The Linux Users' Guide

All you need to know to start using Linux, a free Unix clone. This manual covers the basic Unix commands, as well as the more specific Linux ones. This manual is meant for the beginning Unix user, although it may be useful for more experienced users for reference purposes.

These conventions should be obvious, but we'll include them here for the pedantic.

Bold Used to mark **new concepts**, **WARNINGS**, and **keywords** in a language.

italics Used for *emphasis* in text, and occasionally for quotes or introductions at the beginning of a section. Also used to indicate commands for the user to type when showing screen interaction (see below).

slanted Used to mark **meta-variables** in the text, especially in representations of the command line. For example,

`ls -l` *foo*

where *foo* would "stand for" a filename, such as `/bin/cp`.

`Typewriter` Used to represent screen interaction, as in

```
$ ls -l /bin/cp
-rwxr-xr-x  1 root    wheel    12104 Sep 25 15:53 /bin/cp
```

Also used for code examples, whether it is "C" code, a shell script, or something else, and to display general files, such as configuration files. When necessary for clarity's sake, these examples or figures will be enclosed in thin boxes.

$\boxed{\text{Key}}$ Represents a key to press. You will often see it in this form:

Press $\boxed{\text{return}}$ to continue.

◇ A diamond in the margin, like a black diamond on a ski hill, marks "danger" or "caution." Read paragraphs marked this way carefully.

Acknowledgements

The author would like to thank the following people for their invaluable help either with LINUX itself, or in writing *The* LINUX *Users' Guide*:

Linus Torvalds for providing something to write this manual about.

Karl Fogel has given me much help with writing my LINUX documentation and wrote Chapter 7 and Chapter 8.

Maurizio Codogno wrote much of Chapter 9.

The `fortune` program for supplying me with many of the wonderful quotes that start each chapter. They cheer me up, if no one else.

Contents

Chapter 1

Introduction

How much does it cost to entice a dope-smoking Unix system guru to Dayton?
Brian Boyle, *Unix World*'s First Annual Salary Survey

1.1 Who Should Read This Book

Are you someone who should read this book? Do you want to learn Unix? Have you just gotten LINUX from somewhere, installed it, and want to know what to do next?

If you have this book, the answer to these questions is probably "yes." Anyone who has LINUX, the free Unix clone written by Linus Torvalds, on their PC but doesn't know what to do next should read this book. In this book, we'll cover most of the basic Unix commands, as well as some of the more advanced ones. We'll also talk about GNU Emacs, a powerful editor, and several other large Unix applications.

1.1.1 What You Should Have Done Before Reading This Book

This book relies on a few things that the author can't control. First of all, this book, as do most Unix books, assumes that you have access to a Unix system. More importantly, this Unix system should be an Intel PC running LINUX. This requirement isn't necessary, but when versions of Unix differ, I'll be doing what LINUX expects—nothing else.

LINUX is available in many forms, called distributions. It is hoped that you've found a complete distribution such as SoftLanding Linux Systems or the MCC-Interim release and have installed it. There are differences between the various distributions of LINUX, but for the most part they're small and unimportant. (Occasionally in this book you'll find places that seem a little off. If you do, it's probably because you're using a different distribution than I am. The author is interested in all such cases.)

If you're the superuser (the maintainer, the installer) of the system, you also should have created a normal user account for yourself. Please consult the installation manual(s) for this information. If you aren't the superuser, you should have obtained an account from the superuser. Also, you should have some time and patience. Learning LINUX isn't easy—most people find learning the Macintosh Operating System is easier. However, many people feel that LINUX is more powerful.

Also, this book assumes that you are moderately familiar with some computer terms. Although this requirement isn't necessary, it makes reading the book easier. You should know about computer terms such as 'program' and 'execution'. If you don't, you might want to get someone's help with learning Unix.

1.2 How to Avoid Reading This Book

The best way to learn about almost any computer program is at your computer. Most people find that reading a book without using the program isn't very beneficial. Thus, the best way to learn Unix and LINUX is by using them. Use them for everything you can. Experiment. Don't be afraid—it's *always* possible to mess things up, but you can always reinstall. For better or for worse, though, Unix isn't as intuitively obvious as some other operating systems. Thus, you will probably end up reading at least the first couple of chapters in this book.

1.3 How to Read This Book

The suggested way of learning Unix is to read a little, then to play a little. I suggest the first X chapters—after them, the rest of the book can be read in almost any order. Keep playing until you're comfortable with the concepts, and then start skipping around in the book. You'll find a variety of topics are covered, some of which you might find interesting. After a while, you should feel confident enough to start using commands without knowing what they should do. This is a good thing.

What most people regard as Unix is the Unix shell, a special program that interprets commands. In practice, this is a fine way of looking at things, but you should be aware that Unix really consists of many more things, or much less. (Depending on how you look at it.) This book tells you about how to use the shell, programs that Unix usually comes with, and some programs Unix doesn't always come with.

The current chapter is a meta-chapter—it discusses this book and how to apply this book to getting work done. The other chapters contain:

Chapter 2 discusses where Unix and LINUX came from, and where they might be going. It also talks about the Free Software Foundation and the GNU Project.

Chapter 3 talks about how to start and stop using your computer, and what happens at these times. Much of it deals with topics not needed for using LINUX, but still quite useful and interesting.

Chapter 4 introduces the Unix shell. This is where people actually do work, and run programs. It talks about the basic programs and commands you must know to use Unix.

1.4 Linux Documentation

This book, *The* Linux *Users' Guide*, is intended for the Unix beginner. Luckily, the Linux Documentation Project is also writing books for the more experienced users.

1.4.1 Other Linux Books

The other books include *Installation and Getting Started*, a guide on how to aquire and install Linux, *The* Linux *System Adminstrator's Guide*, how to organize and maintain a Linux system, and *The* Linux *Kernel Hackers' Guide*, a book about how to modify Linux. *The* Linux *Network Administration Guide* talks about how to install, configure, and use a network connection.

1.4.2 What's the Linux Documentation Project?

Like almost everything associated with Linux, the Linux Documentation Project is a collection of people working across the globe. Originally organized by Lars Wirzenius, the Project is now coordinated by Matt Welsh with help from Michael K. Johnson.

It is hoped that the Linux Documentation Project will supply books that will meet all the needs of documenting Linux at some point in time. Please tell us if we've suceeded or what we should improve on.

1.5 Operating Systems

An operating system's primary purpose is to support programs that actually do the work. An operating system is not the tool that does the work, it's the tool that *supports* the work. It's tempting to just want to modify the operating system for its own sake, and if you feel this way often, I suggest you find a copy of *The* Linux *Kernel Hackers' Guide*.

Operating systems (OS, for short) can be simple and minimalist, like DOS, or big and complex, like OS/2 or VMS.[1] Unix tries to be a middle ground. While it supplies more resources and does more then early operating systems, it doesn't try to do *everything* like some other operating systems.

The original design philosophy[2] for Unix was to distribute functionality into small parts, the programs. That way, you can relatively easily achieve new functionality and new features by combining the small parts (programs) in new ways. And if new utilities appear (and they do), you can integrate them into your old toolbox. Unfortunately, programs grow larger and more feature-packed on Unix as well these days, but some of the flexibility, interoperability is there to stay. When I write this document, for example, I'm using these programs actively; `fvwm` to manage my "windows", `emacs` to edit the text, LaTeX to format it, `xdvi` to preview it, `dvips` to prepare it for printing and

[1] Apologies to DOS, OS/2, and VMS users. I've used all three, and each have their good points.

[2] Was mostly determined by the type of hardware Unix was to run on. By sheer accident, the resulting operating system turned out to be very useful on other hardware.

then `lpr` to print it. If I got a new, better dvi previewer tommorow, I could use it instead of `xdvi` without changing the rest of my setup.

The key part of an operating system is called the "kernel." In many operating systems, like Unix, OS/2, or VMS, the kernel supplies functions for running programs to use, and schedules them to be run. It basically says program A can get so much time, program B can get this much time, etc. One school of thought says that kernels should be very small, and not supply a lot of resources, depending on programs to pick up the work. This allows the kernel to be small and fast, but may make programs bigger. Kernels designed like this are called micro-kernels. Another group of people believe that bigger kernels are better and make more efficent operating systems. Some versions of Unix are designed like this, including LINUX. One ironic thing to note here is that micro-kernels aren't necessarily smaller then macro-kernels—it's the philosophy that separates the two.

When you're using an operating system, you want to minimize the amount of work you put into getting your job done. Unix supplies many tools that can help you, but only if you know what these tools do. Spending an hour trying to get something to work and then finally giving up isn't very productive. Hopefully, you already know how to use the correct tools—that way, you won't use the hammer to try and tighten a screw.

The moral of the story? Don't change the way you work to suit the operating system, but be aware of the operating system. Don't wish for a tool that isn't in your box, use the tools in your box to make a new one.

Chapter 2

What's Unix, anyway?

Ken Thompson has an automobile which he helped design. Unlike most automobiles, it has neither speedometer, nor gas gage, nor any of the numerous idiot lights which plague the modern driver. Rather, if the driver makes any mistake, a giant "?" lights up in the center of the dashboard. "The experienced driver," he says, "will usually know what's wrong."

2.1 Unix History

In 1965, Bell Telephone Laboratories (Bell Labs, a division of AT&T) was working with General Electric and Project MAC of MIT to write an operating system called Multics. To make a long story slightly shorter, Bell Labs decided the project wasn't going anywhere and broke out of the group. This, however, left Bell Labs without a good operating system.

Ken Thompson and Dennis Ritchie decided to sketch out an operating system that would meet Bell Labs' needs. When Thompson needed a development environment (1970) to run on a PDP-7, he implemented their ideas. As a pun on Multics, Brian Kernighan gave the system the name UNIX.

Later, Dennis Ritchie invented the "C" programming language. In 1973, UNIX was rewritten in C, which would have a major impact later on. In 1977, UNIX was moved to a new machine, away from the PDP machines it had run on previously. This was aided by the fact UNIX was written in C.

Unix was slow to catch on outside of academic institutions but soon was popular with businesses as well. The Unix of today is different from the Unix of 1970. It has two major versions: System V, from Unix System Laboratories (USL), a subsiderary of Novell[1], and BSD, Berkeley Software Distribution. The USL version is now up to its forth release, or SVR4[2], while BSD's latest version is 4.4. However, there are many different versions of Unix besides these two. Most versions of Unix are

[1] It was recently sold to Novell. Previously, USL was owned by AT&T.

[2] System five, revision four.

developed by software companies and derive from one of the two groupings. Recently, the versions of Unix that are actually used incorporate features from both of them.

