Linux

Nguyễn Quản Bá Hồng[1]

November 1, 2022

[1]Independent Researcher, Ben Tre City, Vietnam
e-mail: `nguyenquanbahong@gmail.com`; website: `https://nqbh.github.io`.

Contents

I have used the following Linux distributions: UBUNTU, KUBUNTU, SUSE, OPENSUSE, where the 1st one is install in my personal notebooks/laptops, the 2nd one is install in my WIAS Dell XPS 15 (KDE provides a lot of good stuffs), while the other ones are install in computer servers at WIAS Berlin. Note that SUSE & OPENSUSE are produced by Germans[1].

Question 0.1. *Which resources I should use to learn deeply Linux in a fast but complete way?*

William Shotts. *The Linux Command Line: A Complete Introduction* (2e) seems a good start.

Question 0.2. *Which e-book readers should I use on Linux?*

[1]In my own experiences, I believe that Germans & Frenches prefer building their own facilities over using those provided by the rest of the world.

Chapter 1

Shotts, 2019. The Linux Command Line: A Complete Introduction. 2e

About the Author. "William Shotts has been a software professional for > 30 years & an avid Linux user for > 20 years. He has an extensive background in software development, including technical support, quality assurance, & documentation. He is also the creator of `LinuxCommand.org`, a Linux education & advocacy site featuring news, reviews, & extensive support for using the Linux command line." – Shotts, 2019, p. 6.
About the Technical Reviewer. "Jordi Gutiérrez Hermoso is a coder, mathematician, & hacker-errant. He runs Debian GNU/Linux exclusively since 2022, both at home & at work. Jordi has been involved with GNU Octave, a free numerical computing environment largely compatible with MATLAB, & with Mercurial, a distributed version control system. He enjoys pure & applied mathematics, skating, swimming, & knitting. Nowadays he thinks a lot about environmental mapping, greenhouse gas emissions, & rhino conservation efforts." – Shotts, 2019, p. 7

Introduction

"I want to tell you a story. No, not the story of how, in 1991, Linus Torvalds wrote the 1st version of the Linux kernel. You can read that story in lots of Linux books. Nor am I going to tell you the story of how, some years earlier, Richard Stallman began the GNU Project to create a free Unix-like operating system. That's an important story too, but most other Linux books have that one, as well.

No, I want to tell you $\boxed{\text{the story of how you take back control of your computer}}$.

When I began working with computers as a college student in the late 1970s, there was a revolution going on. The invention of the microprocessor[1] had made it possible for ordinary people like you & me to actually own a computer. It's hard for many people today to imagine what the world was like when only big business & big government ran all the computers. Let's just say, you couldn't get much done.

Today, the world is very different. Computers are everywhere, from tiny wristwatches to giant data centers to everything in between. In addition to ubiquitous[2] computers, we also have a ubiquitous network connecting them together. This has created a wondrous[3] new age of personal empowerment[4] & $\boxed{\text{creative freedom}}$, but over the last couple of decades something else has been happening. A few giant corporations have been imposing their control over most of the world's computers & deciding what you can & cannot do with them. Fortunately, people from all over the world are doing something about it. They are fighting to maintain control of their computers by writing their own software. They are building Linux.

Many people speak of "freedom" with regard to Linux, but I don't think most people know what this freedom really means. Freedom is the power to decide what your computer does, & the only way to have this freedom is to know what your computer is doing. Freedom is a computer that is without secrets, one where everything can be known if you care enough to find out." – Shotts, 2019, pp. 30–31

Why Use the Command Line?

"Have you ever noticed in the movies when the "superhacker" – you know, the guy who can break into the ultra-secure military computer in < 30 seconds – sits down at the computer, he never touches a mouse? It's because filmmakers realize

[1]**microprocessor** [n] (*computing*) a small unit of a computer that contains all the functions of the central processing unit.
[2]**ubiquitous** [a] present, appearing or found everywhere.
[3]**wondrous** [a] (*literary*) strange, beautiful, & impressive, SYNONYM: **wonderful**.
[4]**empowerment** [n] [uncountable] a positive feeling that you have some control over your life or the situation you are in.

that we, as human beings, instinctively know the only way to really get anything done on a computer is by typing on a keyboard!

Most computer users today are familiar only with the *graphical user interface* (GUI) & have been taught by vendors[5] & pundits[6] that the *command line interface* (*CLI*) is a terrifying thing of the past. This is unfortunate because a good command line interface is a marvelously[7] expressive way of communicating with a computer in much the same way the written word is for human beings. It's been said that "graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible," & this is still very true today.

Since Linux is modeled after the Unix family of operating systems, it shares the same rich heritage of command line tools as Unix. Unix came into prominence[8] during the early 1980s (although it was 1st developed a decade earlier), before the widespread adoption of the graphical user interface &, as a result, developed an extensive command line interface instead. In fact, 1 of the strongest reasons early adopters of Linux chose it over, say, Windows NT was the powerful command line interface that made the "difficult tasks possible." – Shotts, 2019, p. 31

What This Book Is About

"This book is a broad overview of "living" on the Linux command line. Unlike some books that concentrate on just a single program, such as the shell program `bash`, this book will try to convey how to get along with the command line interface in a larger sense. How does it all work? What can it do? What's the best way to use it?

This is not a book about Linux system administration. While any serious discussion of the command line will invariably lead to system administration topics, this book touches on only a few administration issues. It will, however, prepare the reader for additional study by providing a solid foundation in the use of the command line, an essential tool for any serious system administration task.

This book is Linux-centric. Many other books try to broaden their appeal by including other platforms such as generic Unix & macOS. In doing so, they "water down" their content to feature only general topics. This book, on the other hand, covers only contemporary Linux distributions. 95% of the content is useful for users of other Unix-like systems, but this book is highly targeted at the modern Linux command line user." – Shotts, 2019, p. 32

Who Should Read This Book

"This book is for new Linux users who have migrated from other platforms. Most likely you are a "power user" of some version of Microsoft Windows. Perhaps your boss has told you to administer a Linux server, or you're entering the exciting new world of single board computers (SBC) such as the Raspberry Pi. You may just be a desktop user who is tired of all the security problems & wants too give Linux a try. That's fine. All are welcome here.

That being said, there is no shortcut to Linux enlightenment. $\boxed{\text{Learning the command line is challenging \& takes real effort.}}$ It's not that it's so hard, but rather it's so *vast*. The average Linux system has literally *thousands* of programs you can employ on the command line. Consider yourself warned; learning the command line is not a casual[9] endeavor[10].

On the other hand, learning the Linux command line is extremely rewarding. If you think you're a "power user" now, just wait. You don't know what real power is – yet. &, unlike many other computer skills, knowledge of the command line is long-lasting. The skills learned today will still be useful 10 years from now. $\boxed{\text{The command line has survived the test of time.}}$

It is also assumed that you have no programming experience, but don't worry, we'll start you down that path as well." – Shotts, 2019, pp. 32–33

What's in This Book

"This material is presented in a carefully chosen sequence, muck like a tutor sitting next to you guiding you along. Many authors treat this material in a "systematic" fashion, exhaustively[11] covering each topic in order. This makes sense from a writer's perspective but can be very confusing to new users.

[5]**vendor** [n] **1.** a person or company that sells things, especially outside on the street; **2.** (*formal*) a company that sells a particular product; **3.** (*law*) a person who is selling something, especially a house.

[6]**pandit** [n] (also **pundit**) **1.** a Hindu priest or wise man; **2.** (*Indian English*) a teacher; **3.** (*Indian English*) a musician with a lot of skill.

[7]**marvellously** [adv] (US English **marvelously**) very; very well, SYNONYM: **wonderfully**.

[8]**prominence** [n] **1.** [uncountable, singular] the state of being important, well known or easy to notice; **2.** [countable, uncountable] (*medical* or *formal*) a thing that sticks out from something; the fact or state of sticking out from something.

[9]**casual** [a] **1.** [usually before noun] without paying attention to detail; **2.** [usually before noun] not showing much care or thought; **3.** [usually before noun] (of a relationship) lasting only a short time & without deep affection; **4.** [usually before noun] (BE) (of work) not permanent; not regular; **5.** not formal; **6.** [only before noun] happening by chance; doing something by chance.

