Linux at Olin

Allen B. Downey

Linux at Olin

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Chapter 1

1.1 About this book

This book is free. Permission is granted to reproduce, store or transmit the text of this book by any means, electrical, mechanical, or biological, in accordance with the terms of the GNU Free Documentation License as published by the Free Software Foundation.

This book is written for people who have never used Linux before, but I will assume that you have some experience with another operating system, either Windows or Macintosh.

For now, this book should be considered a rough draft. Some of the things I am writing about are changing even as I write, so I'm sure there will be things that are different by the time you read this. And there are probably things I will get wrong.

If you find this book useful (and maybe especially if you don't), help me make it better. Let me know if you find errors, if you have suggestions for clarification, or if you think I've left out something important.

One other thought: like most technology, Linux can be frustrating. There are a lot of things about it that I like, but there are plenty of other things that are just plain stupid. If you find yourself starting to hate it, you might enjoy reading *The Unix-Hater's Handbook* at `http://www.cs.washington.edu/homes/weise/unix-haters.html`.

But, to paraphrase Winston Churchill, Linux is the worst operating system, except for all those others that have been tried.

1.2 What is Ubuntu?

Ubuntu is one of many distributions of Linux. A **distribution** is a collection of software that includes the Linux kernel (the core of the operating system) along with lots of applications. Many of the core applications were created by the GNU Project, which is why many people think that Linux should rightfully be called GNU/Linux (see `http://www.gnu.org/gnu/linux-and-gnu.html`).

Other distributions of Linux include Fedora, Debian and MINT. Distributions have different pros and cons. Some people at Olin choose to run other distributions. Unfortunately, it is hard to provide support for multiple distributions, so people who run other distributions are mostly "swimming at their own risk."

Ubuntu is just one distribution of Linux, and Linux is just one implementation of Unix. Unix is a standard that describes the interface to the operating system, which includes

- A set of operating system features (like abstract processes),
- A set of libraries and system calls,
- A set of core applications,
- Program development tools, compilers and interpreters,
- A command-line interface, and
- A window system.

Strictly speaking, the Unix standard only defines the first two items on this list, but when people talk about Unix systems, they are usually referring to systems that provide all of these items. Other implementations of Unix include Solaris, Irix and HP-UX.

Historically, Unix implementations were created by hardware companies, and they were hardware-specific; Solaris only ran on Suns, Irix only ran on SGI machines, and HP-UX ran on Hewlett-Packard machines. For a long time, Unix was synonymous with expensive hardware.

Linux was one of the first implementations of UNIX for the x86 architecture, which is what most cheap PCs are based on. Now it runs on lots of processors.

1.3 Terminals and CLIs

Personally, I don't like GUIs. I think they are slow, and although they make common operations easy, they tend to make rare but important operations impossible.

When I use a Linux machine, the first thing I do is create a terminal window, which provides a command-line interface; that is, I type commands and the computer executes them.

To open a terminal, search for and launch Terminal. A window should appear. In the window you should see a line of text like this:

```
downey@ubuntu:~$
```

For you the information in brackets will be different. This is a prompt; it is waiting for you to type a command.

If you type the command `whoami`, Linux will tell you the username you are logged in as. `date` prints the current time and date. In order to do more interesting things, we need some files.

1.4 Files and directories

A **file** is a document stored on disk. A **directory** is a named collection of files, also sometimes called a folder. Again, there are two ways to work with files and directories, using the GUI or the CLI. The GUI is tempting because it is easy to get started and it might be familiar if you have used other operating systems.

But the real power of Unix systems is in the command-line interface, and the sooner you learn it, the better. I suggest that you waste no time with the GUI, and start immediately with the CLI.

When you create a terminal, you will normally start out in your home directory; the name of your home directory is the same as your user name. That's why your user name appears twice in the prompt; the first one indicates who you are logged in as, the second indicates what directory you are in.