USL is a company that was 'spun off' from AT&T, and has taken over the maintenance of UNIX since it stopped being a research item. Unix now is much more commericial than it once was, and the licenses cost much more.

Please note the difference between Unix and UNIX. When I say "Unix" I am talking about Unix versions in generally, whether or not USL is involved in them. "UNIX" is the current version of Unix from USL. The distinction is because UNIX is a trademark of Unix System Laboratories.

Current versions of UNIX for Intel PCs cost between $500 and $2000.

2.2 Linux History

Linux was written by Linus Torvalds, and has been improved by countless numbers of people around the world. It is a clone, written entirely from scratch, of the Unix operating system. Neither USL, nor the University of California, Berkeley, was involved in writing Linux. One of the more interesting facts about Linux is that development simulataneously occurs around the world. People from Austrialia to Finland contributed to Linux, and hopefully will continue to contribute.

Linux began with a project to explore the 386 chip. One of Linus's earlier projects was a program that would switch between printing `AAAA` and `BBBB`. This later evolved to Linux.

Linux has been copyrighted under the terms of the GNU General Public License (GPL). This is a license written by the Free Software Foundation (FSF) that is designed to prevent people from restricting the distribution of software. In brief it says that although you can charge as much as you'd like for giving a copy away, you can't prevent the person you sold it to from giving it away for free. It also means that the source code[3] must also be available. This is useful for programmers. The license also says that anyone who modifies the program must also make his version freely redistributable.

Linux supports most of the popular Unix software, including The X Window System. This is a rather large program from MIT allowing computers to create graphical windows, and is used on many different Unix platforms. Linux is mostly System V, mostly BSD compatible and mostly POSIX-1 (a document trying to standardize operating systems) compliant. Linux probably complies with much of POSIX-2, another document from the IEEE to standardize operating systems. It's a mix of all three standards: BSD, System V, and POSIX.

Many of the utilities included with Linux distributions are from the Free Software Foundation and are part of GNU Project. The GNU Project is an effort to write a portable, advanced operating system that will look a lot like Unix. "Portable" means that it will run on a variety of machines, not just Intel PCs, Macintoshes, or whatever. Linux is not easily ported (moved to another computer architechure) because it was written only with the 80386 in mind.

Of course, Torvalds isn't the only big name in Linux's development. The following people also deserve to be recognized:

[3]The instructions that people write, as distinct from zeros and ones.

H. J. Lu has maintained gcc and the LINUX C Library, two items needed for programming.

Of course, I must have missed people in the above list. Sincere thanks and apologies go out to anyone not mentioned here—there must be dozens if not hundreds of you!

2.2.1 LINUX Now

Currently, LINUX is not yet at version 1.0, but as of this writing, is on version "0.99p10". (That's from July 3rd, 1993!) However, it is evolving fast and people expect version 1.0 *before the next ice age*! That's a real good sign, and many people in the LINUX community are looking forward to it.

The items changing the fastest in LINUX right now are TCP/IP support[4] and bug fixes. LINUX is a large system and unfortunately contains bugs which are found and then fixed. Although some people still experience bugs regularly, it is normally because of non-standard or faulty hardware; bugs that effect everyone are few and far between.

Of course, those are just the kernel bugs. Bugs can be present in almost every facet of the system, and inexperienced users have trouble seperating different programs from each other. For instance, a problem might arise that all the characters are some type of gibberish—is it a bug or a "feature"? Surprisingly, this is a feature—the gibberish is caused by certain control sequences that somehow appeared/footnoteHowever, because you have all the source code, you can easily disable this particular escape sequence and recompile the kernel.. Hopefully, this book will help you to tell the different situations apart.

2.2.2 Trivial LINUX Matters

Before we embark on our long voyage, let's get the ultra-important out of the way.

Question: Just how do you pronounce LINUX?

Answer: According to Linus, it should be pronounced with a short *ih* sound, like prInt, mIn-Imal, etc. LINUX should rhyme with Minix, another Unix clone. It should *not* be pronounced like (American pronounciation of) the "Peanuts" character, Linus, but rather *LIH-nucks*. And the *u* is sharp as in rule, not soft as in ducks.

2.2.3 Commercial Software in LINUX

For better or for worse, there is now commercial software available for LINUX. Although it isn't a fancy word processing application, Motif is a package that must be payed for, and the source isn't given out. Motif is a user interface for The X Window System that vaguely resembles MicroSoft Windows.

For any readers interested in the legalities of LINUX, this is allowed by the LINUX license. While the GNU General Public License (reproduced in Appendix A) covers the LINUX kernel, the GNU

[4]That's a form of networking. More on that later.

Library General Public License (reproduced in Appendix B) covers most of the computer code applications depend on.

Please note that those two documents are copyright notices, and not licenses to use. They do *not* regulate how you may use the software, merely under what circumstances you can copy it and any derivative works. Also, copyright notices are enforced by lawsuits by the copyright holders, either the Free Software Foundation or Linus Torvalds. In general, this means you can't go wrong if you obey the spirit of what they're asking—they probably won't sue you and all will be well. (Unless the rights get sold.) It's also a good idea not to think up schemes to get around these two copyrights—it's almost definitely possible, but merely causes grief to all parties involved.

Chapter 3

Getting Started

This login session: \$13.99, but for you \$11.88.

3.1 Starting to Use Your Computer

You may have previous experience with MS-DOS or other single user operating systems, such as OS/2 or the Macintosh. In these operating systems, you didn't have to identify yourself to the computer before using it; it was assumed that you were the only user of the system and could access everything. Well, Unix is a multi-user operating system—not only can more than one person use it at a time, different people are treated differently.[1]

To tell people apart, Unix needs a user to identify him or herself[2] by a process called **logging in**. You see, when you first turn on the computer, several things happen. Since this guide is geared towards LINUX, I'll tell you what happens during the LINUX boot-up sequence.

Please note that if you're using LINUX on some type of computer besides an Intel PC, some things in this chapter won't apply to you. Mostly, they'll be in Sections 3.1.1 and 3.1.2. (Some parts of Section 3.1.2 will pertain.)

3.1.1 Power to the Computer

The first thing that happens when you turn an Intel PC on is that the BIOS executes. BIOS stands for **B**asic **I**nput/**O**utput **S**ystem. It's a program permenantly stored in the computer on read-only chips, normally. For our purposes, the BIOS can never be changed. It performs some minimal tests, and then looks for a floppy disk in the first disk drive. If it finds one, it looks for a "boot sector" on that disk, and starts executing code from it, if any. If there is a disk, but no boot sector, the BIOS will print a message like:

[1]Discrimination? Perhaps. You decide.

[2]From here on in this book, I shall be using the mascaline pronouns to identify all people. This is the standard English convention, and people shouldn't take it as a statement that only men can use computers.

```
Non-system disk or disk error
```

Removing the disk and pressing a key will cause the boot process to continue.

If there isn't a floppy disk in the drive, the BIOS looks for a master boot record (MBR) on the hard disk. It will start executing the code found there, which loads the operating system. On Linux systems, LILO, the **LI**nux **LO**ader, can occupy the MBR position, and will load Linux. For now, we'll assume that happens and that Linux starts to load. (Your particular distribution may handle booting from the hard disk differently. Check with the documentation included in that distribution. Another good reference is the LILO documentation, [1].)

3.1.2 Linux Takes Over

After the BIOS passes control to LILO, LILO passes control to Linux. (This is under normal circumstances. It is also possible for LILO to call DOS or some other PC operating system.) The first thing that Linux does once it starts executing is changes to protected mode. The 80386[3] CPU that controls your computer has two modes (for our purposes) called real mode and protected mode. DOS runs in real mode, as does the BIOS. However, for more advances operating systems, it is necessary to run in protected mode. Therefore, when Linux boots, it discardes the BIOS.

Linux then looks at the type of hardware it's running on. It wants to know what type of hard disks you have, whether or not you have a bus mouse, whether or not you're on a network, and other bits of trivia like that. Linux can't remember things between boots, so it has to ask these questions each time it starts up. Luckily, it isn't asking *you* these questions—it is asking the hardware! The part of Linux asking these questions is the kernel. During boot-up, the Linux kernel will print variations on several messages. You can read about the messages in Section 3.3.

The kernel merely manages other programs, so once it is satisfied everything is okay, it must start another program to do anything useful. The program the kernel starts is called `init` . (Notice the difference in font. Things in that font are usually the names of programs, files, directories, or other computer related items.) After the kernel starts `init`, it never starts another program. The kernel becomes a manager and a provider, not a active program.

So to see what the computer is doing after the kernel boots up, we'll have to examine `init`. `init` goes through a complicated startup sequence that isn't the same for all computers. For Linux there are many versions of `init`, and each does things its own way. It also matters whether your computer is on a network, or what distribution you used to install Linux. Some things that might happen once `init` is started:

- The file systems might be checked. What is a file system, you might ask? A file system is the layout of files on the hard disk. It let's Unix know which parts of the disk are already used, and which aren't. Unfortunately, due to various factors such as power losses, what the file system information thinks is going on in the rest of the disk and the actually layout of the rest of the disk are in conflict. A special program, called `fsck`, can find these situations and hopefully correct them.

[3] When I refer to the 80386, I am also talking about the 80486 unless I specifically say so. Also, I'll be abbreviating 80386 as 386.

- Special routing programs for networks are run.
- Temporary files left by some programs may be deleted.
- The system clock can be correctly updated. This is trickier then one might think, since Unix, by default, wants the time in GMT and your CMOS clock, a battery powered clock in your computer, is probably set on local time.

After `init` is finished with its duties at boot-up, it goes on to its regularly scheduled activities. `init` can be called the parent of all processes on a Unix system. A process is simple a running program; since any one program can be running more than once, there can be two or more processes for any particular program. (Processes can also be sub-programs, but that isn't important right now.) There are as many processes operating as there are programs.

In Unix, a process, an instance of a program, is created by a system call, a service provided by the kernel, called `fork`. `init` `fork`s a couple of processes, which in turn `fork` some of their own. On your Linux system, what `init` runs are several instances of a program called `getty`. `getty` will be covered in...

3.1.3 The User Acts

`getty` performs a fairly simple function. It merely has to prompt the user to log in. This process, knowing as "logging in", is Unix's way of knowing that users are authorized to use the system. It asks for an account name and password. An account name is normally similar to your regular name; you should have already received one from your system administrator, or created your own if you are the system administrator. (Information on doing this should be available in *Installation and Getting Started* or *The* Linux *System Adminstrator's Guide*.)

