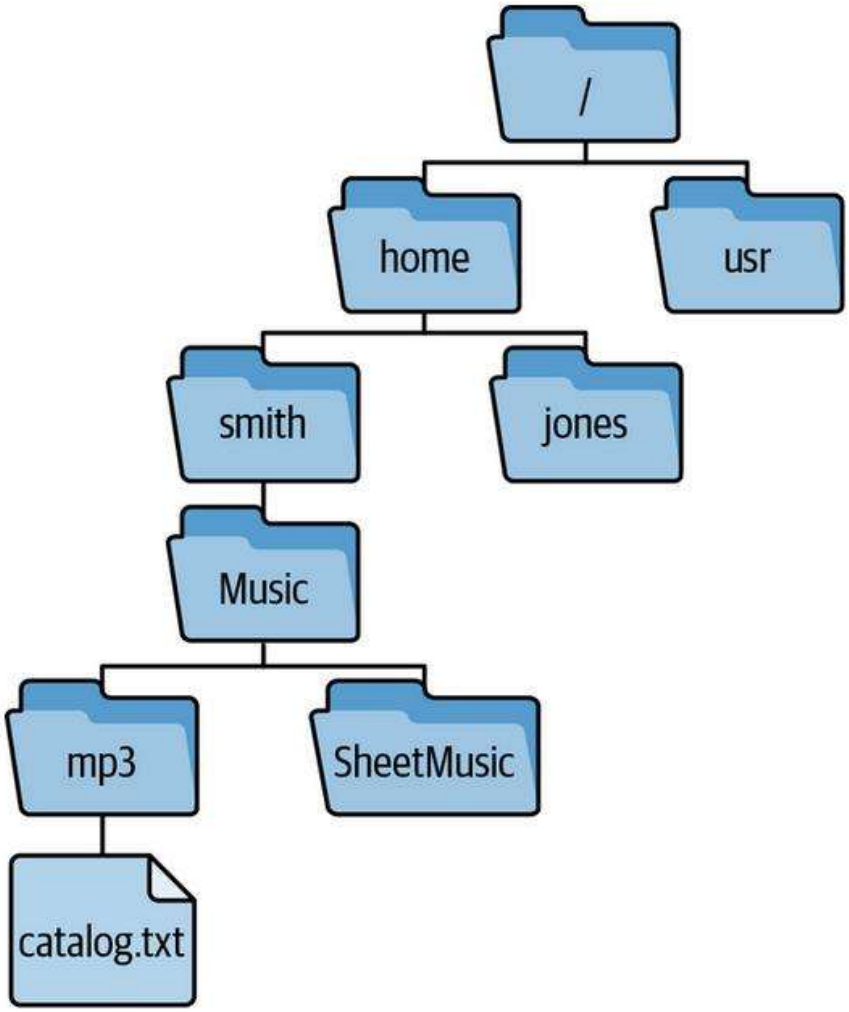
- A single letter, such as `-l`, sometimes followed by a value, as in `-n 10`. Usually the space between the letter and the value can be omitted: `-n10`.
- A word preceded by two dashes, such as `--long`, sometimes followed by a value, as in `--block-size 100`. The space between the option and its value may often be replaced by an equals sign: `--block-size=100`.
- A word preceded by one dash, such as `-type`, optionally followed by a value, as in `-type f`. This option format is rare; one command that uses it is `find`.
- A single letter without a dash. This option format is rare; one command that uses it is `tar`.

Multiple options may often (depending on the command) be combined behind a single dash. For example, the command `ls -al` is equivalent to `ls -a -l`.

Options vary not only in appearance but also in meaning. In the command `ls -l`, the `-l` means "long output," but in the command `wc -l` it means "lines of text." Two programs also might use different options to mean the same thing, such as `-q` for "run quietly" versus `-s` for "run silently." Inconsistencies like these make Linux harder to learn, but you eventually get used to them

The Filesystem, Directories, and Paths

Linux files are contained in directories (folders) that are organized into a tree structure.



The tree begins in a directory called the *root*, denoted by a single forward slash (/), which may contain files and other directories, called *subdirectories*. For example, the directory *Music* has two subdirectories, *mp3* and *SheetMusic*. We call *Music* the parent directory of *mp3* and *SheetMusic*. Directories with the same parent are called *siblings*.

A path through the tree is written as a sequence of directory names separated by forward slashes, such as */home/smith/Music/mp3*. A path may also end with a filename, as in */home/smith/Music/mp3/catalog.txt*. These paths are called *absolute paths* because they begin at the root directory. Paths that begin elsewhere (and don't start with a forward slash) are called *relative paths* because they are relative to the current directory. If your current directory is */home/smith/Music*, then some relative paths are *mp3* (a subdirectory) and *mp3/catalog.txt* (a file). Even a filename by itself, like *catalog.txt*, is a relative path with respect to */home/smith/Music/mp3*.

