Using Unix

Xiong-Fei Du

Updated August 18, 2017

1 Introduction

Learning to use the Unix terminal well is probably one of the most important skills to grasp as a beginner programmer. This document will cover some of the most basic ideas – it is not meant to be a comprehensive documentation. Most of what you learn about using terminal will come with experience. This was written for Linux systems using `bash`. In case you aren't using the `bash` shell, most of this is sill applicable to other shells, such as `csh` or `tcsh`, although some of the syntax might differ slightly.

Linux and Mac OS both have very similar terminals. Windows has a command prompt, but it's a bit different.

2 Keyboard Tricks

In most terminals, the up and down keys allow you to visit previous commands so that you don't have to retype them. Additionally, the tab key will allow you to autofill whatever you're beginning to type so that you don't have to type out really long file names. Also, Shift + Insert will allow you to paste text into terminal.

3 Links

In each directory, every subdirectory and file can be referenced by links. Each directory has links to all of its files and subdirectories. Additionally, in any directory, `.` links to the directory itself and `..` link to its parent directory. `/` refers to the root directory, and `~` refers to the home directory of the user.

4 Files and Directories

Being able to work with files and directories is the most important tool in navigating your directory tree in terminal. The most basic commands are:

`$ ls` shows you all non-hidden files and subdirectories of your current directory. Alternatively `$ ls <dir>` will list all non-hidden files and subdirectories of the directory `<dir>`. Additionally, `$ ls -a` will show you all files and directories, including hidden ones. `$ ls -l` will show you details of all files in the directory.

`$ cd <dir>` changes your current directory to `<dir>`. `$ cd ..` will change your current directory to the parent directory. For example, say that you are in the directory `/home/johndoe`. Then typing `$ cd Documents` will change your current directory to `/home/johndoe/Documents`. If you are in the directory `/home/johndoe`, then typing `$ cd ..` will change your current directory to `/home`. Typing `$ cd /` will change your current directory to the root directory of your machine `/`. Typing `$ cd ~` will change your current directory to your home directory, which might look something like `/home/johndoe`.

`$ mv <src> <dst>` moves a file or directory `<src>` to the directory `<dst>`. For example, `$ mv Documents/pic.jpg Pictures` will move the file `pic.jpg` from the `Documents` subdirectory to the `Pictures` subdirectory. Alternatively, say that you're currently in the `Pictures` directory and you're trying to accomplish the same task. Then `$ mv ../Documents/pic.jpg .` will also work. If you want to rename a file or directory, then `$ mv <src> <dst>` renames a file or directory `<src>` to the file or directory `<dst>`.

`$ cp <src> <dst>` copies a file or directory `<src>` to the directory `<dst>`. For example, `$ cp Documents/pic.jpg Pictures` will copy the file `pic.jpg` from the `Documents` directory to the `Pictures` directory. Alternatively, say that you're currently in the `Pictures` directory and you're trying to accomplish the same task. Then `$ cp ../Documents/pic.jpg .` will also work. Note that if you are copying a directory, just typing `$ cp <src> <dst>` will not copy the subdirectories. To do this, you must use the recursive option, `$ cp -r <src> <dst>`.

`$ rm <src>` will remove a file `<src>`. Sometimes, you want to use the force option `$ rm -f <src>` to forcibly remove a file. You can also use the recursive option `$ rm -r <src>` to remove all files in the entire directory `<src>` and any subdirectories. CAUTION: DO NOT ATTEMPT TO USE ANY VERSION OF `$ rm -rf /` – this will permanently damage your operating system.

`$ mkdir <dir>` will make a directory `<dir>` in your current directory.

`$ rmdir <dir>` will remove the empty directory `<dir>` in your current directory.

`$ pwd` (path to working directory) gives the absolute path to your current directory.

5 Working between machines

`ssh` (Secure Shell) allows users to log in to remote servers. `$ ssh user@example.com` will allow you to log in as `user` onto the server `example.com`.

`scp` (Secure Copy) allows users to copy files between their local machine and their remote machine. For example, to copy a local file to a remote directory, `$ scp <src> user@example.com:~/` will copy the file linked at `<src>` to the home directory of `user` at the remote machine found at `example.com`. To copy a remote file to a local machine, `$ scp user@example.com:~/Pictures/pic.jpg <dst>` will copy the file `pic.jpg` found in the `Pictures` subdirectory of the home directory of `user` on the remote server `example.com` to the local directory linked at `<dst>`, which can just be `.` for the current directory.