When `getty` first starts up, it displays a message of greeting or some such idea. It is up to the system adminstrator what message, if any, it displays. You should see, after all the boot-up procedures are done, something like the following:

```
Welcome to the mousehouse. Please, have some cheese.

mousehouse login:
```

This is, of course, your invitation to **login**. Throughout this manual, we'll be using the fictional (or not so fictional, depending on your machine) user `larry`. Whenever you see `larry`, you should be substituting your own account name. Account names are usually based on real names; bigger, more serious Unix systems will have accounts using the user's last name, or some combination of first and last name, or even some numbers. Possible accounts for Larry Greenfield might be: `larry`, `greenfie`, `lgreenfi`, `lg19`.

`mousehouse` is, by the way, the "name" of the machine I'm working on. It is possible that when you installed Linux, you were prompted for some very witty name. It isn't very important, but whenever it comes up, I'll be using `mousehouse` or, rarely, `lionsden`.

After entering `larry`, I'm faced with the following:

```
mousehouse login: larry
Password:
```

What Linux is asking for is your **password**. When you type in your password, you won't be able to see what you type. Type carefully: it is possible to delete, but you won't be able to see what you are editing. Don't type too slowly if people are watching—they'll be able to learn your password. If you mistype, you'll be presented with another chance to login.

If you've typed your login name and password correctly, a short message will appear, called the message of the day. This could say anything—the system adminstrator decides what it should be. After that, a **prompt** appears. A prompt is just that, something prompting you for the next command to give the system. It should look like this:

```
/home/larry#
```

You'll be seeing a lot of this. Commands will be introduced in the next chapter.

3.2 Leaving the Computer

◇ Do not just turn off the computer! You risk losing valuable data!

Unlike most versions of DOS, it's a bad thing to just hit the power switch when you're done using the computer. It is also bad to reboot the machine (with the reset button) without first taking proper precautions. Linux, in order to improve performance, caches the disk. This means it temporarily stores part of the permanent storage in RAM. The idea of what Linux thinks the disk should be and what the disk actually contains is syncronized every 30 seconds. In order to turn off or reboot the computer, you'll have to go through a procedure telling it to stop caching disk information.

If you're done with the computer, but are logged in (you've entered a username and password), first you must logout. To do so, enter the command `logout`. All commands are sent by pressing the key marked "Enter" or "Return". Until you hit enter, nothing will happen, and you can delete what you've done and start over.

```
/home/larry# logout

Welcome to the mousehouse. Please, have some cheese.

mousehouse login:
```

Now another user can login.

3.2.1 Turning the Computer Off

If this is a single user system, you might want to turn the computer off when you're done with it. To do so, you'll have to log into a special account called `root`. The `root` account is the system adminstrator's account and can access any file on the system. If you're going to turn the computer off, get the password from the system adminstrator. (In a single user system, that's *you*! Make sure you know the default `root` password.) Login as `root`:

```
mousehouse login: root
Password:

Linux, version 0.99pl10.
/# shutdown now

************ GET THE SHUTDOWN MESSAGE CORRECT ***********
```

The command `shutdown now` prepares the system to be reset or turned off. After it displays a message saying it is safe to, do either. You must go through this procedure, however. You risk losing work that you've done if you don't.

3.3 Kernel Messages

The messages printed by the kernel vary from machine to machine, and from kernel version to version. The version of LINUX that is discussed in *this* section is "0.99.10". (Please note that this is a big book, and LINUX develops quickly. Versions in other sections might be different. Usually, this distinction is unimportant.)

3.3.1 Starting Messages

When LINUX first starts up, it writes many messages to the screen which you might not be able to see. LINUX maintains a special file, called `/proc/kmsg`, which stores all these messages for later viewing, and I've included a sample startup sequence here.

- The first thing LINUX does is decides what type of video card and screen you have, so it can pick a good font size. (The smaller the font, the more that can fit on the screen on any one time.) LINUX may ask you if you want a special font, or it might have had a choice compiled in.[4]

  ```
  Console: colour EGA+ 80x25, 8 virtual consoles Serial driver version
  ```

  In this example, the machine owner decided he wanted the standard, large font at compile time. Also, note the misspelling of the word "color." Linus evidentally learned the wrong version of English.

- LINUX has now switched to protected mode, and the serial driver has started to ask questions about the hardware. A driver is a part of the kernel that controls a device, usually a peripheral.

[4] "Compiled" is the process by which a computer program that a human writes gets translated into something the computer understands. A feature that has been "compiled in" has been included in the program.

```
Serial driver version 3.95 with no serial options enabled
tty00 at 0x03f8 (irq = 4) is a 16450
tty01 at 0x02f8 (irq = 3) is a 16450
tty02 at 0x03e8 (irq = 4) is a 16450
```

Here, it found 3 serial ports. A serial port is the equivalent of DOS `COM` ports, and is a device normally used with modems and mice. Actually, when the serial driver first starts up, it only finds out how many ports there are—the other information are defaults.

What it is trying to say is that serial port 0 (`COM1`) has an electronic address of `0x03f8`. When it interrupts the kernel, usually to say that it has data, it uses IRQ 4. An IRQ is another means of a peripheral talking to the software. Each serial port also has a controller chip. The usual one for a port to have is a 16450; other values possible are 8250 and 16550. The differences are beyond the scope of this book.

- Next comes the parallel port driver. A parallel port is normally connected to a printer, and the names for the parallel ports (in LINUX) start with `lp`. `lp` stands for **L**ine **P**rinter, although it could be a laser printer.

```
lp_init: lp0 exists (0), using polling driver
```

That message says it has found one parallel port, and is using the standard driver for it.

- The LINUX kernel also tells you a little about memory usage:

```
Memory: 7296k/8192k available (384k kernel code, 384k reserved, 128k data)
```

This said that the machine had 8 megabytes of memory. Some of this memory was reserved for the kernel—just the operating system. The rest of it could be used by programs. Please note that the 8 megabytes the kernel talks about here is very fast memory called "RAM" for **r**andom **a**ccess **m**emory. This memory is lost once you turn your machine off.

The other type of "memory" is general called a hard disk. It's like a large floppy disk permenantly in your computer—the contents stay around even when the power is off.

- The kernel now moves onto looking at your floppy drives. In this example, the machine has two drives. In DOS, drive "A" is a 5 1/4 inch drive, and drive "B" is a 3 1/2 inch drive. LINUX calls drive "A" `fd0`, and drive "B" `fd1`.

```
Floppy drive(s): fd0 is 1.2M, fd1 is 1.44M
floppy: FDC version 0x90
```

- Now LINUX moves onto less needed things, such as network cards. The following should be described in *The* LINUX *Networking Guide*, and is beyond the scope of this document.[5]

```
SLIP: version 0.7.5 (4 channels): OK
plip.c:v0.04 Mar 19 1993 Donald Becker (becker@super.org)
plip0: using parallel port at 0x3bc, IRQ 5.
plip1: using parallel port at 0x378, IRQ 7.
plip2: using parallel port at 0x278, IRQ 2.
8390.c:v0.99-10 5/28/93 for 0.99.6+ Donald Becker (becker@super.org)
```

[5]This may change in latter 'versions' of this book.

```
WD80x3 ethercard probe at 0x280: FF FF FF FF FF FF not found (0x7f8).
3c503 probe at 0x280:  not found.
8390 ethercard probe at 0x280 failed.
HP-LAN ethercard probe at 0x280: not found (nothing there).
No ethernet device found.
dl0: D-Link pocket adapter: probe failed at 0x378.
```

- The next message you normally won't see as the machine boots up. LINUX supports a FPU, a floating point unit. This is a special chip (or part of a chip, in the case of a 80486DX CPU) that performs arithmetic dealing with non-whole numbers. Some of these chips are bad, and when LINUX tries to identify these chips, the machine "crashes". That is to say, the machine stops functioning. If this happens, you'll see:

  ```
  You have a bad 386/387 coupling.
  ```

  Otherwise, you'll see:

  ```
  Math coprocessor using exception 16 error reporting.
  ```

  if you're using a 486DX. If you are using a 386 with a 387, you'll see:

  ```
  Math coprocessor using irq13 error reporting.
  ```

  If you don't have any type of math coprocessor at all, you'll see:

  ```
  What will they see?
  ```

- The kernel also scans for any hard disks you might have. If it finds any (and it should) it'll look at what partitions you have on them. A partition is a logical seperation on a drive that is used to keep operating systems from interfering with each other. In this example, the computer had one hard disk (`hda`) with four partitions.

  ```
  Partition check:
    hda: hda1 hda2 hda3 hda4
  ```

- Finally, LINUX **mounts** the root partition. The root partition is the disk partition where the LINUX operating system "lives". When LINUX "mounts" this partition, it is making the partition available for use by the user.

  ```
  VFS: Mounted root (ext filesystem).
  ```

3.3.2 Running Messages

The LINUX kernel occasionally sends messages to your screen. The following is a list of some of these messages and what they mean. Frequently, these messages indicate something is wrong. Some of these messages are **critical**, which means the operating system (and all your programs!) stops working. When these messages occur, you should write them down and what you where doing at the time, and send them to Linus. You should see Section 10.2.2.

Luckily, some of these messages are merely informational—hopefully, you'll see them more often!

-
  ```
  Adding Swap: 10556k swap-space
  lp0 on fire
  ******** OBVIOUSLY INCOMPLETE
  ```

Chapter 4

The Unix Shell

A UNIX saleslady, Lenore,
Enjoys work, but she likes the beach more.
    She found a good way
    To combine work and play:
She sells C shells by the seashore.

4.1 Unix Commands

When you first log into a Unix system, you are presented with something that looks like the following:

```
/home/larry#
```

This is called a **prompt**. As its name would suggest, it is prompting you to enter a command. Every Unix command is a sequence of letters, numbers, and characters. There are no spaces, however. Thus, valid Unix commands include `mail`, `cat`, and `CMU_is_Number-5`. Some characters aren't allowed—that's covered later. Unix is also **case-sensitive**. This means that `cat` and `Cat` are different commands.

Case sensitivity is a very personal thing. Some operating systems, such as OS/2 or Windows NT are case preserving, but not case sensitive. In practice, Unix rarely uses the different cases. It is unusual to have a situation where `cat` and `Cat` are different commands.

The prompt is displayed by a special program called the **shell**. The MS-DOS shell is called `COMMAND.COM`, and is very simple compared to most Unix shells. Shells accept commands, and run those commands. They can also be programmed in their own language, and programs written in that language are called "shell scripts".

There are two major types of shells in Unix, Bourne shells, and C shells. Bourne shells are named after their inventor, Steven Bourne. There are many implementations of this shell, and all those specific shell programs are called Bourne shells. Another class of shells, C shells (originally implemented by Bill Joy), are also common. Traditionally, Bourne shells have been used for compatibility, and C shells have been used for interactive use.