[10]**endeavour** [n] (US **endeavor**) (*formal*) **1.** [uncountable, countable] serious effort to achieve something; an attempt to do something, especially something new or difficult; **2.** [countable, usually plural] something that somebody does; [v] **endeavor to do something** (*formal*) to try hard to do or achieve something, SYNONYM: **strive**.

[11]**exhaustive** [a] including everything possible; very thorough or complete.

Another goal is to acquaint[12] you with the Unix way of thinking, which is different from the Windows way of thinking. Along the way, we'll go on a few side trips to help you understand why certain things work the way they do & how they got that way. Linux is not just a piece of software; it's also a small part of the larger Unix culture, which has its own language & history. I might throw in a rant or 2, as well.

This book is divided into 4 parts, each covering some aspect of the command line experience.

- **Part 1, "Learning the Shell,"** starts our exploration of the basic language of the command line including such things as the structure of commands, file system navigation, command line editing, & finding help & documentation for commands.
- **Part 2, "Configuration & the Environment,"** covers editing configuration files that control the computer's operation from the command line.
- **Part 3, "Common Tasks & Essential Tools,"** explores many of the ordinary tasks that are commonly performed from the command line. Unix-like operating systems, such as Linux, contain many "classic" command line programs that are used to perform powerful operations on data.
- **Part 4, "Writing Shell Scripts,"** introduces shell programming, an admittedly rudimentary but easy-to-learn technique for automating many common computing tasks. By learning shell programming, you will become familiar with concepts that can be applied to many other programming languages." – Shotts, 2019, pp. 33–34

How to Read This Book

"Start at the beginning of the book & follow it to the end. It isn't written as a reference work; it's really more like a story with a beginning, middle, & end." – Shotts, 2019, p. 34

Prerequisites

"To use this book, all you will need is a working Linux installation. You can get this in 1 of 2 ways:

- **Install Linux on a (not so new) computer.** It doesn't matter which distribution you choose, though most people today start out with either Ubuntu, Fedora, or OpenSUSE. If in doubt, try Ubuntu 1st. Installing a modern Linux distribution can be ridiculously easy or ridiculously difficult depending on your hardware. I suggest a desktop computer that is a couple of years gold & has at least 2GB of RAM & 6GB of free hard disk space. Avoid laptops & wireless networks if at all possible, as these are often more difficult to get working.
- **Use a "live CD" or USB flash drive.** 1 of the cool things you can do with many Linux distributions is run them directly from a CD-ROM or USB flash drive without installing them at all. Just go into your BIOS setup & set your computer to boot from a CD-ROM drive or USB device & reboot. Using this method is a great way to test a computer for Linux compatibility prior to installation. The disadvantage is that it may be slow compared to having Linux installed on your hard drive. Both Ubuntu & Fedora (among others) have live versions.

Regardless of how you install Linux, you'll need to have occasional superuser (i.e., administrative) privileges to carry out the lessons in this book.

After you have a working installation, start reading & follow along with your own computer. Most of the material in this book is "hands on," so sit down & get typing!

Why I don't call it "GNU/LINUX". In some quarters, it's politically correct to call the Linux operating system the "GNU/Linux operating system." The problem with "Linux" is that there is no completely correct way to name it because it was written by many different people in a vast, distributed development effort. Technically speaking, Linux is the name of the operating system's kernel, nothing more. The kernel is very important, of course, since it makes the operating system go, but it's not enough to form a complete operating system.

Enter Richard Stallman, the genius-philosopher who founded the Free Software movement, started the Free Software Foundation, formed the GNU Project, wrote the 1st version of the GNU C Compiler (`gcc`), created the GNU General Public License (the GPL), etc., etc., etc. He *insists* that you call it "GNU/Linux" to properly reflect the contributions of the GNU Project. While the GNU Project predates the Linux kernel & the project's contributions are extremely deserving of recognition, placing them in the name is unfair to everyone else who made significant contributions. Besides, I think "Linux/GNU" would be more technically accurate since the kernel boots 1st & everything else runs on top of it.

In popular usage, Linux refers to the kernel & all the other free & open source software found in the typical Linux distribution, i.e., the entire Linux ecosystem, not just the GNU components. The operating system marketplace seems to prefer 1-word names such as DOS, Windows, macOS, Solaris, Irix, & AIX. I have chosen to use the popular format. If, however, you prefer to use "GNU/Linux" instead, perform a mental search-&-replace while reading this book. I won't mind." – Shotts, 2019, pp. 34–36

[12]**acquaint** [v] (*formal*) **acquaint somebody/yourself with something** to make somebody/yourself familiar with or aware of something.

What's New in the 2nd Edition

"While the basic structure & content remain the same, this edition of *The Linux Command Line* is peppered with various refinements, classifications, & modernizations, many of which are based on reader feedback. In addition, 2 particular improvements stand out. 1st, the book now assumes `bash` version 4.*x*, which was not in wide use at the time of the original manuscript. This 4th major version of `bash` added several useful new features now covered in this edition. 2nd, Part 4, "Shell Scripting," has been improved to provide better examples of good scripting practice. The scripts included in Part 4 have been revised to make them more robust, & I also fixed a few bugs." "This book is an ongoing project, like many open source software projects." – Shotts, 2019, p. 36

Part I: Learning The Shell

1.1 What Is the Shell?

"When we speak of the command line, we are really referring to the *shell*. The shell is a program that takes keyboard commands & passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called `bash`. The name is acronym for *b*ourne-*a*gain *sh*ell, a reference to the fact that `bash` is an enhanced replacement for `sh`, the original Unix shell program written by Steve Bourne." – Shotts, 2019, p. 38

1.1.1 Terminal Emulators

"When using a graphical user interface (GUI), we need another program called a *terminal emulator* to interact with the shell. If we look through our desktop menus, we will probably find one. KDE uses konsole, & GNOME uses gnome-terminal, though it's likely called simply Terminal on your menu. A number of other terminal emulators are available for Linux, but they all basically do the same thing: give us access to the shell. You will probably develop a preference for 1 or another terminal emulator based on the number of bells & whistles it has." – Shotts, 2019, p. 38

1.1.2 Making Your 1st Keystrokes

"Launch the terminal emulator. Once it comes up, we should see something like this: `[me@linuxbox ~]$`. This is called a *shell prompt*, & it will appear whenever the shell is ready to accept input. While it might vary in appearance somewhat depending on the distribution, it will typically include your `username@machinename`, followed by the current working directory (more about that in a little bit) & a dollar sign.

If the last character of the prompt is a hash mark `#` rather than a dollar sign, the terminal session has superuser privileges. This means either we are logged in as the root user or we selected a terminal emulator that provides superuser (administrative) privileges." – Shotts, 2019, p. 39

1.1.2.1 Command History

"If we press the up arrow, we will see that the previous command entered reappears after the prompt. This is called *command history*. Most Linux distributions remember the last 1,000 commands by default. Press the down arrow & the previous command disappears." – Shotts, 2019, p. 39

1.1.2.2 Cursor Movement

"Recall the previous command by pressing the up arrow again. If we try the left & right arrows, we'll see that we can position the cursor anywhere on the command line. This makes editing commands easy.

A few words about mice & focus. While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. A mechanism built into the X Window System (the underlying engine that makes the GUI go) supports a quick copy-&-paste technique. If you highlight some text by holding down the left mouse button & dragging the mouse over it (or double-clicking a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.

Don't be tempted to use CTRL-C & CTRL-V to perform copy & paste inside a terminal window. They don't work. These control codes have different meanings to the shell & were assigned many years before the release of Microsoft Windows.

Your graphical desktop environment (most likely KDE or GNOME), in an effort to behave like Windows, probably has its focus policy set to "click to focus." This means for a window to get focus (become active), you need to click on it. This is contrary to the traditional X behavior of "focus follows mouse," which means that a window gets focus just by passing the mouse over it. The window will not come to the foreground until you click on it, but it will be able to receive input. Setting the focus policy to "focus follows mouse" will make the copy-&-paste technique even more useful. Give it a try if you can (some desktop environments such as Ubuntu's Unity no longer support it). I think if you give it a chance, you will prefer it. You will find this setting in the configuration program for your window manager." – Shotts, 2019, p. 40

1.1.3 Try Some Simple Commands

"Now that we have learned to enter text in the terminal emulator, let's try a few simple commands. Let's begin with the `date` command, which displays the current time & date.

```
[me@linuxbox ~]$ date
Fri Feb 2 15:09:41 EST 2018
```

A related command is `cal`, which, by default, displays a calendar of the current month.

```
[me@linuxbox ~]$ cal
February 2018
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
```

The console behind the curtain. Even if we have no terminal emulator running, several terminal sessions continue to run behind the graphical desktop. We can access these sessions, called *virtual consoles*, by pressing CTRL-ALT-F1 through CTRL-ALT-F6 on most Linux distributions. When a session is accessed, it presents a login prompt into which we can enter our username & password. To switch from 1 virtual console to another, press `alt-F1` through ALT-F6. On most systems, we can return to the graphical desktop by pressing `alt-F7`.