Directories are nested one inside the other. On many Unix systems your home directory is contained in a directory named `home` which is contained in the top-level directory, which is named `/`.

The **path name** for a directory specifies the complete path from `/` on down. The path name for my home directory is `/home/downey`. To see your path, type `pwd`, which stands for "print working directory", that is, the directory you are currently working in.

```
$ pwd
/home/downey
```

1.5 Listing files

To list the contents of the current directory, type `ls` and hit return. On a new install, your home directory looks something like this:

```
$ ls
Desktop         Music          Templates
Documents       Pictures       Videos
Downloads       Public
```

These are all directories Ubuntu created for you to put stuff in.

Try this:

```
$ ls /
bin   dev  home    lib         misc  opt   root  selinux  tmp  var
boot  etc  initrd  lost+found  mnt   proc  sbin  sys      usr
```

This is the contents of the top-level directory `/`. Notice that `home` is one of the entries. Most of these weird abbreviations are Unix standards; for example, on most systems `/tmp` is used for temporary files, `/etc` contains system configuration files, and `/bin` contains binary files (executable binaries, to be more precise).

Try `ls /bin`. The files you see are all executable commands. Notice that `ls` is one of them! When you type `ls`, the program that gets executed is `/bin/ls`. In other words, `ls` is just an application like any other.

The output from ls is color-coded. Directories are blue. Most executable files are green. Cyan files are links, which are pointers to other files (you can ignore that for now). The red files are "set-uid root" files, which means that when they execute, they run as root! For example, `ping` is frequently used to check whether a machine on the network is running. Try this:

```
$ ping google.com
```

Ping is a good way to check whether your network connection is working. If you are connected to the network, you should see something like this:

```
PING google.com (72.14.204.99) 56(84) bytes of data.
64 bytes from iad04s01-in-f99.1e100.net (72.14.204.99): icmp_seq=1 ttl=48 time=14.9 ms
64 bytes from iad04s01-in-f99.1e100.net (72.14.204.99): icmp_seq=2 ttl=48 time=14.7 ms
```

It will keep going, printing one line per second, until you stop it by pressing Control-c. You can stop most running programs by typing Control-c.

1.6 Commands and arguments

Most Unix commands are short, lower-case, and arcane. At first, they will seem weird, but once you are familiar with them, you will appreciate their brevity, because you will be able to type them very fast.

Most Unix commands are verbs, and many of them take a direct object, which is sometimes called an argument. For example, the object of `ls` is the directory you want to list. If you omit the object, `ls` performs its default behavior, which is to list the working directory.

`cd` stands for change directory. Try this:

```
$ cd /etc
$ pwd
/etc
$ cd network
$ pwd
/etc/network
$ cd
$ pwd
/home/downey
```

In the first case, the argument for `cd` is a complete path (it begins with `/`), so it takes us to `/etc` which contains lots of files (and other directories) that control system settings.

In the second case, the argument is a directory that is in the current directory. It is called a **relative path** because it is relative to the current directory. So it takes us to `/etc/network`.

In the third case, there is no argument, so `cd` performs its default behavior, which is to go back to your home directory.

Here are two more examples to try:

```
$ cd ~
$ pwd
/home/downey
$ cd ~downey
$ pwd
/home/downey
```

The tilde (`~`) is a nickname for your home directory, and `~username` is a nickname for username's directory.

1.7 User and Superuser

Linux is a multi-user operating system, which means that each user has an account that keeps track of:

- The files (and directories) that belong to each user, and
- The settings and preferences for each user.

In addition to the user accounts, each machine has one **superuser** account, which is used for system administration.

Regular users are limited; usually they can only read their own files, and they cannot modify files that belong to other users or files that are part of the system.

But the superuser can do anything! To run a command as superuser, use `sudo`. For example, if you try to install a new package as yourself:

```
$ apt-get install emacs
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
```

But if you try

```
$ sudo apt-get install emacs
```

It should prompt you for your password and then install emacs.