Linux comes with a Bourne shell called `bash`, written by the Free Software Foundation. `bash` stands for **B**ourne **A**gain **Sh**ell, one of the many bad puns in Unix. It is an advanced Bourne shell, with many features commonly found in C shells, and is the default.

When you first login, the prompt is displayed by `bash`, and you are running your first Unix program, the `bash` shell.

4.1.1 A Typical Unix Command

The first command to know is `cat`. To use it, type `cat`, and then return:

```
/home/larry# cat
```

If you now have a cursor on a line by itself, you've done the correct thing. There are several variances you could have typed—some would work, some wouldn't.

- If you misspelled `cat`, you would have seen

  ```
  /home/larry# ct
  ct: command not found
  /home/larry#
  ```

  Thus, the shell informs you that it couldn't find a program named "`ct`" and gives you another prompt to work with. Remember, Unix is case sensitive: `CAT` is a misspelling.

- You could have also placed whitespace before the command, like this:[1]

  ```
  /home/larry#␣␣␣␣␣cat
  ```

  This produces the correct result and runs the `cat` program.

- You might also press return on a line by itself. Go right ahead—it does absolutely nothing.

I assume you are now in `cat`. Hopefully, you're wondering what it is doing. For all you hopefuls, no, it is not a game. `cat` is a useful utility that won't seem useful at first. Type anything, and hit return. What you should have seen is:

```
/home/larry# cat
Help! I'm stuck in a Linux program!
Help!  I'm stuck in a Linux program!
```

(The *slanted* text indicates what the user types.) What `cat` seems to do is echo the text right back at yourself. This is useful at times, but isn't right now. So let's get out of this program and move onto commands that have more obvious benefits.

[1]The ' ' indicates that the user typed a space.

To end many Unix commands, type Ctrl-d[2]. Ctrl-d is the end-of-file character, or EOF for short. Alternatively, it stands for end-of-text, depending on what book you read. I'll refer to it as an end-of-file. It is a control character that tells Unix programs that you (or another program) is done entering data. When `cat` sees you aren't typing anything else, it terminates.

For a similar idea, try the program `sort`. As its name indicates, it is a sorting program. If you type a couple of lines, then press Ctrl-d, it will output those lines in a sorted order. By the way, these types of programs are called **filters**, because they take in text, filter it, and output the text slightly differently. (Well, `cat` is a very basic filter and doesn't change the input.) We will talk more about filters later.

4.2 Helping Yourself

The `man` command displays reference pages for the command/footnoteOr system call, subroutine, file format etc. you spesify. For example;

```
$ man cat

cat(1)                                                      cat(1)

NAME
  cat - Concatenates or displays files

SYNOPSIS
  cat [-benstuvAET] [--number] [--number-nonblank] [--squeeze-blank]
  [--show-nonprinting] [--show-ends] [--show-tabs] [--show-all]
  [--help] [--version] [file...]

DESCRIPTION
  This manual page documents the GNU version of cat ...
```

There's about one full page of information about `cat`. Try it. Don't expect to understand it, though. It assumes quite some Unix knowledge. When you've read the page, there's probably a little black block at the bottom of your screen, reading `--more--`, `Line 1` or something similar. This is the more-prompt, and you'll learn to love it.

Instead of just letting the text scroll away, `man` stops at the end of each page, waiting for you to decide what to do now. If you just want to go on, press Space and you'll advance a page. If you want to exit (quit) the manual page you are reading, just press q You'll be back at the shell prompt, and it'll be waiting for you to enter a new command.

There's also a keyword function in `man`. If you for example type `man -k signal`, you'll get a listing of all commands, system calls, and other documented parts of Unix that have the word `signal` in their short description. This can be very useful when you're looking for a tool to do something, but you don't know it's name—or if it even exists!

[2]Hold down the key labeled "Ctrl" and press "d", then let go.

4.3 Storing Information

Filters are very useful once you are an experienced user, but they have one small problem. How do you store the information? Surely you aren't expected to type everything in each time you are going to use the program! Of course not. Unix provides **files** and **directories**.

A directory is like a folder: it contains pieces of paper, or files. A large folder can even hold other folders—directories can be inside directories. In Unix, the collection of directories and files is called the file system. Initially, the file system consists of one directory, called the "root" directory. Inside this directory, there are more directories, and inside those directories are files and yet more directories.[3]

Each file and each directory has a name. It has both a short name, which can be the same as another file or directory somewhere else on the system, and a long name which is unique. A short name for a file could be `joe`, while it's "full name" would be `/home/larry/joe`. The full name is usually called the **path**. The path can be decode into a sequence of directories. For example, here is how `/home/larry/joe` is read:

```
/home/larry/joe
```

First, we are in the root directory.
 This signifies the directory called `home`. It is inside the root directory.
  This is the directory `larry`, which is inside `home`.
   `joe` is inside `larry`. A path could refer to either a directory or a filename, so `joe` could be either. All the items *before* the short name must be directories.

An easy way of visualizing this is a tree diagram. To see a diagram of a typical LINUX system, look at Figure 4.1. Please note that this diagram isn't complete—a full LINUX system has over 8000 files!—and shows only some of the standard directories. Thus, there may be some directories in that diagram that aren't on your system, and your system almost certainly has directories not listed there.

4.3.1 Looking at Directories with `ls`

Now that you know that files and directories exist, there must be some way of manipulating them. Indeed there is. The command `ls` is one of the more important ones. It **lis**ts files. If you try `ls` as a command, you'll see:

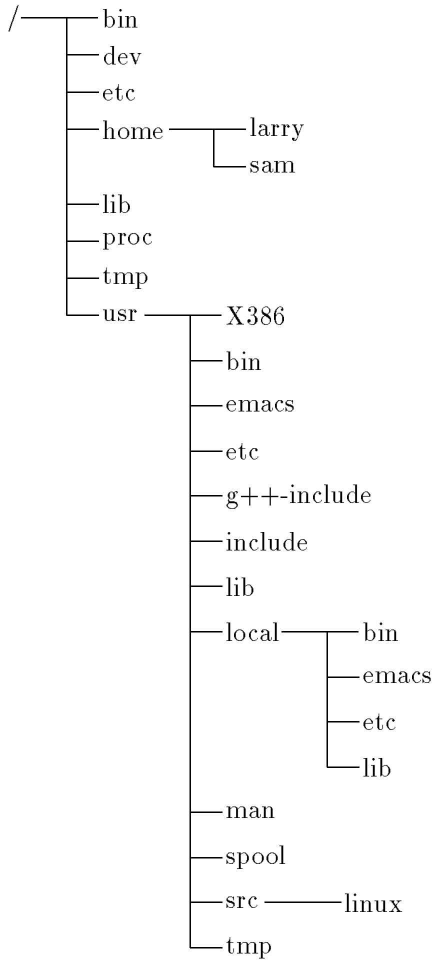
```
/home/larry# ls
/home/larry#
```

That's right, you'll see nothing. Unix is intensionally terse: it gives you nothing, not even "no files" if there aren't any files. Thus, the lack of output was `ls`'s way of saying it didn't find any files.

But I just said there could be 8000 or more files lying around: where are they? You've run into the concept of a "current" directory. You can see in your prompt that your current directory is

[3]There may or may not be a limit to how "deep" the file system can go. You can easily have directories 10 levels down.

Figure 4.1: A typical (abridged) Unix directory tree.



/home/larry, where you don't have any files. If you want a list of files of a more active directory, try the root directory:

```
/home/larry# ls /
bin      etc      install  mnt      root     user     var
dev      home     lib      proc     tmp      usr      vmlinux
/home/larry#
```

In the above command, "`ls /`", the directory is a **parameter**. The first word of the command is the command name, and anything after it is a parameter. Some commands have special parameters called **options** or **switches**. To see this, try:

```
/home/larry# ls -F /
bin/      etc/      install/  mnt/      root/     user/     var@
dev/      home/     lib/      proc/     tmp/      usr/      vmlinux
/home/larry#
```

The `-F` is an option that lets you see which ones are directories, which ones are special files, which are programs, and which are normal files. Anything with a slash is a directory. We'll talk more about `ls`'s features later. It's a surprisingly complex program!

Now, there are two lessons to be learned here. First, you should learn what `ls` does. Try a few other directories that are shown in Figure 4.1, and see what they contain. Naturally, some will be empty, and some will have many, many files in them. I suggest you try `ls` both with and without the `-F` option. For example, `ls /usr/local` looks like:

```
/home/larry# ls /usr/local
archives  bin       emacs     etc       ka9q      lib       tcl
/home/larry#
```

The second lesson is more general. Many Unix commands are like `ls`. They have options, which are generally one character after a dash, and they have parameters. Occasionally, the line between the two isn't so clear.

Unlike `ls`, some commands require certain parameters and/or options. To show what commands generally look like, we'll use the following form:

`ls` [-arF] [*directory*]

That's a command template and you'll see it whenever a new command is introduced. Anything contained in brackets ("[" and "]") is optional: it doesn't have to be there. Anything *slanted* should usually be changed before trying the command. You'll rarely have a directory *named* `directory`.

4.3.2 The Current Directory and `cd`

Using directories would be cumbersome if you had to type the full path each time you wanted to access a directory. Instead, Unix shells have a feature called the "current" or "present" or "working" directory. Your setup most likely displays your directory in your prompt: `/home/larry`. If it doesn't, try the command `pwd`, for **p**resent **w**orking **d**irectory.

```
mousehouse>pwd
/home/larry
mousehouse>
```

As you can see, `pwd` tells you your current directory[4]—a very simple command. Most commands act, by default, on the current directory, such as `ls`. We can change our current directory using `cd`. For instance, try:

```
/home/larry# cd /home
/home# ls -F
larry/    sam/      shutdown/  steve/    user1/
/home#
```

[4]You'll see all the terms in this book: present working directory, current directory, or working directory. I prefer "current directory", although at times the other forms will be used for stylistic purposes.

A generic template looks like:

`cd` [*directory*]

If you omit the *directory*, you're returned to your home, or original, directory. Otherwise, `cd` will change you to the specified directory. For instance:

```
/home# cd
/home/larry# cd /
/# cd home
/home# cd /usr
/usr# cd local/bin
/usr/local/bin#
```

As you can see, `cd` allows you to give either absolute or relative pathnames. An "absolute" path starts with / and specifies all the directories before the one you wanted. A "relative" path is in relation to your current directory. In the above example, when I was in `/usr`, I made a relative move to `local/bin`—`local` is a directory under `usr`, and `bin` is a directory under `local`!