To see the current amount of free space on our disk drives, enter `df`.

```
[me@linuxbox ~]$ df
tmpfs              783256     2288    780968   1% /run
/dev/nvme0n1p2 491039648 79254728 386768040  18% /
tmpfs             3916268   154944   3761324   4% /dev/shm
tmpfs                5120        4      5116   1% /run/lock
/dev/nvme0n1p1      94759     5329     89430   6% /boot/efi
tmpfs              783252      156    783096   1% /run/user/1000
```

Likewise, to display the amount of free memory, enter the `free` command." – Shotts, 2019, pp. 41–42

```
[me@linuxbox ~]$ free
               total        used        free      shared  buff/cache   available
Mem:         7832540     6709784      217512      327756      905244      502532
Swap:              0           0           0
```

1.1.4 Ending a Terminal Session

"We can end a terminal session by closing the terminal emulator window, by entering the `exit` command at the shell prompt, or by pressing `ctrl-D`. `[me@linuxbox ~]$ exit`." – Shotts, 2019, p. 42

1.2 Navigation

"The 1st thing we need to learn (besides how to type) is how to navigate the file system on our Linux system. In this chapter, we will introduce the following commands: `pwd`: print name of current working directory, `cd`: change directory, `ls`: list directory contents." – Shotts, 2019, p. 43

1.2.1 Understanding the File System Tree

"Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a *hierarchical directory structure*. This means they are organized in a tree-like pattern of directories (sometimes called *folders* in other systems), which may contain files & other directories. The 1st directory in the file system is called the *root directory*. The root directory contains files & subdirectories, which contain more files & subdirectories, & so on.

Note that unlike Windows, which has a separate file system tree for each storage device, Unix-like systems such as Linux always have a single file system tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly, *mounted*) at various points on the tree according to the whims of the *system administrator*, the person (or people) responsible for the maintenance of the system." – Shotts, 2019, pp. 43–44

1.2.2 The Current Working Directory

"Most of us are probably familiar with a graphical file manager that represents the file system tree. Notice that the tree is usually shown upended, i.e., with the root at the top & the various branches descending below.

However, the command line has no pictures, so to navigate the file system tree, we need to think of it in a different way.

Imagine that the file system is a maze shaped like an upside-down tree & we are able to stand in the middle of it. At any given time, we are inside a single directory, & we can see the files contained in the directory & the pathway to the directory above us (called the *parent directory*) & any subdirectories below us. The directory we are standing in is called the *current working directory*. To display the current working directory, we use the `pwd` (print working directory) command.

```
[me@linuxbox ~]$ pwd
/home/me
```

When we 1st log in to our system (or start a terminal emulator session), our current working directory is set to our *home directory*. Each user account is given its own home directory, & it is the only place a regular user is allowed to write files." – Shotts, 2019, pp. 44–45

1.2.3 Listing the Contents of a Directory

"To list the files & directories in the current working directory, we use the `ls` command.

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Actually, we can use the `ls` command to list the contents of any directory, not just the current working directory, & there are many other fun things it can do as well. We'll spend more time with `ls` in Chap. 3." – Shotts, 2019, p. 45

1.2.4 Changing the Current Working Directory

"To change our working directory (where we are standing in the tree-shaped maze), we use the `cd` command. To do this, type `cd` followed by the pathname of the desired working directory. A pathname is the route we take along the branches of the tree to get to the directory we want. We can specify pathnames in 1 of 2 different ways: as *absolute pathnames* or as *relative pathnames*. Let's deal with absolute pathnames 1st." – Shotts, 2019, p. 45

1.2.4.1 Absolute Pathnames

"An absolute pathname beings with the root directory & follows the tree branch by branch until the path to the desired directory or file is completed. E.g., there is a directory on your system in which most of the system's programs are installed. The directory's pathname is `/usr/bin`. I.e., from the root directory (represented by the leading slash in the pathname) there is a directory called `usr` that contains a directory called `bin`.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Listing of many, many files ...
```

Now we can see that we have changed the current working directory to `/usr/bin` & that it is full of files. Notice how the shell prompt has changed? As a convenience, it is usually set up to automatically display the name of the working directory." – Shotts, 2019, p. 46

1.2.4.2 Relative Pathnames

"Where an absolute pathname starts from the root directory & leads to its destination, a relative pathname starts from the working directory. To do this, it uses a couple of special notations to represent relative positions in the file system tree. These special notations are `.` (dot) & `..` (dot dot).

The `.` notation refers to the working directory, & the `..` notation refers to the working directory's parent directory. Here is how it works. Let's change the working directory to `/usr/bin` again.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now let's say that we wanted to change the working directory to the parent of `/usr/bin`, which is `/usr`. We could do that 2 different ways, either with an absolute pathname:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

or with a relative pathname:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

2 different methods with identical results. Which one should use? [13]

Likewise, we can change the working directory from `/usr` to `/usr/bin` in 2 different ways, either using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

or using a relative pathname:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now, there is something important to point out here. In almost all cases, we can omit the `./` part because it is implied. Typing the following does the same thing:

```
[me@linuxbox usr]$ cd bin
```

In general, if we do not specify a pathname to something, the working directory will be assumed.

Important facts about filenames. On Linux systems, files are named in a manner similar to that of other systems such as Windows, but there are some important differences.

- Filenames that begin with a period character are hidden. This only means that `ls` will not list them unless you say `ls -a`. When you account was created, several hidden files were placed in your home directory to configure things for your account. In Chap. 11 we will take a closer look at some of these files to see how you can customize your environment. In addition, some applications place their configuration & setting files in your home directory as hidden files.
- Filenames & commands in Linux, like Unix, are case sensitive. The filenames `File1` & `file1` refer to different files.
- Though Linux supports long filenames that may contain embedded spaces & punctuation characters, limit the punctuation characters in the names of files you create to period, dash, & underscore. *Most important, do not embed spaces in filenames.* If you want to represent spaces between words in a filename, use underscore characters. You will thank yourself later.
- Linux has no concept of a "file extension" like some other operating systems. You may name files any way you like. The contents or purpose of a file is determined by other means. Although Unix-like operating systems don't use file extensions to determine the contents/purpose of files, many application programs do." – Shotts, 2019, pp. 46–48

1.2.4.3 Some Helpful Shortcuts

Some useful ways to quickly change the current working directory. CD Shortcuts: • `cd`: Changes the working directory to your home directory. • `cd -`: Changes the working directory to the previous working directory. • `cd ~user_name`: Changes the working directory to the home directory of `user_name`.

[13]The one that requires the least typing!

1.3 Exploring the System

"Now that we know how to move around the file system, it's time for a guided tour of our Linux system. Before we start, however, we're going to learn some more commands that will be useful along the way. • `ls`: List directory contents. • `file`: Determine file type. • `less`: View file contents." – Shotts, 2019, p. 50

1.3.1 More Fun with `ls`

"The `ls` command is probably the most used command, & for good reason. With it, we can see directory contents & determine a variety of important file & directory attributes. As we have seen, we can simply enter `ls` to get a list of files & subdirectories contained in the current working directory.

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Besides the current working directory, we can specify the directory to list, like so:

```
me@linuxbox ~]$ ls /usr
bin games include lib local sbin share src
```

We can even specify multiple directories. In the following example, we list both the user's home directory (symbolized by the ~ character) & the `/usr` directory:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop Documents Music Pictures Public Templates Videos
/usr:
bin games include lib local sbin share src
```

We can also change the format of the output to reveal more detail.