1.8 Editing files

One of several popular text editors for Unix is `emacs`, which stands for "Editor MACroS", but some people think it stands for "Escape Meta Alt Control Shift", because `emacs` has some pretty elaborate keyboard shortcuts. The first version of `emacs` was written in 1975 by Richard Stallman.

To install emacs, you might have to run `sudo apt-get install emacs`, unless you did it already.

To launch emacs, type `emacs` in a terminal. An emacs window will appear. Notice that while the emacs window is running, the terminal where you started it is inactive. That can be annoying, so I'll show you how to avoid it.

First, exit emacs by selecting Exit from the File menu, or use the keyboard shortcut Control-X Control-C. Emacs and its users often abbreviate keyboard shortcuts by writing C for Control and M for Meta (which is the Alt key on your keyboard), so this shortcut may appear as C-x C-c.

Now start emacs again by typing `emacs &`. The ampersand says you want to run emacs **in the background**, which allows the terminal to keep running in the foreground (metaphorically speaking).

Now start typing in the `emacs` window. If you write more than one line of text, you will notice a difference between `emacs` and most word processors. It doesn't wrap words at the end of a line. When you are writing a program, that's probably a good thing, but if you are writing a paper, it's not as good. If you want word wrap, you can find it in the Options menu.

To save your file, select File→Save As... or use the keyboard shortcut C-x C-w (Control-x Control-w). Instead of popping up a window, `emacs` moves the cursor to the **minibuffer** at the bottom of the window. It fills in the path of the current directory for you, which is probably `~/`. You just have to provide a filename. Name your file `example.txt`.

By the way, **buffer** is the term emacs uses to refer to the text you are editing in `emacs`. The name is intended to remind you that you are not really editing the file; the changes that you make are not saved until you write the contents of the buffer into a file. Other operating systems work the same way, but they tend to blur the distinction between the document you are editing and the document that is saved on disk.

Now exit `emacs`.

1.9 Practice

Try the following to see how much of this is making sense.

- Move back to your home directory and confirm that you are there.
- List the contents of `/var` without changing directories.
- List the contents of `/var/log` without changing directories.
- Change the working directory to `/var` and confirm that you are there.

- List the contents of `/var` with the minimum number of keystrokes (2 plus return).
- List the contents of `/var/log` with the minimum number of keystrokes (6 plus return).
- Move down into `log`.
- Print the “tail” of the file named `messages` by typing `tail messages`. You should get something like:

  ```
  tail: cannot open ‘messages’ for reading: Permission denied
  ```

  Why is permission denied? Because only root is allowed to read this file, which contains a log of system messages.
- Try `sudo tail messages`.

Congratulations, you are now able to understand this joke: `http://xkcd.com/149`.

Chapter 2

2.1 Free as in Freedom

GNU/Linux is Free Software. That's free as in free speech, not necessarily free as in free beer. The fundamental idea behind Free Software is that users have a right to read, modify and contribute to the software they use.

One of the earliest proponents of Free Software was (and still is) Richard Stallman, who has been in residence (literally) at the MIT AI Lab since the Jurassic era. Stallman's primary argument for Free Software is that it is morally wrong to deprive users of the rights that they are entitled to. But this is a moral argument, so it has a limited ability to persuade people who don't accept its premise.

The Open Source movement is based on a similar set of ideas, but it tends to emphasize the practical advantages of Free Software, including features, adaptability, reliability, security, and cost. This argument has the potential to be persuasive, but it depends on an empirical question. Is Free Software actually less expensive than proprietary software (including all costs of operation)? Is it really more reliable?

The argument for Free Software that I find most persuasive is analogous to the argument for academic freedom. Academic institutions are based on the assumption that creating knowledge is a public enterprise; since we depend on public goods (the accumulated knowledge published by our predecessors), we have an obligation to produce public goods. This is partly a moral argument, since it would be wrong to take advantage of the generosity of others without making a reciprocal contribution. It is also a practical argument. If we stop sharing knowledge, we undermine our ability to create new knowledge.