There are two directories used *only* for relative pathnames: "`.`" and "`..`". . The directory "`.`" refers to the current directory and "`..`" is the parent directory. These are "shortcut" directories. They exist in *every* directory, but don't really fit the "folder in a folder" concept. Even the root directory has a parent directory—it's its own parent!

The file `./chapter-1` would be the file called `chapter-1` in the current directory. Occasionally, you need to put the "`./`" for some commands to work, although this is rare. In most cases, `./chapter-1` and `chapter-1` will be identical.

The directory "`..`" is most useful in backing up:

```
/usr/local/bin# cd ..
/usr/local# ls -F
archives/  bin/       emacs@     etc/       ka9q/      lib/       tcl@
/usr/local# ls -F ../src
cweb/      linux/     xmris/
/usr/local#
```

In this example, I changed to the parent directory using `cd ..`, and I listed the directory `/usr/src` from `/usr/local` using `../src`. Note that if I was in `/home/larry`, typing `ls -F ../src` wouldn't do me any good!

One other shortcut for lazy users: the directory / is your home directory:

```
/usr/local# ls -F ~/
/usr/local#
```

You can see at a glance that there isn't anything in your home directory! Actually, / will become more useful as we learn more about how to manipulate files.

4.3.3 Using mkdir to Create Your Own Directories

Creating your own directories is extremely simple under Unix, and can be a useful organizational tool. To create a new directory, use the command **mkdir**. Of course, **mkdir** stands for **m**a**k**e **dir**ectory.

`mkdir` *directory*

Let's do a small example to see how this works:

```
/home/larry# ls -F
/home/larry# mkdir report-1993
/home/larry# ls -F
report-1993/
/home/larry# cd report-1993
/home/larry/report-1993#
```

mkdir can actually take more than one parameter, and you can specify either the full pathname or a relative pathname; **report-1993** in the above example is a relative pathname.

```
/home/larry/report-1993# mkdir /home/larry/report-1993/chap1 ~/report-1993/chap2
/home/larry/report-1993# ls -F
chap1/  chap2/
/home/larry/report-1993#
```

Finally, there is the opposite of **mkdir**, **rmdir** for **rem**ove **dir**ectory. **rmdir** works exactly as you think it should work:

`rmdir` *directory*

An example of **rmdir** is:

```
/home/larry/report-1993# rmdir chap1 chap3
rmdir: chap3: No such file or directory
/home/larry/report-1993# ls -F
chap2/
/home/larry/report-1993# cd ..
/home/larry# rmdir report-1993
rmdir: report-1993: Directory not empty
/home/larry#
```

As you can see, **rmdir** will refuse to remove a non-existant directory, as well as a directory that has anything in it. (Remember, **report-1993** has a subdirectory, **chap2**, in it!) There is one more interesting thing to think about **rmdir**: what happens if you try to remove your current directory? Let's find out:

```
/home/larry# cd report-1993
/home/larry/report-1993# ls -F
chap2/
/home/larry/report-1993# rmdir chap2
/home/larry/report-1993# rmdir .
rmdir: .: Operation not permitted
/home/larry/report-1993#
```

Another situation you might want to consider is what happens if you try to remove the parent of your current directory. In fact, this isn't even a problem: the parent of your current directory isn't empty, so it can't be removed!

4.4 Moving Information

All of these fancy directories are very nice, but they really don't help unless you have some place to store you data. The Unix Gods saw this problem, and they fixed it by giving the users "files". We will learn more about creating and editing files in the next few chapters.

The primary commands for manipulating files under Unix are `cp`, `mv`, and `rm`. Respectively, they stand for copy, move, and remove.

4.4.1 `cp` Like a Monk

`cp` is a very useful utility under Unix, and extremely powerful. It enables one person to copy more information in a second than a fourteenth century monk could do in a year.

◇ Be careful with `cp` if you don't have a lot of disk space. No one wants to see `Error saving--disk full`. `cp` can also overwrite existing files—I'll talk more about that danger later.

The first parameter to `cp` is the file to copy—the last is where to copy it. You can copy to either a different filename, or a different directory. Let's try some examples:

```
/home/larry# ls -F /etc/rc
/etc/rc
/home/larry# cp /etc/rc .
/home/larry# ls -F
rc
/home/larry# cp rc frog
/home/larry# ls -F
frog  rc
/home/larry#
```

The first `cp` command I ran took the file `/etc/rc`, which contains commands that the Unix system runs on boot-up, and copied it to my home directory. `cp` doesn't delete the source file, so I didn't do anything that could harm the system. So two copies of `/etc/rc` exist on my system now, both named `rc`, but one is in the directory `/etc` and one is in `/home/larry`.

Then I created a *third* copy of `/etc/rc` when I typed `cp rc frog`—the three copies are now: `/etc/rc`, `/home/larry/rc` and `/home/larry/frog`. The contents of these three files are the same, even if the names aren't.

The above example illustrates two uses of the command `cp`. Are there any others? Let's take a look:

- `cp` can copy files between directories if the first parameter is a file and the second parameter is a directory.
- It can copy a file and change it's name if both parameters are file names. Here is one danger of `cp`. If I typed `cp /etc/rc /etc/passwd`, `cp` would normally create a new file with the contents identical to `rc` and name it `passwd`. However, if `/etc/passwd` already existed, `cp` would destroy the old file without giving you a chance to save it!
- Let's look at another example of `cp`:

```
/home/larry# ls -F
frog  rc
/home/larry# mkdir rc_version
/home/larry# cp frog rc rc_version
/home/larry# ls -F
frog        rc        rc_version/
/home/larry# ls -F rc_version
frog  rc
/home/larry#
```

  How did I just use `cp`? Evidentally, `cp` can take *more* than two parameters. What the above command did is copied all the files listed (`frog` and `rc`) and placed them in the `rc_version` directory. In fact, `cp` can take any number of parameters, and interprets the first $n-1$ parameters to be files to copy, and the n^{th} parameter as what directory to copy them too.

◇ You cannot rename files when you copy more than one at a time—they always keep their short name. This leads to an interesting question. What if I type `cp frog rc toad`, where `frog` and `rc` exist and `toad` isn't a directory? Try it and see.

One last thing in this section—how can you show the parameters that `cp` takes? After all, the parameters can mean two different things. When that happens, we'll have two different lines:

`cp` *source destination-name*
`cp` *file1 file2 . . . fileN destination-directory*

4.4.2 Pruning Back with `rm`

Now that we've learned how to create millions of files with `cp` (and believe me, you'll find new ways to create more files soon), it may be useful to learn how to delete them. Actually, it's very simple: the command you're looking for is `rm`, and it works just like you'd expect.

Any file that's a parameter to `rm` gets deleted:

`rm` *file1 file2 . . . fileN*

For example:

```
/home/larry# ls -F
frog         rc        rc_version/
/home/larry# rm frog toad rc
rm: toad: No such file or directory
/home/larry# ls -F
rc_version/
/home/larry#
```

As you can see, `rm` is extremely unfriendly. Not only does it not ask you for confirmation, but it will also delete things even if the whole command line wasn't correct. This could actually be dangerous. Consider the difference between these two commands:

```
/home/larry# ls -F
toad  frog/
/home/larry# ls -F frog
toad
/home/larry# rm frog/toad
/home/larry#
```

and this

```
/home/larry# rm frog toad
rm: frog is a directory
/home/larry# ls -F
frog/
/home/larry#
```

◇ As you can see, the difference of *one* character made a world of difference in the outcome of the command. It is vital that you check your command lines before hitting $\boxed{\text{return}}$!

4.4.3 A Forklift Can Be Very Handy

Finally, the other file command you should be aware of is `mv`. `mv` looks a lot like `cp`, except that it deletes the original file after copying it. Thus, it's a lot like using `cp` and `rm` together. Let's take a look at what we can do:

```
/home/larry# cp /etc/rc .
/home/larry# ls -F
rc
/home/larry# mv rc frog
/home/larry# ls -F
frog
/home/larry# mkdir report
/home/larry# mv frog report
/home/larry# ls -F
```

```
report/
/home/larry# ls -F report
frog
/home/larry#
```

As you can see, **`mv`** will rename a file if the second parameter is a file. If the second parameter is a directory, **`mv`** will move the file to the new directory, keeping it's shortname the same:

> **`mv`** *old-name new-name*
> **`mv`** *file1 file2 ... fileN new-directory*

◇ You should be very careful with **`mv`**—it doesn't check to see if the file already exists, and will remove any old file in its way. For instance, if I had a file named **`frog`** already in my directory **`report`**, the command **`mv frog report`** would delete the file **`/report/frog`** and replace it with **`/frog`**.

In fact, there is one way to make **`rm`**, **`cp`** and **`mv`** ask you before deleting files. The **`-i`** option. If you use an **alias**, you can make the shell do **`rm -i`** automatically when you type **`rm`**. You'll learn more about this later.

Chapter 5

Working with Unix

```
better !pout !cry
better watchout
lpr why
santa claus <north pole >town

cat /etc/passwd >list
ncheck list
ncheck list
cat list | grep naughty >nogiftlist
cat list | grep nice >giftlist
santa claus <north pole > town

who | grep sleeping
who | grep awake
who | egrep 'bad|good'
for (goodness sake) {
        be good
}
```

Unix is a powerful system for those who know how to harness its power. In this chapter, I'll try to describe various ways to use Unix's shell, `bash`, more efficently.

5.1 Wildcards

In the previous chapter, you learned about the file maintence commands `cp`, `mv`, and `rm`. Occasionally, you want to deal with more than one file at once—in fact, you might want to deal with many files at once. For instance, you might want to copy all the files beginning with `data` into a directory called `/backup`. You could do this by either running many `cp` commands, or you could list every file on one command line. Both of these methods would take a long time, however, and you have a large chance of making an error.

A better way of doing that task is to type:

```
/home/larry/report# ls -F
1993-1          1994-1          data1           data5
1993-2          data-new        data2
/home/larry/report# mkdir ~/backup
/home/larry/report# cp data* ~/backup
/home/larry/report# ls -F ~/backup
data-new        data1           data2           data5
/home/larry/report#
```

As you can see, the asterix told `cp` to take all of the files beginning with `data` and copy them to `/backup`. Can you guess what `cp d*w  /backup` would have done?

5.1.1 What *Really* Happens?

Good question. Actually, there are a couple of special characters intercepted by the shell, `bash`. The character "`*`", an asterix, says "replace this word with all the files that will fit this specification". So, the command `cp data*  /backup`, like the one above, gets changed to `cp data-new data1 data2 data5  /backup` before it gets run.