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Videos
```

By adding `-l` to the command, we changed the output to the long format." – Shotts, 2019, pp. 50–51

1.3.1.1 Options & Arguments

"This brings us to a very important point about how most commands work. Commands are often followed by 1 or more *options* that modify their behavior &, further, by 1 or more *arguments*, the items upon which the command acts. So, most commands look kind of like this: `command -options arguments`. Most commands use options, which consist of a single character preceded by a dash, e.g., `-l`. Many commands, however, including those from the GNU Project, also support *long options*, consisting of a word preceded by 2 dashes. Also, many commands allow multiple short options to be strung together. In the following example, the `ls` command is given 2 options, which are the `l` option to produce long format output, & the `t` option to sort the result by the file's modification time.

```
[me@linuxbox ~]$ ls -lt
```

We'll add the long option `--reverse` to reverse the order of the sort.

```
[me@linuxbox ~]$ ls -lt --reverse
```

Remark 1.1. *Command options, like filenames in Linux, are case sensitive.*

The `ls` command has a large number of possible options, the most common of which are listed in Table 3.1. Common `ls` Options. Option : Long option : Description.

- `-a`: `--all`: List all files, even those with names that begin with a period, which are normally not listed (i.e., hidden).

- `-A`: `--almost-all`: Like the `-a` option except it does not list. (current directory) & `..` (parent directory).
- `-d`: `--directory`: Ordinarily, if a directory is specified, `ls` will list the contents of the directory, not the directory itself. Use this option in conjunction with the `-l` option to see details about the directory rather than its contents.
- `-F`: `--classify`: This option will append an indicator character to the end of each listed name. E.g., it will append a forward slash `/` if the name is a directory.
- `-h`: `--human-readable`: In long format listings, display file sizes in human-readable format rather than in bytes.
- `-l`: Display results in long format.
- `-r`: `--reverse`: Display the results in reverse order. Normally, `ls` displays its results in ascending alphabetical order.
- `-s`: Sort results by file size.
- `-t`: Sort by modification in time." – Shotts, 2019, pp. 51–53

1.3.1.2 A Longer Look at Long Format

As we aw earlier, the `-l` option causes `ls` to display its results in long format. This format contains a great deal of useful information. Here is the `Examples` directory from an Ubuntu system:

```
-rw-r--r-- 1 root root 3576296 2017-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2017-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root 47584 2017-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root 44355 2017-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root 34391 2017-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root 32059 2017-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root 159744 2017-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root 27837 2017-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root 98816 2017-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root 453764 2017-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root 358374 2017-04-03 11:05 ubuntu Sax.ogg
```

Table 3-2 provides us with a look at the different fields from 1 of the files & their meanings. Table 3-2: `ls` Long Listing Fields. Field : Meaning.

- `-rw-r--r--`: Access rights to the file. The 1st character indicates the type of file. Among the different types, a leading dash means a regular file, while a `d` indicates a directory. The next 3 characters are the access rights for the file's owner, the next 3 are for members of the file's group, & the final 3 are for everyone else.
- `1`: File's number of hard links.
- `root`: The username of the file's owner.
- `root`: The name of the group that owns the file.
- `32059`: Size of the file in bytes.
- `2017-04-03 11:05`: Date & time of the file's last modification.
- `oo-cd-cover.odf`: Name of the file." – Shotts, 2019, pp. 53–54

1.3.2 Determining a File's Type with file

"As we explore the system, it will be useful to know what files contain. To do this, we will use the `file` command to determine a file's type. As we discussed earlier, filenames in Linux are not required to reflect a file's contents. While a filename like `picture.jpg` would normally be expected to contain a JPEG-compressed image, it is not required to in Linux. We can invoke the `file` command this way: `file filename`. When invoked, the `file` command will print a brief description of the file's contents. E.g.:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

There are many kinds of files. In fact, 1 of the common ideas in Unix-like operating systems such as Linux is that "everything is a file." As we proceed with our lessons, we will see just how true that statement is.

While many of the files on our system are familiar, e.g., MP3 & JPEG files, there are many kinds that are a little less obvious & a few that are quite strange." – Shotts, 2019, pp. 54–55

1.3.3 Viewing File Contents with `less`

"The `less` command is a program to view text files. Throughout our Linux system, there are many files that contain human-readable text. The `less` program provides a convenient way to examine them.

Why would we want to examine text files? Because many of the files that contain system settings (called *configuration files*) are stored in this format, & being able to read them gives us insight about how the system works. In addition, some of the actual programs that the system uses (called *scripts*) are stored in this format. In later chapters, we will learn how to edit text files in order to modify system settings & write our own scripts, but for now we will just took at their contents.

The `less` command is used like this: `less filename`.

What is "text"? There are many ways to represent information on a computer. All methods involve defining a relationship between the information & some numbers that will be used to represent it. Computers, after all, understand only numbers, & all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed video files), while others are rather simple. 1 of the earliest & simplest is called ASCII text. ASCII (pronounced "as-key") is short for American Standard Code for Information Interchange. This is a simple encoding scheme that was 1st used on Teletype machines to map keyboard characters to numbers.

Text is a simple 1-to-1 mapping of characters to numbers. It is very compact. 50 characters of text translates to 50 bytes of data. It is important to understand that text only contains a simple mapping of characters to numbers. It is not the same as a word processor document such as one created by Microsoft Word or LibreOffice Writer. Those files, in contrast to simple ASCII text, contain many non-text elements that are used to describe its structure & formatting. Plain ASCII text files contain only the characters themselves & a few rudimentary control codes such as tabs, carriage returns, & line feeds.

Throughout a Linux system, many files are stored in next format, & there are many Linux tools that work with text files. Even Windows recognizes the importance of this format. The well-known NOTEPAD.EXE program is an editor for plain ASCII text files.

Once started, the `less` program allows us to scroll forward & backward through a text file. E.g., to examine the file that defines all the system's user accounts, enter the following command:

```
[me@linuxbox ~]$ less /etc/passwd
```

Once the `less` program starts, we can view the contents of the file. If the file is longer than 1 page, we can scroll up & down. To exit `less`, press `q`. Table 3-3 lists the most common keyboard commands used by `less`. Table 3-3: `less` Commands. Command : Action

- PAGE UP or `b`: Scroll back 1 page.
- PAGE DOWN or space: Scroll forward 1 page.
- Up arrow: Scroll up 1 line.
- Down arrow: Scroll down 1 line.
- `G`: Move to the end of the text file.
- `1G` or `g`: Move to the beginning of the text file.
- `/characters`: Search forward to the next occurrence of `characters`.
- `n`: Search for the next occurrence of the previous search.
- `h`: Display help screen.
- `q`: Quit `less`.

Less is more. The `less` program was designed as an improved replacement of an earlier Unix program called *more*. The name *less* is a play on the phrase "less is more" – a motto of modernist architects & designers.

`less` falls into the class of programs called *papers*, programs that allow the easy viewing of long text documents in a page-by-page manner. Whereas the `more` program could only page forward, the `less` program allows paging both forward & backward & has many other features as well." – Shotts, 2019, pp. 55–57

1.3.4 Taking a Guided Tour

"The file system layout on a Linux system is much like that found on other Unix-like systems. The design is actually specified in a published standard called the *Linux Filesystem Hierarchy Standard.* Not all Linux distributions conform to the standard exactly, but most come pretty close.

Remember the copy-&-paste trick! If you are using a mouse, you can double-click a filename to copy it & middle-click to paste it into commands.

Next, we are going to wander around the file system ourselves to see what makes our Linux system tick. This will give us a chance to practice our navigation skills. 1 of the things we will discover is that many of the interesting files are in plain human-readable text. As we go about our tour, try the following: • `cd` into a given directory. • List the directory contents with `ls -l`. 1. If you see an interesting file, determine its contents with `file`. 2. If it looks like it might be text, try viewing it with `less`. If we accidentally attempt to view a non-text file & it scrambles the terminal window, we can recover by entering the `reset` command.

As we wander around, don't be afraid to look at stuff. Regular users are largely prohibited from messing things up. That's the system administrator's job! If a command complains about something, just move on to something else. Spend some time looking around. The system is ours to explore. Remember, $\boxed{\text{in Linux, there are no secrets!}}$

Table 3-4 lists just a few of the directories we can explore. There may be some slight differences depending on our Linux distribution. Don't be afraid to look around & try more! Table 3-4: Directories Found on Linux Systems. Directory : Comments.

- `/`: The root directory, where everything begins.
- `/bin`: Contains binaries (programs) that must be present for the system to boot & run.
- `/boot`: Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), & the boot loader. Interesting files include `/boot/grub/grub.conf`, or `menu.lst`, which is used to configure the boot loader, & `/boot/vmlinuz` (or something similar), the Linux kernel.
- `/dev`: This is a special directory that contains *device nodes.* "Everything is a file" also applies to devices. Here is where the kernel maintains a list of all the devices it understands.
- `/etc`: The `/etc` directory contains all the system-wide configuration files. It also contains a collection of shell scripts that start each of the system services at boot time. Everything in this directory should be readable text. While everything in `/etc` is interesting, here are some all-time favorites: `/etc/crontab`, a file that defines when automated jobs will run; `/etc/fstab`, a table of storage devices & their associated mount points; & `/etc/passwd`, a list of the user accounts.
- `/home`: In normal configurations, each user is given a directory in `/home`. Ordinary users can write files only in their home directories. This limitation protects the system from errant[14] user activity.
- `/lib`: Contains shared library files used by the core system programs. These are similar to dynamic link libraries (DLLs) in Windows.
- `/lost+found`: Each formatted partition or device using a Linux file system, such as `ext3`, will have this directory. It is used in the case of a partial recovery from a file system corruption event. Unless something really bad has happened to your system, this directory will remain empty.
- `/media`: On modern Linux systems, the `/media` directory will contain the mount points for removable media such as USB drives, CD-ROMs, & so on, that are mounted automatically at insertion.
- `/mnt`: On older Linux systems, the `/mnt` directory contains mount points for removable devices that have been mounted manually.
- `/opt`: The `/opt` directory is used to install "optional" software. This is mainly used to hold commercial software products that might be installed on the system.
- `/proc`: The `/proc` directory is special. It's not a real file system in the sense of files stored on your hard drive. Rather, it is a virtual file system maintained by the Linux kernel. The "files" it contains are peepholes[15] into the kernel itself. The files are readable & will give you a picture of how the kernel sees your computer.
- `/root`: This is the home directory for the root account.

[14]**errant** [a] [only before noun] (*formal or humorous*) **1.** doing something that is wrong; not behaving in an acceptable way; **2.** (of a husband or wife) not sexually faithful.

[15]**peephole** [n] a small opening in a wall, door, etc. that you can look through.

- `/sbin`: This directory contains "system" binaries. These are programs that perform vital system tasks that are generally reserved for the superuser.
- `/tmp`: The `/tmp` directory is intended for the storage of temporary, transient files created by various programs. Some configurations cause this directory to be emptied each time the system is rebooted.
- `/usr`: The `/usr` directory tree is likely the largest one on a Linux system. It contains all the programs & support files used by regular users.
- `/usr/bin`; `/usr/bin` contains the executable programs installed by your Linux distribution. It is not uncommon for this directory to hold thousands of programs.
- `/usr/lib`: The shared libraries for the programs in `/usr/bin`.
- `/usr/local`: The `/usr/local` tree is where programs that are not included with your distribution but are intended for system-wide use are installed. Programs compiled from source code are normally installed in `/usr/local/bin`. On a newly installed Linux system, this tree exists, but it will be empty until the system administrator puts something in it.
- `/usr/sbin`: Contains more system administration programs.
- `/usr/share`: `/usr/share` contains all the shared data used by programs in `/usr/bin`. This includes things such as default configuration files, icons, screen backgrounds, sound files, & so on.
- `/usr/share/doc`: Most packages installed on the system will include some kind of documentation. In `/usr/share/doc`, we will find documentation files organized by package.
- `/var`: With the exception of `/tmp` & `/home`, the directories we have looked at so far remain relatively static; i.e., their contents don't change. The `/var` directory tree is where data that is likely to change is stored. Various databases, spool files, user mail, & so forth, are located here.
- `/var/log`: `/var/log` contains *log files*, records of various system activity. These are important & should be monitored from time to time. The most useful ones are `/var/log/messages` & `/var/log/syslog`. Note that for security reasons on some systems, you must be the superuser to view log files." – Shotts, 2019, pp. 57–61

1.3.5 Symbolic Links

"As we look around, we are likely to see a directory listing (e.g., `/lib`) with an entry like this:

```
lrwxrwxrwx 1 root root    11 2018-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Notice how the 1st letter of the listing is `l` & the entry seems to have 2 filenames? This is a special kind of a file called a *symbolic link* (also known as a *soft link* or *symlink*). In most Unix-like systems, it is possible to have a file referenced by multiple names. While the value of this might not be obvious, it is really a useful feature.

Picture this scenario: a program requires the use of a shared resource of some kind contained in a file named "foo," but "foo" has frequent version changes. It would be good to include the version number in the filename so the administrator or other interested party could see what version of "foo" is installed. This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it & change it to look for a new resource name every time a new version of the resource is installed. That doesn't sound like fun at all.

Here is where symbolic links save the day. Suppose we install version 2.6 of "foo," which has the filename "foo-2.6," & then create a symbolic link simply called "foo" that points to "foo-2.6." This means that when a program opens the file "foo," it is actually opening the file "foo-2.6." Now everybody is happy. The programs that rely on "foo" can find it, & we can still see what actual version is installed. What it is time to upgrade to "foo-2.7," we just add the file to our system, delete the symbolic link "foo," & create a new one that points to the new version. Not only does this solve the problem of the version upgrade, it also allows us to keep both versions on our machine. Imagine that "foo-2.7" has a bug (damn those developers!), & we need to revert to the old version. Again, we just delete the symbolic link pointing to the new version & create a new symbolic link pointing to the old version.

The directory listing at the beginning of this section (from the `/lib` directory of a Fedora system) shows a symbolic link called `libc.so.6` that points to a shared library file called `libc-2.6.so`. This means that programs looking for `libc.so.6` will actually get the file `libc-2.6.so`." – Shotts, 2019, pp. 61–63

1.3.6 Hard Links

"While we are on the subject of links, we need to mention that there is a 2nd type of link called *hard links*. Hard links also allow files to have multiple names, but they do it in a different way." – Shotts, 2019, p. 63

1.3.7 Summing Up

"With our tour behind us, we have learned a lot about our system. We've seen various files & directories & their contents. 1 thing you should take away from this is how open the system is. In Linux there are many important files that are plain human-readable text. Unlike many proprietary systems, Linux makes everything available for examination & study." – Shotts, 2019, p. 63

1.4 Manipulating Files & Directories

"At this point, we are ready for some real work! This chapter will introduce 5 of the most frequently used Linux commands. The following commands are used for manipulating both files & directories: • `cp`: Copy files & directories. • `mv`: Move/rename files & directories. • `mkdir`: Create directories. 1. `rm`: Remove files & directories. • `ln`: Create hard & symbolic links.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag & drop a file from 1 directory to another, cut & paste files, delete files, & so on. So why use these old command line programs?

The answer is power & flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. E.g., how could we copy all the HTML files from 1 directory to another but copy only files that do not exist in the destination directory or are newer than the versions in the destination directory? It's pretty hard with a file manager but pretty easy with the command line. `cp -u *.html destination`." – Shotts, 2019, pp. 64–65

1.4.1 Wildcards

"Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful. Because the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are called *wildcards*. Using wildcards (which is also known as *globbing*) allows you to select filenames based on patterns of characters. Table 4-1 lists the wildcards & what they select. Table 4-1: Wildscards. Wildcard : Meaning.

- `*`: Matches any characters.
- `?`: Matches any single character.
- `[characters]`: Matches any character that is a member of the set `characters`.
- `[[:class:]]` Matches any character that is a member of the specified `class`.

Table 4-2 lists the most commonly used character classes. Table 4-2: Commonly Used Character Classes. Character class : Meaning.

- `[:alnum:]`: Matches any alphanumeric character.
- `[:alpha:]`: Matches any alphabetic character.
- `[:digit:]`: Matches any numeral.
- `[:lower:]`: Matches any lowercase letter.
- `[:upper:]`: Matches any uppercase letter.

Using wildcards makes it possible to construct sophisticated selection criteria for filenames. Table 4-3 provides some examples of patterns & what they match. Table 4-3: Wildcard Examples. Pattern : Matches.

- `*`: All files.
- `g*`: Any file beginning with `g`:
- `b*.txt`: Any file beginning with `b` followed by any characters & ending with `.txt`.
- `Data???`: Any file beginning with `Data` followed by exactly 3 characters.
- `[abc]*`: Any file beginning with either an `a`, a `b`, or a `c`.
- `BACKUP.[0-9][0-9][0-9]`: Any file beginning with `BACKUP.` followed by exactly 3 numerals.
- `[[:upper:]]*`: Any file beginning with an uppercase letter.