6 Signals in Shell

Ctrl + C sends the signal `SIGINT` to the foreground process, which usually means terminating the foreground process. For example, if you're running a program and you want to kill the program, Ctrl + C will do so.

Ctrl + D registers an `EOF` (end of file) on standard input, which usually means exiting your current environment. For example, if you are remotely logged in via `ssh`, then Ctrl + D will end your session and log you out. Similarly, if you're in some interpreter, such as `python` for example, then Ctrl + D will exit that interpreter.

7 Using `vim`

`vim` is a very popular command-line text editor that allows you to edit files. I'm not going to get into the `vim` versus `emacs` holy war – this is just to get an overview of the basics in `vim`. Although `vim` can be difficult to learn for those used to word processors like Microsoft Office Word, it can substantially increase your productivity as you get better – the idea of `vim` is so that your hands never leave the keyboard. For that matter, clicking on the terminal won't change your position within the file, unless you edit your `~/.vimrc` file.

The command `vim example.txt` will allow you to open the file linked at `example.txt`. If this file does not already exist, then `vim` will create it.

Now, you are in the normal mode for `vim`. In normal mode, anything you type will be treated as a command and will not change the file itself.

To actually edit the file, pressing the `insert` key, the `a` key, or the `i` key will allow you to enter insert mode, which allows you to insert characters into the file. You can press any of those keys again to enter replace mode, where anything you type will replace the character at your current position. The `esc` key will allow you to exit insert mode and return to normal mode.

Now back in normal mode, you can type the `v` key to enter visual mode, which allows you to highlight text so that you can do stuff like copy-paste and delete. In normal mode, hitting Shift + v allows you to enter visual block mode, which allows you to select blocks of text. This is good if you want to highlight entire lines of code. In visual mode, `c` will cut what you've selected and enter insert mode. `y` will copy the text and exit visual mode. In normal mode, `p` will paste it. Again, the `esc` key will allow you to exit visual mode and return to normal mode.

In normal mode, you can also type `:` to enter the command line to give a command. The most commonly used ones are the following. `:w` writes/saves your file without quitting. `:q!` quits `vim` without saving. `:wq` writes your file and then quits. `:37` will jump to line 37. There are tons of other commands out there.

The settings for `vim` can be found in the `.vimrc` file in the home directory. You can change these settings by using `$ vim ~/.vimrc` (note the irony of using `vim` to change the `vim` settings).

There are tons of other cool features that can be used in `vim` that I'm not going to get into. To learn more about `vim`, just give the command `$ vimtutor`.

In case you hate `vim`, there are other text editors out there too, such as `emacs`, Sublime Text, and `nano`.

8 Executables

Executables are simply the programs that you run in Unix. Everything that you can run is an executable, whether it's a compiler, an interpreter, a bash script, or your own compiled program.

To run an executable, simply type the command to the link of the executable. For example, if you compiled an executable `a.out` in your current directory, then run it by simply typing `$ ./a.out` in the command line.

However, this can cause issues sometimes since we would have to give the full path to link files in different directories. For example, let's just say that we want the Python interpreter to interpret the file `example.py` and our Python interpreter is installed in a system location such as `/example/path/to/bin/python`. In the directory that contains `example.py`, we don't want to give the command `$ /example/path/to/bin/python example.py` every time we want interpret our Python script – we want to be able to just type `$ python example.py`. To do this, we can change our `.bashrc` file located in the home directory by adding the line `export PATH=$PATH:/example/path/to/bin`. Then we can run the command `$ source ~/.bashrc`, and then, the command `$ python example.py` will work.

9 Permissions

When you run the command `$ ls -l`, the first column contains the permissions of each file. The first character will be `d` if the link is a directory, otherwise it's a file. The next three characters are the owner's permissions, the following three characters are the group's permissions, and the next three characters are the general user's permissions. `r` is for read, `w` is for write, and `x` is for execute.

A common problem that you'll likely run into when you're running a `bash` script or something else is that you'll get a permission denied error. That's because you did not set that script to be executable. To change this, just give the command `$ chmod +x <src>` to make the file `<src>` executable.

`sudo` allows the user to use the security permissions of another user, usually the administrator. However, depending on your user settings, you might not be able to use `sudo`, especially on work, school, or government computers.