To illustrate this, let me introduce a new command, `echo`. `echo` is an extremely simple command; it echoes back, or prints out, any parameters. Thus:

```
/home/larry# echo Hello!
Hello!
/home/larry# echo How are you?
How are you?
/home/larry# cd report
/home/larry/report# ls -F
1993-1          1994-1          data1           data5
1993-2          data-new        data2
/home/larry/report# echo 199*
1993-1 1993-2 1994-1
/home/larry/report# echo *4*
1994-1
/home/larry/report# echo *2*
1993-2 data2
/home/larry/report#
```

As you can see, the shell expands the wildcard and passes all of the files to the program you tell it to run. This raises an interesting question: what happens if there are *no* files that meet the wildcard specification? Try `echo /rc/fr*og` and see what happens...The shell will pass the wildcard specification verbatim to the program.

5.1.2 The Question Mark

In addition to the asterix, the shell also interprets a question mark as a special character. A question mark will match one, and only one character. For instance, `ls /etc/??` will display all two letter files in the the `/etc` directory.

5.2 Time Saving with `bash`

5.2.1 Command-Line Editing

Occasionally, you've typed a long command to **`bash`** and, before you hit return, notice that there was a spelling mistake early in the line. You could just delete all the way back and retype everything you need to, but that takes much too much effort! Instead, you can use the arrow keys to move back there, delete the bad character or two, and type the correct information.

There are many special keys to help you edit your command line, most of them similar to the commands used in GNU Emacs. For instance, $\boxed{\text{C-t}}$ flips two adjacent characters.[1] You'll be able to find most of the commands in the chapter on Emacs, Chapter 7.

5.2.2 Command and File Completion

Another feature of **`bash`** is automatic completion of your command lines. For instance, let's look at the following example of a typical **`cp`** command:

```
/home/larry# ls -F
this-is-a-long-file
/home/larry# cp this-is-a-long-file shorter
/home/larry# ls -F
shorter              this-is-a-long-file
/home/larry#
```

It's a big pain to have to type every letter of **`this-is-a-long-file`** whenever you try to access it. So, create **`this-is-a-long-file`** by copying **`/etc/rc`** to it[2]. Now, we're going to do the above **`cp`** command very quickly and with a smaller chance of mistyping.

Instead of typing the whole filename, type **`cp th`** and press and release the $\boxed{\text{Tab}}$. Like magic, the rest of the filename shows up on the command line, and you can type in **`shorter`**. Unfortunately, **`bash`** cannot read your thoughts, and you'll have to type all of **`shorter`**.

When you type $\boxed{\text{Tab}}$, **`bash`** looks at what you've typed and looks for a file that starts like that. For instance, if I type **`/usr/bin/ema`** and then hit $\boxed{\text{Tab}}$, **`bash`** will find **`/usr/bin/emacs`** since that's the only file that begins **`/usr/bin/ema`** on my system. However, if I type **`/usr/bin/ld`**

[1] $\boxed{\text{C-t}}$ means hold down the key labeled "Ctrl", then press the "t" key. Then release the "Ctrl" key.

[2] `cp /etc/rc this-is-a-long-file`

and hit $\boxed{\text{Tab}}$, `bash` beeps at me. That's because three files, `/usr/bin/ld`, `/usr/bin/ldd`, and `/usr/bin/ld86` start `/usr/bin/ld` on my system.

If you try a completion and `bash` beeps, you can immediately hit $\boxed{\text{Tab}}$ again to get a list of all the files your start matches so far. That way, if you aren't sure of the exact spelling of your file, you can start it and scan a much smaller list of files.

5.3 The Standard Input and The Standard Output

Let's try to tackle a simple problem: getting a listing of the `/usr/bin` directory. If all we do is `ls /usr/bin`, some of the files scroll off the top of the screen. How can we see all of the files?

5.3.1 Unix Concepts

The Unix operating system makes it very easy for programs to use the terminal. When a program writes something to your screen, it is using something called **standard output**. Standard output, abbreviated as stdout, is how the program writes things to a user. The name for what you tell a program is **standard input** (stdin). It's possible for a program to communicate with the user without using standard input or output, but very rare—all of the commands we have covered so far use stdin and stdout.

For example, the `ls` command prints the list of the directories to standard output, which is normally "connected" to your terminal. An interactive command, such as your shell, `bash`, reads your commands from standard input.

It is also possible for a program to write to **standard error**, since it is very easy to make standard output point somewhere besides your terminal. Standard error, stderr, is almost always connected to a terminal so an actual human will read the message.

In this section, we're going to examine three ways of fiddling with the standard input and output: input redirection, output redirection, and pipes.

5.3.2 Output Redirection

A very important feature of Unix is the ability to **redirect** output. This allows you, instead of viewing the results of a command, to save it in a file or send it directly to a printer. For instance, to redirect the output of the command `ls /usr/bin`, we place a `>` sign at the end of the line, and say what file we want the output to be put in:

```
/home/larry# ls
/home/larry# ls -F /usr/bin > listing
/home/larry# ls
listing
/home/larry#
```

As you can see, instead of writing the names of all the files, the command created a totally new file in your home directory. Let's try to take a look at this file using the command `cat`. If you think back, you'll remember `cat` was a fairly useless command that copied what you typed (the standard input) to the terminal (the standard output). `cat` can also print a file to the standard output if you list the file as a parameter to `cat`:

```
/home/larry# cat listing
...
/home/larry#
```

The exact output of the command `ls /usr/bin` appeared in the contents of `listing`. All well and good, although it didn't solve the original problem.[3]

However, `cat` does do some interesting things when it's output is redirected. What does the command `cat listing > newfile` do? Normally, the `> newfile` says "take all the output of the command and put it in `newfile`." The output of the command `cat listing` is the file `listing`. So we've invented a new (and not so efficent) method of copying files.

How about the command `cat > fox`? `cat` by itself reads in each line typed at the terminal (standard input) and prints it right back out (standard output) until it reads Ctrl-d. In this case, standard output has been redirected into the file `fox`. Now `cat` is serving as a rudimentary editor:

```
/home/larry# cat > fox
The quick brown fox jumps over the lazy dog.
press Ctrl-d
```

We've now created the file `fox` that contains the sentence "The quick brown fox jumps over the lazy dog." One last use of the versitile `cat` command is to con`cat`enate files together. `cat` will print out every file it was given as a parameter, one after another. So the command `cat listing fox` will print out the directory listing of `/usr/bin`, and then it will print out our silly sentence. Thus, the command `cat listing fox > listandfox` will create a new file containing the contents of both `listing` and `fox`.

5.3.3 Input Redirection

Like redirecting standard output, it is also possible to redirect standard input. Instead of a program reading from your keyboard, it will read from a file. Since input redirection is related to output redirection, it seems natural to make the special character for input redirection be `<`. It too, is used after the command you wish to run.

This is generally useful if you have a data file and a command that expects input from standard input. Most commands also let you specify a file to operate on, so `<` isn't used as much in day-to-day operations as other techniques.

[3]For impatient readers, the command you might want to try is `more`. However, there's still a bit more to talk about before we get there.

5.3.4 Solution: The Pipe

Many Unix commands produce a large amount of information. For instance, it is not uncommon for a command like `ls /usr/bin` to produce more output than you can see on your screen. In order for you to be able to see all of the information that a command like `ls /usr/bin`, it's necessary to use another Unix command, called `more`.[4] `more` will pause once every screenful of information. For instance, `move < /etc/rc` will display the file `/etc/rc` just like `cat /etc/rc` would, except that `more` will let you read it.[5]

However, that doesn't help the problem that `ls /usr/bin` displays more information than you can see. `more < ls /usr/bin` won't work—input redirection only works with files, not commands! You *could* do this:

```
/home/larry# ls /usr/bin > temp-ls
/home/larry# more temp-ls
...
/home/larry# rm temp-ls
```

However, Unix supplies a much cleaner way of doing that. You can just use the command `ls /usr/bin | more`. The character "`|`" indicates a **pipe**. Like a water pipe, a Unix pipe controls flow. Instead of water, we're controlling the flow of information!

A useful tool with pipes are programs called **filters**. A filter is a program that reads the standard input, changes it in some way, and outputs to standard output. `more` is a filter—it reads the data that it gets from standard input and displays it to standard output one screen at a time, letting you read the file.

Other filters include the programs `cat`, `sort`, `head`, and `tail`. For instance, if you wanted to read only the first ten lines of the output from `ls`, you could use `ls /usr/bin | head`.

5.4 Multitasking

5.4.1 The Basics

Job control refers to the ability to put processes (another word for programs, essentially) in the background and bring them to the foreground again. That is to say, you want to be able to make something run while you go and do other things, but have it be there again when you want to tell it something or stop it. In Unix, the main tool for job control is the shell—it will keep track of jobs for you, if you learn how to speak its language.

The two most important words in that language are `fg`, for "foreground", and `bg`, for "background". To find out how they work, use the command `yes` at a prompt.

[4] `more` is named because that's the prompt it originally displayed: `--more--`. In many versions of Linux the `more` command is identical to a more advanced command that does all that `more` can do and more. Its name? `less`, of course!

[5] `more` also allows the command `more /etc/rc`.

```
/home/larry# yes
```

This will have the startling effect of running a long column of `y`'s down the left hand side of your screen, faster than you can follow. (There are good reasons for this strange command to exist, but we won't go into them now). To get them to stop, you'd normally type ctrl-C to kill it, but instead you should type ctrl-Z this time. It appears to have stopped, but there will be a message before your prompt, looking more or less like this:

```
[1]+  Stopped                  yes
```

It means that the process `yes` has been *suspended* in the background. You can get it running again by typing `fg` at the prompt, which will put it into the foreground again. If you wish, you can do other things first, while it's suspended. Try a few `ls`'s or something before you put it back in the foreground.

Once it's returned to the foreground, the `y`'s will start coming again, as fast as before. You do not need to worry that while you had it suspended it was "storing up" more `y`'s to send to the screen: when a program is suspended the whole program doesn't run until you bring it back to life. (And you can type ctrl-C to kill it for good, once you've seen enough).

Let's pick apart that message we got from the shell:

```
[1]+  Stopped                yes
```

The number in brackets is the **job number** of this job, and will be used when we need to refer to it specifically. (Naturally, since job control is all about running multiple processes, we need some way to tell one from another). The `+` following it tells us that this is the "current job" — that is, the one most recently moved from the foreground to the background. If you were to type `fg`, you would put the job with the `+` in the foreground again. (More on that later, when we discuss running multiple jobs at once). The word `Stopped` means that the job is "stopped". The job isn't dead, but it isn't running right now. Linux has saved it in a special suspended state, ready to jump back into the action should anyone request it. Finally, the `yes` is the name that was typed on the command line to start the program.