Similarly, all new software is built on a foundation of previous software, and a big part of that foundation is Free Software. The majority of applications that people recognize, and the protocols that underlie them, were developed at universities and public research labs, and made available as Free Software, before they were commercial products. These include Web browsers and servers, search engines, email, word processors, Internet protocols including TCP/IP, and standards and applications for images, sound and movies, and public-key cryptography.

One of my favorite quotes sums up this idea:

> "As we enjoy great Advantages from the Inventions of others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously."
>
> —Benjamin Franklin

2.2 `cat` and `more`

If you have been following directions, you should be back in your home directory, and there should be a file there named `example.txt`. The suffix `txt` indicates that this is a **plain text** file, which means that it contains a sequence of characters with no formatting or style information.

Use `pwd` to confirm that you are in your home directory (or just look at the prompt, or the window title!), and use `ls` to see if your file is there.

```
$ ls example.txt
example.txt
```

To see the contents of the file, you can use `cat`:

```
$ cat example.txt
```

The contents of the file should appear on the screen.

Let's get our hands on something a little more interesting. Type:

```
$ wget http://www.gutenberg.org/files/159/159.txt
```

`wget` is an application that downloads files from the Internet; in this case it is getting a file from the Gutenberg Project, which provides books in plain text format.

To see the contents of the file you just downloaded, type `cat 159.txt`. Whoa! Congratulations, you just speed-read *The Island of Doctor Moreau*, by H. G. Wells.

If, for some reason, you can't read that fast, you might want to use `more`, which displays the contents of a file a little at a time, and waits for you to press a key to get more.

Type `more 159` and then press the Tab key. Unix fills in the rest of the file name for you. This is called **file name completion** and it is a very useful feature. One of the big drawbacks of the command-line interface is that you have to do more typing. File name completion provides at least some relief.

Press Return to run `more`. Each time you press a key, you get the next page of text. If you don't want to read the whole book, you can press `q` or Control-C to quit.

2.3 Text processing

Books from Project Gutenberg are in plain text. The nice thing about plain text is that it can be read by practically any application.

For example, `wc` is a small application that counts the number of lines, words, and characters in a file. Type `wc 159` and then press Tab to complete the file name and Return to run the command. You should get something like this:

```
  5333  46634 267688 159.txt
```

There are 5333 lines, 46634 words, and 267688 characters in this file. The output from Unix commands tends to be terse, because often the output from one command becomes the input to another.

2.4 `grep`

`grep` is an application that searches a file for a certain word (or pattern), and prints all the lines that match. For example, to see all the lines that contain the word "Moreau", type:

```
$ grep Moreau 159.txt
```

Don't forget to use file name completion to save typing. Again, the output might take more than one screen, so you can take the output from `grep` and **pipe** it to `more`, using the "pipe" character `|`. Here's what the command looks like:

```
$ grep Moreau 159.txt | more
```

But don't type the whole thing! Instead, press the Up Arrow key. You should see the previous command. You can press the Up and Down keys to scroll through your command history; then you can edit a command and run it again.

So, find the previous `grep` command and add on the `| more`, then hit return. You should be able to page through the lines that mention Moreau.

Now use the arrow keys to scroll up and edit the `grep` command and search for lines that contain the word "fruit". Does it appear in the book? How about "apple"?

If you want to search for a phrase that contains spaces, you have to use quotation marks:

```
$ grep "Are we not Men" 159.txt
```

One limitation of this approach is that if you are searching for a phrase, and the phrase is broken between lines, `grep` won't find it.

As an exercise, use `grep` and `wc` to count the number of times Montgomery is mentioned.

2.5 Redirection

Most Linux commands display results on the screen. You can use a pipe to redirect the output to another process or you can use `>` to redirect the output into a file:

```
$ grep "Are we not Men" 159.txt > lines.txt
$ wc lines.txt
  8 110 525 lines.txt
```

This example stores the output from `grep` in a file called `lines.txt` and then uses `wc` to count the lines.