- `[![:digit:]]*`: Any file not beginning with a numeral.
- `*[[:lower:]123]`: Any file ending with a lowercase letter or the numerals `1`, `2`, or `3`.

Wildcards work in the GUI, too. Wildcards are especially valuable not only because they are used so frequently on the command line but because they are also supported by some graphical file managers.

- In Nautilus (the file manager for GNOME), you can select files using the Edit ▷ Select Pattern menu item. Just enter a file selection pattern with wildcards & the files in the currently viewed directory will be highlighted for selection.
- In some versions of Dolphin & Konqueror (the file managers for KDE), you can enter wildcards directly on the location bar. E.g., if you want to see all the files starting with a lowercase `u` in the `/usr/bin` directory, enter `/usr/bin/u*` in the location bar & it will display the result.

Many ideas originally found in the command line interface make their way into the graphical interface, too. It is 1 of the many things that make the Linux desktop so powerful.

Wildcards can be used with any command that accepts filenames as arguments.

Character ranges. If you coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` & `[a-z]` character range notations. These are traditional Unix notations & worked in older versions of Linux as well. They can still work, but you have to be careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them & use character classes instead." – Shotts, 2019, pp. 65–67

1.4.2 `mkdir` – Create Directories

"The `mkdir` command is used to create directories. It works like this: `mkdir directory...`. Note that when 3 periods follow an argument in the description of a command (as in the preceding example), it means that the argument can be repeated. Thus, the following command `mkdir dir1` would create a single directory named `dir1`. This command `mkdir dir1 dir2 dir3` would create 3 directories named `dir1, dir2`, & `dir3`." – Shotts, 2019, pp. 67–68

1.4.3 `cp` – Copy Files & Directories

"The `cp` command copies files or directories. It can be used 2 different ways. The following: `cp item1 item2` copies the single file or directory `item1` to the file or directory `item2`. This command: `cp item... directory` copies multiple items (either files or directories) into a directory." – Shotts, 2019, p. 68

1.4.3.1 Useful Options & Examples

"Table 4-4 describes some of the commonly used options (the short option & the equivalent long option) for `cp`. Table 4-4: `cp` Options. Option : Meaning.

- `-a, --archive`: Copy the files & directories & all of their attributes, including ownerships & permissions. Normally, copies take on the default attributes of the user performing the copy. We'll take a look at file permissions in Chap. 9.
- `-i, --interactive`: Before overwriting an existing file, prompt the user for confirmation. If this option is not specified, `cp` will silently (meaning there will be no warning) overwrite files.
- `-r, --recursive`: Recursively copy directories & their contents. This option (or the `-a` option) is required when copying directories.
- `-u, --update`: When copying files from 1 directory to another, only copy files that either don't exist or are newer than the existing corresponding files in the destination directory. This is useful when copying large numbers of files as it skips files that don't need to be copied.
- `-v, --verbose`: Display informative messages as the copy is performed.

Table 4-5 provides some examples of these commonly used options. Table 4-5: `cp` Examples: Command. Results.

- `cp file1 file2`: Copy `file1` to `file2`. If `file2` exists, it is overwritten with the contents of `file1`. If `file2` does not exist, it is created.
- `cp -i file1 file2`: Same as previous command, except that if `file2` exists, the user is prompted before it is overwritten.
- `cp file1 file2 dir1`: Copy `file1` & `file2` into directory `dir1`. The directory `dir1` must already exist.

- `cp dir1/* dir2`: Using a wildcard, copy all the files in `dir1` into `dir2`. The directory `dir2` must already exist.
- `cp -r dir1 dir2`: Copy the contents of directory `dir1` to directory `dir2`. If directory `dir2` does not exist, it is created &, after the copy, will contain the same contents as directory `dir1`. If directory `dir2` does exist, then directory `dir1` (& its contents) will be copied into `dir2`." – Shotts, 2019, pp. 68–70

1.4.4 `mv` – Move & Rename Files

"The `mv` command performs both file moving & file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. `mv` is used in much the same way as `cp`, as shown here: `mv item1 item2` to move or rename the file or directory `item1` to `item2`. It's also used as follows: `mv item... directory` to move 1 or more items from 1 directory to another." – Shotts, 2019, p. 70

1.4.4.1 Useful Options & Examples

"`mv` shares many of the same options as `cp`, as described in Table 4-6. Table 4-6: `mv` Options. Option : Meaning.

- `-i, --interactive`: Before overwriting an existing file, prompt the user for confirmation. If this option is not specified, `mv` will silently overwrite files.
- `-u, --update`: When moving files from 1 directory to another, only move files that either don't exist or are newer than the existing corresponding files in the destination directory.
- `-v, --verbose`: Display informative messages as the move is performed.

Table 4-7 provides some examples of using the `mv` command. Table 4-7: `mv` Examples. Command : Results.

- `mv file1 file2`: Move `file1` to `file2`. If `file2` exists, it is overwritten with the contents of `file1`. If `file2` does not exist, it is created. In either case, `file1` ceases to exist.
- `mv -i file1 file2`: Same as the previous command, except that if `file2` exists, the user is prompted before it is overwritten.
- `mv file1 file2 dir1`: Move `file1` & `file2` into directory `dir1`. The directory `dir1` must already exist.
- `mv dir1 dir2`: If directory `dir2` does not exist, create directory `dir2` & move the contents of directory `dir1` into `dir2` & delete directory `dir1`. If directory `dir2` does exist, move directory `dir1` (& its contents) into directory `dir2`." – Shotts, 2019, pp. 70–71

1.4.5 `rm` – Remove Files & Directories

"The `rm` command is used to remove (delete) files & directories, as shown here: `rm item...` where `item` is 1 or more files or directories." – Shotts, 2019, pp. 71–7

1.4.5.1 Useful Options & Examples

"Table 4-8 describes some of the common options for `rm`. Table 4-8: `rm` Options. Option : Meaning.

- `-i, --interactive`: Before deleting an existing file, prompt the user for confirmation. If this option is not specified, `rm` will silently delete files.
- `-r, --recursive`: Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
- `-f, --force`: Ignore nonexistent files & do not prompt. This overrides the `--interactive` option.
- `-v, --verbose`: Display information messages as the deletion is performed.

Be careful with `rm`! Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with `rm`, it's gone. Linux assumes you're smart & you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type the following: `rm *.html`. This is correct, but if you accidentally place a space between the `*` & the `.html` like so: `rm * .html` the `rm` command will delete all the files in the directory & then complain that there is no file called `.html`.

Here is a useful tip: whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard 1st with `ls`. This will let you see the files that will be deleted. Then press the up arrow to recall the command & replace `ls` with `rm`.

Table 4-9 provides some examples of using the `rm` command. Table 4-9: `rm` Examples. Command : Results.

- `rm file1`: Delete `file1` silently.
- `rm -i file1`: Same as the previous command, except that the user is prompted for confirmation before the deletion is performed.
- `rm -r file1 dir1`: Delete `file1` & `dir1` & its contents.
- `rm -rf file1 dir1`: Same as the previous command, except that if either `file1` or `dir1` does not exist, `rm` will continue silently." – Shotts, 2019, pp. 71–73

1.4.6 `ln` – Create Links

"The `ln` command is used to create either hard or symbolic links. It is used in 1 of 2 ways. The following creates a hard link: `ln file link`. The following creates a symbolic link: `ln -s item link` where `item` is either a file or a directory." – Shotts, 2019, p. 73

1.4.6.1 Hard Links

"Hard links are the original Unix way of creating links, compared to symbolic links, which are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have 2 important limitations.

- A hard link cannot reference a file outside its own file system. This means a link cannot reference a file that is not on the same disk partition as the link itself.
- A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when you list a directory containing a hard link, you will see no special indication of the link. When a hard link is deleted, the link is removed, but the contents of the file itself continue to exist (i.e., its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next." – Shotts, 2019, pp. 73–74

1.4.6.2 Symbolic Links

"Symbolic links were created to overcome the limitations of hard links. They work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut, though of course they predate the Windows feature by many years.