Before we go on, let's kill this job and start it again in a different way. The command is named `kill` and can be used in the following way:

```
/home/larry# kill %1
[1]+ Stopped                 yes
```

That message about it being "stopped" again is misleading. To find out whether it's still alive (that is, either running or frozen in a suspended state), type `jobs`:

```
/home/larry# jobs
[1]+ Terminated                 yes
```

There you have it—the job has been terminated! (It's possible that the `jobs` command showed nothing at all, which just means that there are no jobs running in the background. If you just killed a job, and typing `jobs` shows nothing, then you know the kill was successful. Usually it will tell you the job was "terminated".)

Now, start `yes` running again, like this:

```
/home/larry# yes > /dev/null
```

If you read the section about input and output redirection, you know that this is sending the output of `yes` into the special file `/dev/null`. `/dev/null` is a black hole that eats any output sent to it (you can imagine that stream of `y`'s coming out the back of your computer and drilling a hole in the wall, if that makes you happy).

After typing this, you will not get your prompt back, but you will not see that column of `y`'s either. Although output is being sent into `/dev/null`, the job is still running in the foreground. As usual, you can suspend it by hitting ctrl-Z. Do that now to get the prompt back.

```
/home/larry# yes > /dev/null
["yes" is running, and if we type ctrl-z right now, we'll suspend
  it and get the prompt back.  Imagine that I just did that...]
[1]+  Stopped                 yes >/dev/null

/home/larry#
```

Hmm... is there any way to get it to actually *run* in the background, while still leaving us the prompt for interactive work? Of course there is, otherwise I wouldn't have asked. The command to do that is **bg**:

```
/home/larry# bg
[1]+ yes >/dev/null  &
/home/larry#
```

Now, you'll have to trust me on this one: after you typed `bg`, `yes > /dev/null` began to run again, but this time in the background. In fact, if you do things at the prompt, like `ls` and stuff, you might notice that your machine has been slowed down a little bit (piping a steady stream of single letters out the back of the machine does take some work, after all!) Other than that, however, there are no effects. You can do anything you want at the prompt, and `yes` will happily continue to sending its output into the black hole.

There are now two different ways you can kill it: with the `kill` command you just learned, or by putting the job in the foreground again and hitting it with an interrupt (ctrl-C). Let's try the second way, just to understand the relationship between `fg` and `bg` a little better;

```
/home/larry# fg
yes >/dev/null

[now it's in the foreground again.  Imagine that I hit ctrl-C
to terminate it]

/home/larry#
```

There, it's gone. Now, start up a few jobs running in simultaneously, like this:

```
/home/larry# yes  > /dev/null &
[1] 1024
/home/larry# yes | sort > /dev/null &
[2] 1026
/home/larry# yes | uniq > /dev/null
[and here, type ctrl-Z to suspend it, please]

[3]+  Stopped                 yes | uniq >/dev/null
```

The first thing you might notice about those commands is the trailing `&` at the end of the first two. Putting an `&` after a command tells the shell to start in running in the background right from the very beginning. (It's just a way to avoid having to start the program, type ctrl-Z, and then type `bg`.) So, we started those two commands running in the background. The third is suspended and inactive at the moment. You should notice that the machine has definitely become slower now, as the two running ones require significant amounts of CPU time.

Each one told you it's job number. The first two also showed you their **Process IDentification numbers**, or `PID`'s, immediately following the job number. The PID's are normally not something you need to know, but occasionally come in handy.

Let's kill the second one, since I think it's making your machine slow. You could just type `kill %2`, but that would be too easy. Instead, do this:

```
/home/larry # fg %2
[and then hit ctrl-C to kill it]
```

As this demonstrates, `fg` takes parameters beginning with `%` as well. In fact, you could just have typed this:

```
/home/larry # %2
[and then hit ctrl-C to kill it]
```

This works because the shell automatically interprets a job number as a request to put that job in the foreground. It can tell job numbers from other numbers by the preceding `%`. Now type `jobs` to see which jobs are left running:

```
/home/larry # jobs
[1]-  Running                 yes >/dev/null  &
[3]+  Stopped                 yes | uniq >/dev/null
```

That pretty much says it all. The `-` means that job number 1 is second in line to be put in the foreground, if you just type `fg` without giving it any parameters. However, you can get to it by naming it, if you wish:

```
/home/larry # fg %1
yes >/dev/null
[now type ctrl-Z to suspend it]
```

```
[1]+  Stopped                yes >/dev/null
```

Having changed to job number 1 and then suspending it has also changed the priorities of all your jobs. You can see this with the `jobs` command:

```
/home/larry # jobs
[1]+  Stopped                yes >/dev/null
[3]-  Stopped                yes | uniq >/dev/null
```

Now they are both stopped (because both were suspended with ctrl-Z), and number 1 is next in line to come to the foreground by default. This is because you put it in the foreground manually, and then suspended it. The `+` always refers to the most recent job that was suspended from the foreground. You can start it running again:

```
/home/larry # bg
[1]+ yes >/dev/null  &
/home/larry# jobs
[1]-  Running                yes >/dev/null
[3]+  Stopped                yes | uniq >/dev/null
```

Notice that now it is running, and the other job has moved back up in line and has the `+`.

Well, enough of that. Kill them all so you can get your machine back:

```
/home/larry# kill %1
/home/larry# kill %3
```

You should see various messages about termination of jobs – nothing dies quietly, it seems. To summarize what you should know about job control now:

ctrl-z	DOS equiv.: Hah! DOS doesn't have real job control... This key combination usually causes a program to suspend, although a few programs ignore it. Once suspended, the job can be run in the background or killed. Parameters: none — it's not really a command, just a signal.
fg	DOS equiv.: none whatsoever. Maybe someday... This is a shell-builtin command that returns a job to the foreground. To find out which one this is by default, type `jobs` and look for the one with the `+`. Parameters: job number (optional – defaults to the one with `+`).
&	When an `&` is added to the end of the command line, it tells the command to run in the background automatically. This job is then subject to all the usual methods of job control detailed here.
bg	This is a shell-builtin command that causes a suspended job to run in the background. To find out which one this is by default, type `jobs` and look for the one

	with the `+`. One way to think of `bg` is that it's really just `fg &`! Parameters: job number (optional – defaults to the one with `+`).
kill	This is a shell-builtin command that causes a background job, either suspended or running, to terminate. You should always specify the job number or PID, and if you are using job numbers, remember to precede them with a `%`. Parameters: job number (preceded by `%`) or PID (no `%` necessary).
jobs	This shell command just lists information about the jobs currently running or suspending. Sometimes it also tells you about ones that have just exited or been terminated.
ctrl-c	This is the generic interrupt character. Usually, if you type it while a program is running in the foreground, it will kill the program (sometimes it takes a few tries). However, not all programs will respond to this method of termination.

5.4.2 What Is Really Going On Here?

It is important to understand that job control is done by the shell. There is no program on the system called `fg`; rather, `fg`, `bg`, `&`, `jobs`, and `kill` are all shell-builtins (actually, sometimes `kill` is an independent program, but the `bash` shell used by Linux has it built in). This is a logical way to do it: since each user wants their own job control space, and each user already has their own shell, it is easiest to just have the shell keep track of the user's jobs. Therefore, each user's job numbers are meaningful only to that user: my job number [1] and your job number [1] are probably two totally different processes. In fact, if you are logged in more than once, each of your shells will have unique job control data, so you as a user might have two different jobs with the same number running in two different shells.

The way to tell for sure is to use the Process ID numbers (`PID`'s). These are system-wide — each process has its own unique `PID` number. Two different users can refer to a process by its `PID` and know that they are talking about the same process (assuming that they are logged into the same machine!)

If you start to program in C on your Linux system, you will soon learn that the shell's job control is just an interactive version of the function calls `fork` and `execl`. This is too complex to go into here, but may be helpful to remember later on when you are programming and want to run multiple processes from a single program.

5.5 Virtual Consoles: Being in Many Places at Once

Linux supports **virtual consoles**. These are a way of making your single machine seem like multiple terminals, all connected to one Linux kernel. Thankfully, using virtual consoles is one of the simplest things about Linux: there are "hot keys" for switching among the consoles quickly. To try it, log in to your Linux system, hold down the left $\boxed{\text{Alt}}$ key, and press $\boxed{\text{F2}}$ (that is, the function key number

2).[6]

You should find yourself at another login prompt. Don't panic: you are now on virtual console (VC) number 2! Log in here and do some things — a few `ls`'s or whatever — to confirm that this is a real login shell. Now you can return to VC number 1, by holding down the left $\boxed{\text{Alt}}$ and pressing $\boxed{\text{F1}}$. Or you can move on to a *third* VC, in the obvious way ($\boxed{\text{Alt}}$-$\boxed{\text{F3}}$).

Linux systems generally come with four VC's enabled by default. You can increase this all the way to eight; this should be covered in `The Linux System Administrator's Guide`. It involves editing a file in `/etc` or two. However, four should be enough for most people.

Once you get used to them, VC's will probably become an indispensable tool for getting many things done at once. For example, I typically run Emacs on VC 1 (and do most of my work there), while having a communications program up on VC 3 (so I can be downloading or uploading files by modem while I work, or running jobs on remote machines), and keep a shell up on VC 2 just in case I want to run something else without tying up VC 1.

[6] Make sure you are doing this from text consoles: if you are running X windows or some other graphical application, it probably won't work, although rumor has it that X Windows will soon allow virtual console switching under Linux.

Chapter 6

Powerful Little Programs

6.1 The Power of Unix

The power of Unix is hidden in small commands that don't seem too useful when used alone, but when combined with other commands (either directly or indirectly) produce a system that's much more powerful and flexible than most other operating systems. The commands I'm going to talk about in this chapter include `sort`, `grep`, `more`, `cat`, `wc`, `spell`, `diff`, `head`, and `tail`. Unfortunately, it isn't totally intuitive what these names mean right now. Let's cover what each of these utilities do seperately and then I'll give some examples of how to use them together.

6.2 Operating on Files

In addition to the commands like `cd`, `mv`, and `rm` you learned in Chapter 4, there are other commands that just operate on files but not the data in them. These include `touch`, `chmod`, `du`, and `df`. All of these files don't care what is *in* the file—the merely change some of the things Unix remembers about the file.

Some of the things these commands manipulate:

- The time stamp. Each file has three dates associated with it.[1] The three dates are the creation time (when the file was created), the last modification time (when the file was last changed), and the last access time (when the file was last read).
- The owner. Every file in Unix is owned by one user or the other.
- The group. Every file also has a group of users it is associated with. The most common group for user files is called `users`, which is usually shared by all the user account on the system.

[1] Older filesystems in LINUX only stored one date, since they were derived from Minix. If you have one of these filesystems, some of the information will merely be unavailable—operation will be mostly unchanged.