2.6 `sed`

As another example of plain text processing, let's replace every instance of the word "Moreau" with the word "Downey". The application `sed` does just that:

```
$ sed s/Moreau/Downey/ 159.txt
```

The second argument tells `sed` to substitute (hence the "s") "Moreau" with "Downey". Here are a few lines from *The Island of Doctor Downey*:

> "Downey!" I heard him call, and for the moment I do not think I noticed. Then as I handled the books on the shelf it came up in consciousness: Where had I heard the name of Downey before? I sat down before the window, took out the biscuits that still remained to me, and ate them with an excellent appetite. Downey!

Mmm. Biscuits.

Note that `sed` displays results on the screen, but it doesn't modify the original file. To save the results, you can redirect them to a file. You can also pipe the results to another process.

You can even pipe the output from `sed` back into `sed` to make another substitution. As an exercise, use `sed` to change "Moreau" to "Downey" and "Montgomery" to your name, then redirect the result into a file.

2.7 The power of plain text

There are thousands (or more) of Unix applications that read and write plain text. Depending on what you want to do, you can use an existing application, or modify one, or write a new one.

By piping the output from one file into another, you can often perform complex computations without having to write a program.

You can read an interview with Andy Hunt and Dave Thomas at `http://www.artima.com/intv/plain.html` where they talk about the advantages (and disadvantages) of plain text. Here is an excerpt:

> Dave Thomas: The problem is, once we store data in a non-transparent, inaccessible format, then we need code to read it, and that code disappears. Code is disappearing all the time. You probably can't go to a store and ask for a copy of Word 1, or whatever the first version of Word was called. So we are losing vast quantities of information, because we can no longer read the files.
>
> ...
>
> Another reason for using plain text is it allows you to write individual chunks of code that cooperate with each other. One of the classic examples of this is the Unix toolset: a set of small sharp tools that you can join together. You join them by feeding the plain text output of one into the plain text input of the next. There's no concept of trying to make sure the word count program outputs things in a format that's compatible with the next tool in the chain. It's just plain text to plain text, and that's a very powerful way to do it.

2.8 Making directories

Within your home directory, it is often a good idea to create subdirectories that contain related files. `mkdir` is the command that creates new directories. `mkdir sd` creates a new directory (in the current directory) named `sd`. If you type `ls` you should see it.

You can `cd` into the directory in the usual way. Try this:

```
$ cd sd
$ pwd
/home/downey/sd
$ cd ..
```

What directory do you end up in? `..` is a special file name that refers to the **parent** of the current directory; that is, the directory that contains the current directory.

`rmdir` is the command that removes directories. Type `rmdir sd` to remove the directory you just created. As a precaution, `rmdir` will only let you remove a directory that is empty.

2.9 (Re)moving files

If you want to change the name of a file, you have to "move" the file with `mv`. It works pretty much as you'd expect:

```
$ mkdir sd
$ mv sd eng2510
```

The directory formerly known as `sd` is now `eng2510`. If you want to get rid of a file altogether, you can remove it with `rm`.

```
$ rm example.txt
```

Poof! No more `example.txt`. This file is not just gone, it is really, really gone. I mean gone, gone. Not coming back gone. What I'm saying is... you get the idea.

The point is that `rm` is unforgiving. Use it very carefully.

WARNING: one of the easiest ways to lose data in UNIX is by clobbering an existing file with `mv`. For example, try the following

```
$ touch a.txt
$ touch b.txt
$ ls
```

`touch` creates a new, empty file (among other things). Now run

```
$ mv a.txt b.txt
$ ls
```

Whatever was in `a.txt` is now in `b.txt`, and whatever was in `b.txt` is now gone. I mean gone, gone. Not coming back gone. What I'm saying is...

Be careful with `mv`.