Two special relative paths are a single dot (.), which refers to the current directory, and two dots in a row (..), which refers to the current directory's parent. Both can be part of larger paths. For example, if your current directory is */home/smith/Music/mp3*, then the path .. refers to *Music*, the path ../../../.. refers to the root directory, and the path *../SheetMusic* refers to a sibling of *mp3*.

You and every other user on a Linux system have a designated directory, called your *home directory*, where you can freely create, edit, and delete files and directories. Its path is usually */home/* followed by your username, such as */home/smith*.

Directory Movement

At any moment, your command line (shell) operates in a given directory, called your *current directory*, *working directory*, or *current working directory*. View the path of your current directory with the `pwd` (print working directory) command:

```
$ pwd
/home/smith
```

Move between directories with the `cd` (change directory) command, supplying the path (absolute or relative) to your desired destination:

```
# Absolute path

$ cd /usr/local

# Relative path leading to
/usr/local/bin
$ cd bin

# Relative path leading to
/usr/local/etc
$ cd ../etc
```

Creating and Editing Files

Edit files with a standard Linux text editor by running any of the following commands:

`emacs`

> Once emacs is running, type Ctrl-h followed by t for a tutorial.

`nano`

> Visit nano-editor.org for documentation.

`vim` or `vi`

> Run the command `vimtutor` for a tutorial.

To create a file, simply provide its name as an argument, and the editor creates it:

```
$ nano newfile.txt
```

Alternatively, create an empty file with the `touch` command, supplying the desired filename as an argument:

```
$ touch funky.txt
$ ls
funky.txt
```

File and Directory Handling

List the files in a directory (by default, your current directory) with the `ls` command:

```
$ ls
animals.txt
```

See attributes of a file or directory with a "long" listing (`ls -l`):

```
$ ls -l
-rw-r--r-- 1 smith smith  325 Jul  3
17:44 animals.txt
```

Left to right, the attributes are the file permissions (`-rw-r--r--`) the owner (`smith`) and group (`smith`), the size in bytes (`325`), the last modification date and time (`Jul 3` of this year at `17:44`), and the filename (`animals.txt`).

By default, `ls` does not print filenames that begin with a dot. To list these files, which are often called *dot files* or *hidden files*, add the `-a` option:

```
$ ls -a
.bashrc    .bash_profile
animals.txt
```

Copy a file with the `cp` command, supplying the original filename and the new filename:

```
$ cp animals.txt beasts.txt
$ ls
animals.txt   beasts.txt
```

Rename a file with the `mv` (move) command, supplying the original filename and the new filename:

```
$ mv beasts.txt creatures.txt
$ ls
animals.txt   creatures.txt
```

Delete a file with the `rm` (remove) command:

```
$ rm creatures.txt
```

Create a directory with `mkdir`, rename it with `mv`, and delete it (if empty) with `rmdir`:

```
$ mkdir testdir
$ ls
animals.txt   testdir
$ mv testdir newname
$ ls
animals.txt   newname
$ rmdir newname
$ ls
animals.txt
```

Copy one or more files (or directories) into a directory:

```
$ touch file1 file2 file3
$ mkdir dir
$ ls
dir   file1   file2   file3
$ cp file1 file2 file3 dir
```

```
$ ls
dir   file1   file2   file3
$ ls dir
file1   file2   file3
$ rm file1 file2 file3
```

Continuing, move one or more files (or directories) into a directory:

```
$ touch thing1 thing2 thing3
$ ls
dir   thing1   thing2   thing3
$ mv thing1 thing2 thing3 dir
$ ls
dir
$ ls dir
file1   file2   file3   thing1
thing2   thing3
```

Delete a directory and all its contents with `rm -rf`. Take care before running this command because it is not reversible.

File Viewing and Permissions

Print a text file on the screen with the `cat` command:

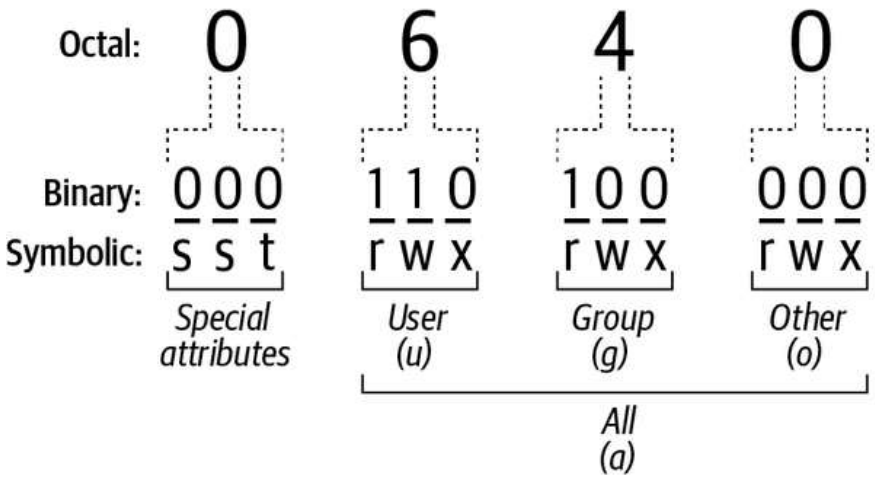
```
$ cat animals.txt
```

View a text file one screenful at a time with the `less` command:

```
$ less animals.txt
```

While running `less`, display the next page by pressing the spacebar. To exit `less`, press q. For help, press h.