10 Regular Expressions

The Unix terminal also recognizes regular expressions. The most basic and most useful example is when you want to copy all files in your current directory to a directory `<dst>`, you can just do `$ cp * <dst>`. Or let's say that you're trying to remove all of the files that end in `.c` in your current directory, you could do `$ rm *.c`.

11 `tar`

`tar` files allow you to create an archive of a set of files and directories into one file and transfer them easily. The `cf` options allow you to archive the files. For example, if you want to archive all `.c` files in your current directory, the file `readme.txt` in your current directory, and all of your `.pdf` files in your `example` subdirectory into a file `foo.tar`, you can give the command `$ tar -cf foo.tar`

`*.c readme.txt example/*.pdf`. To do it the other way around, say that you want to extract everything in `foo.tar`. Then you can give the command `$ tar -xf foo.tar`. The `c` option creates the archive, the `x` option extracts the archive, and the `f` option specifies the file. You can also add the `v` (verbose) option, which most people do in practice, so that you can see what it's doing.

Additionally, most people also choose to add the `z` option to compress their `tar` archive. Using the same example, if you want to compress all `.c` files in your current directory, the file `readme.txt` in your current directory, and all of your `.pdf` files in your `example` subdirectory into a file `foo.tgz` while also printing out what exactly it's doing, you can give the command `$ tar -czvf foo.tgz *.c readme.txt example/*.pdf`. To do it the other way around, to extract everything in `foo.tgz`, you can give the command `$ tar -xzvf foo.tgz`. It's standard practice to use the file extension `.tgz` or `.tar.gz` to distinguish a compressed archive from an uncompressed archive `.tar`.

It's also worth noting that `tar` archives also preserve the structure of the directories and subdirectories. This can be useful if you just want to create a tarball of everything in a directory. For example, if you want to `tar` your entire `Documents` subdirectory, then you can give the command `$ tar -czvf foo.tgz Documents`. Then you can move the file `foo.tgz` around, and when you give the command `$ tar -xzvf foo.tgz`, it will extract the tarball `foo.tgz` and create an exact copy of your `Documents` directory within your current directory.

12 `make`

Makefiles are a simpler alternative to shell scripts that are usually used to create tarballs or compile executables. Makefiles are always simply named `Makefile`. You can only have one makefile in a directory.

A very simple makefile might look like this:

```
CC=gcc

all: bar

bar:
    $(CC) -o bar foo1.c foo2.c

clean:
    rm bar

blah.tgz:
    tar -czvf blah.tgz bar *.c
```

In this example, say we are working on files `foo1.c` and `foo2.c` in the same directory that contains this makefile. Then the command `$ make` or `$ make bar` will have the same effect as the command `$ gcc -o bar foo1.c foo2.c`. The command `$ make clean` is usually coded to undo what `$ make` does, and in this case, `$ make clean` will do the same thing as the command `$ rm bar`. Additionally `$ make blah.tgz` will have the same effect as the command `$ tar -czvf blah.tgz bar *.c`. Generally speaking, most people by convention name the make targets as the names of the file that target will produce. This is because `make` is smart enough to recognize if any relevant files have been changed when this is done. If the affected files were not changed since the last time you ran `make`, then `make` will skip that step. This feature can save a lot of time when recompiling large programs. There are also other fancy things that makefiles can do. Note that you always use a tab rather than 4 spaces for indentation in makefiles.

13 Variables and Programming in Shell

`bash` also allows you to set variables. For example, you can give the command `$ x=5` or `$ y="foo bar"`. Then if you give the command `$ echo $x` or `$ echo $y`, you can see what is stored in the variables `x` and `y`. Variables can be referenced by `$x`, if you forget the `$` sign, then you'll just be referencing the string `x`. The `$ echo <arg>` command is basically just a print command in `bash`. Important variables set by default include `HOME`, which references your home directory, `SHELL`, which references where and what your shell is, and `PATH`, which has a list of links to places where your shell will look for executables.

Also, you want to avoid putting spaces in strings, but in case you do, you can always use the escape character \ right before you enter a space, or you can put the entire string in either double or single quotes, e.g. `foo\ bar` or `"foo bar"`. Otherwise, `bash` will treat it as separate strings.

Also, `bash` is technically Turing-complete, which means that you can write any computable program in `bash` – but why would anyone ever want to do that...