2.10 `cp`

`cp` is the command that copies files. `cp` takes two arguments, the name of the source file, and the destination, which can be either a file name or a directory name.

For example,

```
$ cp .bashrc .bashrc.old
```

makes a copy of `.bashrc` named `bashrc.old`. If you are about to edit a configuration file, and you want to save the old one, you might do something like that.

If the second argument is a directory, then the copy will go into the destination directory, and have the same name as the original.

```
$ mkdir temp
$ cp 159.txt temp
```

This example create a new directory and then put a copy of `159.txt` into it (assuming that `159.txt` is in the current directory).

To copy something from another directory into the current directory, you can use `.` as the destination. For example,

```
$ cp /etc/passwd .
```

makes a copy of `/etc/passwd` in the current directory. You might think its funny that you can read a file named `passwd`, but it doesn't actually contain passwords. Take a look; it contains some interesting stuff (especially the last line).

2.11 Customizing bash

When you type a command in a terminal, the program that executes the command is called a **shell**, because it acts as a layer that covers the raw operating system.

bash is one of several shells that provide a similar set of capabilities. Some of the features you have seen in this chapter, like file name completion and command history editing, are provided by bash.

Many of the properties of bash can be customized by editing a special configuration file called .bashrc. Configuration files in Unix often end in rc, and many of them begin with a dot (.), because file names that begin with a dot are invisible!

Go to your home directory and type ls -a. The -a option means that you want *all* the files, including the invisible ones. You should see configuration files for emacs, gnome, bash, and other applications.

To edit your bash configuration, type emacs .bashrc &. You should see something like this:

```
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
```

This is a minimal config file provided by the system. The lines that begin with hashes (#) are comments; they don't do anything. The lines that begin with if and end with fi read the contents of the file /etc/bashrc, which contains general configuration information for everyone.

If you want, you can look at the contents of /etc/bashrc, but you can't edit it, because any changes you made would affect all users. But you can add lines to your .bashrc to add to or override the properties that got set in in /etc/bashrc.

One of the things /etc/bashrc does is set the prompt, as we discussed in Section 1.3. I prefer a simple prompt that doesn't take up much room, so I have the following line in my .bashrc:

```
# User specific aliases and functions
export PS1="$ "
```

PS1 is the name of the variable that controls the prompt, and we have to export it so that it has the effect we want. The right-hand side of the equals sign is the new prompt, in this case just a dollar sign and a space.

You can put this line pretty much anywhere in your .bashrc, but it's conventional to put it after the comment about "aliases and functions".

After you change your `.bashrc`, all new terminals will see the change, but if you have terminals already running, you have to tell them to read and execute `.bashrc` again. Here's how:

```
$ . .bashrc
```

After that, you should see your new prompt.

For some cool ideas about what you can put in your prompt, check out `http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/`, especially Section 2.

2.12 Aliases

An **alias** is a nickname for a command. For example, the application that reads PDF files is called `evince`, but that's too long, so I added the following line to my `.bashrc`:

```
alias ev=evince
```

Now I can type `ev filename.pdf` and the shell knows that I want to run `evince`.

While we are at it, this is probably a good time to take some of the bite out of `rm`. If you give `rm` the `-i` option, it asks before it deletes files. It's annoying, but it reduces the chance that you will delete something important by accident.

If you want to use the `-i` option all the time, you can create an **alias** so that whenever you type `rm`, it will always do `rm -i`. Add the following line to `.bashrc`:

```
alias rm="rm -i"
```

Now open a new terminal, or tell an old terminal to read `.bashrc` again, and then try this:

```
$ touch temp.txt
$ rm temp.txt
rm: remove regular empty file ‘temp.txt’? y
```

If you want to delete something without being interrogated, you can use the `-f` option, which stands for "force".

As an exercise, add an alias to your `.bashrc` so that when you type backup, it back up your home directory to your User Share on fsvs01 (see `http://wikis.olin.edu/linux/doku.php?id=backup_and_restore`).