A file pointed to by a symbolic link & the symbolic link itself are largely indistinguishable from one another. E.g., if you write something to the symbolic link, the referenced file is written to. When you delete a symbolic link, however, only the *link* is deleted, not the *file* itself. If the file is deleted before the symbolic link, the link will continue to exist but will point to nothing. In this case, the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence. The concept of links can seem confusing, but hang in there. We're going to try all this stuff, & it will, ideally, become clear." – Shotts, 2019, p. 74

1.4.7 Building a Playground

"Because we are going to do some real file manipulation, let's build a safe place to "play" with our file manipulation commands. 1st we need a directory to work in. We'll create one in our home directory & call it `playground`." – Shotts, 2019, p. 75

1.4.7.1 Creating Directories

"The `mkdir` command is used to create a directory. To create our playground directory, we will 1st make sure we are in our home directory & will then create the new directory.

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's crate a couple of directories inside it called `dir1` & `dir2`. To do this, we will change our current working directory to `playground` & execute another `mkdir`.

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create both directories with a single command." – Shotts, 2019, p. 75

1.4.7.2 Copying Files

"Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the `passwd` file from the `/etc` directory to the current working directory.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file.

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2018-01-10 16:40 dir1
drwxrwxr-x 2 me me 4096 2018-01-10 16:40 dir2
-rw-r--r-- 1 me me 1650 2018-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the `-v` option (verbose) to see what it does.

```
[me@linuxbox playground]$ cp -v /etc/passwd .
'/etc/passwd' -> './passwd'
```

The `cp` command performed the copy again but this time displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the 1st copy without any warning. Again, this is a case of `cp` assuming that we know what we're doing. To get a warning, we'll include the `-i` (interactive) option.

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite './passwd'?
```

Responding to the prompt by entering a `y` will cause the file to be overwritten; any other character (e.g., `n`) will cause `cp` to leave the file alone." – Shotts, 2019, pp. 75–76

1.4.7.3 Moving & Renaming Files

"Now, the name `passwd` doesn't seem very playful & this is a playground, so let's change it to something else.

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories & back again. The following moves it 1st to the directory `dir1`:

```
[me@linuxbox playground]$ mv fun dir1
```

The following then moves it from `dir1` to `dir2`:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

Finally, this command brings it back to the current working directory:

```
[me@linuxbox playground]$ mv dir2/fun .
```

Next, let's see the effect of `mv` on directories. 1st we will move our data file into `dir1` again, like this:

```
[me@linuxbox playground]$ mv fun dir1
```

Then we move `dir1` into `dir2` & confirm it with `ls`.

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2018-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2018-01-10 16:33 fun
```

Note that because `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back.

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

" – Shotts, 2019, pp. 76–77

1.4.7.4 Creating Hard Links

"Now we'll try some links. We'll 1st create some hard links to our data file, like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have 4 instances of the file `fun`. Let's take a look at our playground directory.

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2018-01-14 16:17 dir1
drwxrwxr-x 2 me me 4096 2018-01-14 16:17 dir2
-rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun-hard
```

1 thing we notice is that both the 2nd fields in the listings for `fun` & `fun-hard` contain a `4`, which is the number of hard links that now exist for the file. Remember that a file will always have at least 1 link because the file's name is created by a link. So, how do we know that `fun` & `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` & `fun-hard` are both the same size (field `5`), our listing provides no way to be sure. To solve this problem, we're going to have a dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of 2 parts.

- The data part containing the file's contents.
- The name part that holds the file's name.

When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode*, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the `-i` option.

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me me 4096 2018-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me me 4096 2018-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun
12353538 -rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun-hard
```

In this version of the listing, the 1st field is the inode number & as we can see, both `fun` & `fun-hard` share the same inode number, which confirms they are the same file." – Shotts, 2019, pp. 77–78

1.4.7.5 Creating Symbolic Links

"Symbolic links were created to overcome the 2 disadvantages of hard links:

- Hard links cannot span physical devices.
- Hard links cannot reference directories, only files.

Symbolic links are a special type of file that contains a text pointer to the target file or directory. Creating symbolic links is similar to creating hard links.

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The 1st example is pretty straightforward; we simply add the `-s` option to create a symbolic link rather than a hard link. But what about the next two? Remember that when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output, shown here:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me    6 2018-01-15 15:17 fun-sym -> ../fun
```

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading `l` in the 1st field & that it points to `../fun`, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice, too, that the length of the symbolic link file is 6, the number of characters in the string `../fun` rather than the length of the file to which it is pointing.

When creating symbolic links, you can either use absolute pathnames, as shown here:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. In most cases, using relative pathnames is more desirable because it allows a directory tree containing symbolic links & their referenced files to be renamed &/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories.

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me me    4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir2
-rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me    3 2018-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. The presence of a broken link is not in & of itself dangerous, but it is rather messy. If we try to use a broken link, we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links here:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir1
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir2
```

1 thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` " – Shotts, 2019, pp. 79–80

1.4.7.6 Removing Files & Directories

"As we covered earlier, the `rm` command is used to delete files & directories. We are going to use it to clean up our playground a little bit. 1st, let's delete 1 of our hard links.

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me me    4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir2
-rw-r--r-- 3 me me 1650 2018-01-10 16:33 fun
lrwxrwxrwx 1 me me    3 2018-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone, & the link count shown for `fun` is reduced from 4 to 3, as indicated in the 2nd field of the directory listing. Next, we'll delete the file `fun`, & just for enjoyment, we'll include the `-i` option to show what that does.

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file 'fun'?
```

Enter `y` at the prompt & the file is deleted. But let's look at the output of `ls` now. Notice what happened to `fun-sym`? Because it's a symbolic link pointing to a now-nonexistent file, the link is broken.

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me me
4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir2
lrwxrwxrwx 1 me me
3 2018-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. The presence of a broken link is not in & of itself dangerous, but it is rather messy. If we try to use a broken link, we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links here:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir1
drwxrwxr-x 2 me me 4096 2018-01-15 15:17 dir2
```

1 thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory & use `rm` with the recursive option (`-r`) to delete `playground` & all of its contents, including its subdirectories.

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Creating symlinks with the GUI. The file managers in both GNOME & KDE provide an easy & automatic method of creating symbolic links. With GNOME, holding the CTRL-SHIFT keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file." – Shotts, 2019, pp. 80–82

1.4.8 Summing Up

"We've covered a lot of ground here, & it will take a while for it all to fully sink in. Perform the playground exercise over & over until it makes sense. It is important to get hood understanding of basic file manipulation commands & wildcards. Feel free to expand on the playground exercise by adding more files & directories, using wildcards to specify files for various operations. The concept of links is a little confusing at 1st, but take the time to learn how they work. They can be a real lifesaver!" – Shotts, 2019, p. 82

1.5 Working with Commands

"Up to this point, we have seen a series of mysterious commands, each with its own options & arguments. In this chapter, we will attempt to remove some of that mystery & even create our own commands. The command introduced in this chapter are as follows: • `type`: Indicate how a command name is interpreted. • `which`: Display which executable program will be executed. • `help`: Get help for shell builtins. • `man`: Display a command's manual page. • `apropos`: Display a list of appropriate commands. • `info`: Display a command's info entry. • `whatis`: Display 1-line manual page descriptions. • `alias`: Create an alias for a command." – Shotts, 2019, p. 83

1.5.1 What Exactly Are Commands?

"A command can be 1 of 4 different things.

- **An executable program** like all those files we saw in `/usr/bin`. Within this category, programs can be *compiled binaries* such as programs written in C & C++, or programs written in *scripting languages* such as the shell, Perl, Python, Ruby, & so on.
- **A command built into the shell itself.** `bash` supports a number of commands internally called *shell builtins*. The `cd` command, e.g., is a shell builtin.
- **A shell function.** Shell functions are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment & writing shell functions in later chapters, but for now, just be aware that they exist.
- **An alias.** Aliases are commands that we can define ourselves, built from other commands." – Shotts, 2019, pp. 83–84

1.5.2 Identifying Commands

"It is often useful to know exactly which of the 4 kinds of commands is being used, & Linux provides a couple of ways to find out." – Shotts, 2019, p. 84

1.5.2.1 `type` – Display a Command's Type

"The `type` command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this: `type` *`command`* where `command` is the name of the command you want to examine. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to ‘ls --color=tty’
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for 3 different commands. Notice the one for `ls` (taken from a Fedora system) & how the `ls` command is actually an alias for the `ls` command with the `--color=tty` option added. Now we know why the output from `ls` is displayed in color!" – Shotts, 2019, pp. 84–85

1.5.2.2 `which` – Display an Executable's Location

"Sometimes there is > 1 version of an executable program installed on a system. While this is not common on desktop systems, it's not unusual on larger servers. To determine the exact location of a given executable, the `which` command is used.