The `chmod` command sets a file to be readable, writable, and/or executable by yourself, a given group of users, or everybody.



Here are some common operations with `chmod`. Make a file readable and writable by you, and only readable by everyone else:

```
$ chmod 644 animals.txt
$ ls -l
-rw-r--r-- 1 smith smith  325 Jul  3
17:44 animals.txt
```

Protect it from all other users:

```
$ chmod 600 animals.txt
$ ls -l
-rw------- 1 smith smith  325 Jul  3
17:44 animals.txt
```

Make a directory readable and enterable by everyone, but writable only by you:

```
$ $ mkdir dir
$ chmod 755 dir
$ ls -l
drwxr-xr-x 2 smith smith  4096 Oct  1
12:44 dir
```

Protect a directory from all other users:

```
$ chmod 700 dir
$ ls -l
drwx------ 2 smith smith  4096 Oct  1
12:44 dir
```

Normal permissions don’t apply to the superuser, who can read and write all files and directories on the system.

Processes

When you run a Linux command, it launches one or more Linux *processes*, each with a numeric process ID called a *PID*. See your shell's current processes with the `ps` command:

```
$ ps
   PID TTY          TIME CMD
  5152 pts/11   00:00:00 bash
117280 pts/11   00:00:00 emacs
117273 pts/11   00:00:00 ps
```

or all running processes for all users with:

```
$ ps -uax
```

Kill a process of your own with the `kill` command, supplying the PID as an argument. The superuser (Linux administrator) can kill any user's process.

```
$ kill 117280
[1]+  Exit 15                 emacs
animals.txt
```

Viewing Documentation

The `man` command prints documentation about any standard command on your Linux system. Simply type `man` followed by the name of the command. For example, to view documentation for the `cat` command, run the following:

```
$ man cat
```

The displayed document is known as the command's *manpage*. When someone says "view the manpage for grep," they mean run the command `man grep`.

`man` displays documentation one page at a time using the program `less`,[2] so the standard keystrokes for `less` will work

Keystroke	Action
h	Help—display a list of keystrokes for `less`
spacebar	View the next page
b	View the previous page
Enter	Scroll down one line
<	Jump to the beginning of the document
>	Jump to the end of the document

/	Search forward for text (type the text and press Enter)
?	Search backward for text (type the text and press Enter)
n	Locate the next occurrence of the search text
q	Quit `man`

Shell Scripts

To run a bunch of Linux commands as a unit, follow these steps:

1. Place the commands in a file.
2. Insert a magical first line.
3. Make the file executable with `chmod`.
4. Execute the file.

The file is called a *script* or *shell script*. The magical first line should be the symbols `#!` (pronounced “shebang”) followed by the path to a program that reads and runs the script

```
#!/bin/bash
```

Here is a shell script that says hello and prints today’s date. Lines beginning with `#` are comments:

```
#!/bin/bash
# This is a sample script
echo "Hello there!"
date
```

Use a text editor to store these lines in a file called *howdy*. Then make the file executable by running either of these commands:

```
$ chmod 755 howdy

$ chmod +x howdy
```

and run it:

```
$ ./howdy
Hello there!
Fri Sep 10 17:00:52 EDT 2021
```

The leading dot and slash (`./`) indicate that the script is in your current directory. Without them, the Linux shell won't find the script:

```
$ howdy
howdy: command not found
```

Linux shells provide some programming language features that are useful in scripts. `bash`, for example, provides `if` statements, `for` loops, `while` loops, and other control structures. A few examples are sprinkled throughout the book. See `man bash` for the syntax.

Becoming the Superuser

Some files, directories, and programs are protected from normal users, including you:

```
$ touch /usr/local/avocado
#Try to create a file in a system
#directory
touch: cannot touch
'/usr/local/avocado': Permission
denied
```

"Permission denied" usually means you tried to access protected resources. They are accessible only to the Linux superuser (username `root`). Most Linux systems come with a program called `sudo` (pronounced "soo doo") that lets you become the superuser for the duration of a single command. If you installed Linux yourself, your account is probably set up already to run `sudo`. If you're one user on somebody else's Linux system, you might not have superuser privileges; speak to your system administrator if you're not sure.

Assuming you're set up properly, simply run `sudo`, supplying it with the desired command to run as the superuser. You'll be prompted for your login

password to prove your identity. Supply it correctly, and the command will run with root privileges:

```
$ sudo touch /usr/local/avocado
# Create the file as root
[sudo] password for smith: password
here
$ ls -l /usr/local/avocado
# List the file
-rw-r--r-- 1 root root 0 Sep 10 17:16
avocado
$ sudo rm /usr/local/avocado
```

`sudo` may remember (cache) your passphrase for a while, depending on how `sudo` is configured, so it might not prompt you every time