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` works only for executable programs, not builtins or aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin – e.g., `cd` – we either get no response or get an error message.

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in
(/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games)
```

This result is a fancy way of saying "command not found"." – Shotts, 2019, p. 85

1.5.3 Getting a Command's Documentation

"With this knowledge of what a command is, we can now search for the documentation available for each kind of command." – Shotts, 2019, p. 85

1.5.3.1 `help` – Get Help for Shell Builtins

"`bash` has a built-in help facility available for each of the shell builtins. To use it, type `help` followed by the name of the shell builtin. Here's an example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the value of the
    HOME shell variable. If DIR is "-", it is converted to $OLDPWD.

    The variable CDPATH defines the search path for the directory containing
    DIR.  Alternative directory names in CDPATH are separated by a colon (:).
    A null directory name is the same as the current directory.  If DIR begins
    with a slash (/), then CDPATH is not used.

    If the directory is not found, and the shell option 'cdable_vars' is set,
    the word is assumed to be  a variable name.  If that variable has a value,
    its value is used for DIR.

    Options:
      -L force symbolic links to be followed: resolve symbolic
     links in DIR after processing instances of '..'
      -P use the physical directory structure without following
     symbolic links: resolve symbolic links in DIR before
     processing instances of '..'
      -e if the -P option is supplied, and the current working
     directory cannot be determined successfully, exit with
     a non-zero status
      -@ on systems that support it, present a file with extended
     attributes as a directory containing the file attributes

    The default is to follow symbolic links, as if '-L' were specified.
    '..' is processed by removing the immediately previous pathname component
    back to a slash or the beginning of DIR.

    Exit Status:
    Returns 0 if the directory is changed, and if $PWD is set successfully when
    -P is used; non-zero otherwise.
```

A note on notation. When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
    cd [-L|[-P[-e]]] [dir]
```

This notation says that the command `cd` may be followed optionally by either a `-L` or a `-P` & further, if the `-P` option is specified the `-e` option may also be included followed by the optional argument `dir`.

While the output of `help` for the `cd` commands is concise & accurate, it is by no means tutorial & as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there." – Shotts, 2019, pp. 85–87

1.5.3.2 `--help` – Display Usage Information

"Many executable programs support a `--help` option that displays a description of the command's supported syntax & options. E.g.:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as needed
  -v, --verbose     print a message for each created directory
  -Z                   set SELinux security context of each created directory
                         to the default type
      --context[=CTX]  like -Z, or if CTX is specified then set the SELinux
                         or SMACK security context to CTX
      --help     display this help and exit
      --version  output version information and exit

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/mkdir>
or available locally via: info '(coreutils) mkdir invocation'
```

Some programs don't support the `--help` option, but try it anyway. Often it results in an error message that will reveal the same usage information." – Shotts, 2019, p. 87

1.5.3.3 `man` – Display a Program's Manual Page

"Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them. It is used like this: `man` ***`program`*** where `program` is the name of the command to view.

Man pages vary somewhat in format but generally contain the following: • A title (the page's name). • A synopsis of the command's syntax. • A description of the command's purpose. • A listing & description of each of the command's options.

Man pages, however, do not usually include examples & are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the `ls` command.

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The "manual" that `man` displays is broken into sections & covers not only user commands but also system administration commands, programming interfaces, file formats, & more. Table 5-1 describes the layout of the manual. Table 5-1: Man Page Organization. Section : Contents.

- 1: User commands
- 2: Programming interfaces for kernel system calls.
- 3: Programming interfaces to the C library.
- 4: Special files such as device nodes & drivers.
- 5: File formats.
- 6: Games & amusements such as screen savers.
- 7: Miscellaneous.
- 8: System administration commands.

Sometimes we need to refer to a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the 1st instance of a match, probably in section 1. To specify a section number, we use `man` like this: `man section search_term`. Here's an example:

```
[me@linuxbox ~]$ man 5 passwd
```

This will display the man page describing the file format of the `/etc/passwd` file." – Shotts, 2019, pp. 87–89

1.5.3.4 `apropos` – Display Appropriate Commands

"It is also possible to search the list of man pages for possible matches based on a search term. It's crude but sometimes helpful. Here is an example of a search for man pages using the search term *partition*:

```
[me@linuxbox ~]$ apropos partition
addpart (8)          - tell the kernel about the existence of a partition
cfdisk (8)           - display or manipulate a disk partition table
cgdisk (8)           - Curses-based GUID partition table (GPT) manipulator
delpart (8)          - tell the kernel to forget about a partition
fdisk (8)            - manipulate disk partition table
fixparts (8)         - MBR partition table repair utility
gdisk (8)            - Interactive GUID partition table (GPT) manipulator
gparted (8)          - GNOME Partition Editor for manipulating disk partitions.
parted (8)           - a partition manipulation program
partprobe (8)        - inform the OS of partition table changes
partx (8)            - tell the kernel about the presence and numbering of on...
repart.d (5)         - Partition Definition Files for Automatic Boot-Time Rep...
resizepart (8)       - tell the kernel about the new size of a partition
sfdisk (8)           - display or manipulate a disk partition table
sgdisk (8)           - Command-line GUID partition table (GPT) manipulator fo...
systemd-gpt-auto-generator (8) - Generator for automatically discovering and ...
systemd-repart (8)   - Automatically grow and add partitions
systemd-repart.service (8) - Automatically grow and add partitions
```

The 1st field in each line of output is the name of the man page, & the 2nd field shows the section. Note that the `man` command with the `-k` option performs the same function as `apropos`." – Shotts, 2019, pp. 89–90

1.5.3.5 `whatis` – Display 1-line Manual Page Descriptions

"The `whatis` program displays the name & a 1-line description of a man page matching a specified keyword.

```
[me@linuxbox ~]$ whatis ls
ls                  (1) - list directory contents
```

The most brutal man page of them all. As we have seen, the manual pages supplied with Linux & other Unix-like systems are intended as reference documentation & not as tutorials. Many man pages are hard to read, but I think the grand prize for difficulty has got to go to the man page for `bash`. As I was doing research for this book, I gave the `bash` man page careful review to ensure that I was covering most of its topics. When printed, it's > 80 pages long & extremely dense, & its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate & concise, as well as being extremely complete. So check it out if you dare & look forward to the day when you can read it & it all makes sense." – Shotts, 2019, p. 90

1.5.3.6 `info` – Display a Program's Info Entry

1.6 Redirection

1.7 Seeing the World as the Shell Sees It

1.8 Advanced Keyboard Tricks

1.9 Permissions

1.10 Processes

Part II: Configuration & The Environment

1.11 The Environment

1.12 A Gentle Introduction to vi

1.13 Customizing the Prompt

Part III: Common Tasks & Essential Tools

1.14 Package Management

1.15 Storage Media

1.16 Networking

1.17 Searching for Files

1.18 Archiving & Backup

1.19 Regular Expressions

1.20 Text Processing

1.21 Formatting Output

1.22 Printing

1.23 Compiling Programs

Part IV: Writing Shell Scripts

1.24 Writing Your 1st Script

1.25 Starting a Project

1.26 Top-Down Design

1.27 Flow Control: Branching with `if`

1.28 Reading Keyboard Input

1.29 Flow Control: Looping with `while/until`

1.30 Troubleshooting

1.31 Flow Control: Branching with `case`

1.32 Positional Parameters

1.33 Flow Control: Looping with `for`

1.34 Strings & Numbers

1.35 Arrays

1.36 Exotica

Chapter 2

Miscellaneous

2.1 Calibre

Official website: https://calibre-ebook.com/. See also, e.g., Wikipedia/Calibre (software). Enable Dark Mode in Calibre:

```
$ sudo nano /etc/profile.d/calibre.sh
```

```
export CALIBRE_USE_SYSTEM_THEME=1
$ sudo service gdm restart
```

& further customization. `[inserting]`

Bibliography

Shotts, William (2019). “The Linux Command Line: A Complete Introduction”. In: p. 640.