- The permissions. Every file has permissions associated with it which tell Unix who can access what file, or change it, or, in the case of programs, execute it. Each of these permissions can be toggled seperately for the owner, the group, and all other users.

`touch` *file1 file2 . . . fileN*

`touch` will update the time stamps of the files listed on the command line to the current time. If a file doesn't exist, `touch` will create it. It is also possible to specify the time that `touch` will set files to—consult the the manpage for `touch`.

`chmod` [-Rcfv] *mode file1 file2 . . . fileN*

The command used to change the permissions on a file is called `chmod`. Before I go into how to use the command, let's discuss what permissions are in Unix. Each file has a group of permissions associated with it. These permissions tell Unix whether or not the file can be read from, written to, or executed as a program.

du

df

6.3 What's in the File?

There are two major commands used in Unix for listing files, `cat` and `more`. I've talked about both of them in Chapter 5.

`cat` [-nA] [*file1 file2 . . . fileN*]

`cat` is not a user friendly command—it doesn't wait for you to read the file, and is mostly used in conjuction with pipes. However, `cat` does have some useful command-line options. For instance, `n` will number all the lines in the file, and `A` will show control characters as normal characters instead of (possibly) doing strange things to your screen. (Remember, to see some of the stranger and perhaps "less useful" options, use the `man` command: `man cat`.) `cat` will accept input from stdin if no files are specified on the command-line.

`more` [-l] [+*linenumber*] [*file1 file2 . . . fileN*]

`more` is much more useful, and is the command that you'll want to use when browsing ASCII text files. The only interesting option is `l`, which will tell `more` that you aren't interested in treating the character Ctrl-L as a "new page" character. `more` will start on a specified linenumber.

Since `more` is an interactive command, I've summarized the major interactive commands below:

Spacebar Moves to the next screen of text.

$\boxed{\texttt{d}}$ This will scroll the screen by 11 lines, or about half a normal, 25-line, screen.

$\boxed{\texttt{/}}$ Searches for a regular expression. While a regular expression can be quite complicated, you can just type in a text string to search for. For example, `/toad`$\boxed{\texttt{return}}$ would search for the next occurence of "toad" in your current file. A slash followed by a return will search for the next occurence of what you last searched for.

$\boxed{\texttt{n}}$ This will also search for the next occurence of your regular expression.

$\boxed{\texttt{:}}\boxed{\texttt{n}}$ If you specified more than one file on the command line, this will move to the next file.

$\boxed{\texttt{:}}\boxed{\texttt{p}}$ This will move the the previous file.

$\boxed{\texttt{q}}$ Exits from `more`.

`head` [-*lines*] [*file1 file2 . . . fileN*]

`head` will display the first ten lines in the listed files, or the first ten lines of stdin if no files are specified on the command line. Any numeric option will be taken as the number of lines to print, so `head -15 frog` will print the first fifteen lines of the file `frog`.

`tail` [-*lines*] [*file1 file2 . . . fileN*]

Like `head`, `tail` will display only a fraction of the file. Naturally, `tail` will display the end of the file, or the last ten lines that come through stdin. `tail` also accepts a option specifying the number of lines.

6.4 Commands to Operate on File Attributes

A file attribute is, for example, who "owns" the file or whether or not a file is an executable.

6.5 Commands to Operate of File Contents

This section discusses the commands that will alter a file, perform a certain operation on the file, or display statistics on the file.

`grep` [-nvwx] [-*number*] *expression* [*file1 file2 . . . fileN*]

One of the most useful commands in Unix is `grep`, the generalized **r**egular **e**xpression **p**arser. This is a fancy name for a utility which can only search a text file. The easiest way to use `grep` is like this:

```
/home/larry# cat animals
Animals are very interesting creatures. One of my favorite animals is
the tiger, a fearsome beast with large teeth.
I also like the lion---it's really neat!
/home/larry# grep iger animals
the tiger, a fearsome beast with large teeth.
/home/larry#
```

One disadvantage of this is, although it shows you all the lines containing your word, it doesn't tell you where to look in the file—no line number. Depending on what you're doing, this might be fine. For instance, if you're looking for errors from a programs output, you might try `a.out | grep error`, where `a.out` is your program's name.

If you're interested in where the match(es) are, use the `n` switch to `grep` to tell it to print line numbers. Use the `v` switch if you want to see all the lines that *don't* match the specified expression.

Another feature of `grep` is that it matches only parts of a word, like my example above where `iger` matched `tiger`. To tell `grep` to only match whole words, use the `w`, and the `x` switch will tell grep to only match whole lines.

Remember, if you don't specify any files, `grep` will examine stdin.

`wc` [-clw] [*file1 file2* . . . *fileN*]

`wc` stands for **w**ord **c**ount. It simply counts the number of words, lines, and characters in the file(s). If there aren't any files specified on the command line, it operates on stdin.

The three parameters, `clw`, stand for **c**haracter, **l**ine, and **w**ord respectively, and tell `wc` which of the three to count. Thus, `wc -cw` will count the number of characters and words, but not the number of lines. `wc` defaults to counting everything—words, lines, and characters.

One nice use of `wc` is to find how many files are in the present directory: `ls | wc -w`. If you wanted to see how many files that ended with `.c` there were, try `ls *.c | wc -w`.

`spell` [*file1 file2* . . . *fileN*]

`spell` is a very simple Unix spelling program, usually for American English.[2] `spell` is a filter, like most of the other programs we've talked about, which sucks in an ASCII text file and outputs all the words it considers misspellings. `spell` operates on the files listed in the command line, or, if there weren't any there, stdin.

A more sophisticated spelling program, `ispell` is probably also available on your machine. `ispell` will offer possible correct spellings and a fancy menu interface if a filename is specified on the command line or will run as a filter-like program if no files are specified.

While operation of `ispell` should be fairly obvious, consult the man page if you need more help.

[2]While there are versions of this for several other European languages, the copy on your LINUX machine is most likely for American English and only American English. Sorry.

`cmp` *file1* [*file2*]

`cmp` **comp**ares two files. The first must be listed on the command line, while the second is either listed as the second parameter or is read in from standard input. `cmp` is very simple, and merely tells you where the two files first differ.

`diff` *file1 file2*

One of the most complicated standard Unix commands is called `diff`. The GNU version of `diff` has over twenty command line options! It is a much more powerful version of `cmp` and shows you what the differences are instead of merely telling you where the first one is.

Since talking about even a good portion of `diff` is beyond the scope of this book, I'll just talk about the basic operation of `diff`. In short, `diff` takes two parameters and displays the differences between them on a line-by-line basis. For instance:

```
/home/larry# cat frog
Animals are very interesting creatures. One of my favorite animals is
the tiger, a fearsome beast with large teeth.
I also like the lion---it's really neat!
/home/larry# cp frog toad
/home/larry# diff frog toad
/home/larry# cat dog
Animals are very nteresting creatures. One of my favorite animals is

the tiger, a fearsome beast with large teeth.
I also   like the lion---it's really neat!
/home/larry# diff frog dog
1c1,2
< Animals are very interesting creatures. One of my favorite animals is
---
> Animals are very nteresting creatures. One of my favorite animals is
>
3c4
< I also like the lion---it's really neat!
---
> I also   like the lion---it's really neat!
/home/larry#
```

As you can see, `diff` outputs nothing when the two files are identical. Then, when I compared two different files, it had a section header, `1c1,2` saying it was comparing line 1 of the left file, `frog`, to lines 1–2 of `dog` and what differences it noticed. Then it compared line 3 of `frog` to line 4 of `dog`. While it may seem strange at first to compare different line numbers, it is much more efficent then listing out every single line if there is an extra return early in one file.

Chapter 7

Editing files with Emacs

FUNNY SOMETHING OR OTHER

7.1 What's `emacs`?

In order to get anything done on a computer, you need a way to put text into files, and a way to change text that's already in files. An **editor** is a program for doing this. `Emacs` is one of the most popular editors around—partly because it's very easy for a complete beginner to get actual work done with it. To learn it, you need to find a file of plain text (letters, numbers, and the like), copy it to your home directory (we don't want to modify the actual file, if it contains important information), and invoke Emacs on the file:

```
prompt> emacs some_file.txt
```

"Invoking" Emacs can have different effects depending on where where you do it. From a plain console displaying only text characters, Emacs will just take over the whole console. If you invoke it from X Windows, Emacs will actually bring up its own window. I will assume that you are doing it from a text console, but everything carries over logically into the X Windows version—just substitute the word "window" in the places I've written "screen".

You should see the contents of the file filling most of the screen (or as much of the file as fits on the screen, if it's a lot of text). Then, one line from the bottom of the screen appears in reverse video. This line is called the **mode-line** in Emacs. It should say something like:

```
-----Emacs: some_file.txt          (Fundamental)--Top------------------------
```

The word "Top" might be "All" instead, and there might be other minor differences. The one blank line (in regular video) immediately below the mode-line is called the **minibuffer**, or sometimes the **echo area**. Emacs uses the minibuffer to flash messages at you, and occasionally uses it to read input from you, when necessary. Ignore it for now; we won't be making much use of the minibuffer for a while.

Before you actually change any of the text in the file, you need to learn how to move around. The cursor should be at the beginning of the file, in the upper-left corner of the screen. To move forward, type `C-f` (that is, hold down the Control key while you press "f", for "forward"). It will move you forward a character at a time, and if you hold both keys down, your system's automatic key-repeat should take effect in a half-second or so. Notice how when you get to the end of the line, it moves smoothly on to the next line. `C-b` (for "backward") does the opposite. And, while we're at it, `C-n` and `C-p` take you to the next and previous lines, respectively.[1]

Hold down `C-b` until you've been taken all the way back to the upper-left corner, and then keep it held a little longer. You should hear an annoying bell sound, and see the message "`Beginning of buffer`" appear in the minibuffer. At this point you might wonder, "But what is a buffer?" Well, here's how it works:

When Emacs works on a file, it doesn't actually work on the file itself. Instead, it copies the contents of the file into a special Emacs work area called a **buffer**, where you can modify it to your heart's content. When you are done working, you tell Emacs to save the buffer—in other words, to write the buffer's contents into the corresponding file. Until you do this, the file remains unchanged, and the buffer's contents exist only inside of Emacs.

With that in mind, prepare to insert your first character into the buffer. Until now, everything we have done has been "non-destructive", so this is a big moment. You can choose any character you like, but if you want to do this in style, I suggest using a nice, solid, capital "X". As you type it, take a look at the beginning of the mode-line at the bottom of the screen. When you change the buffer so that its contents are no longer the same as those of the file on disk, Emacs displays two asterisks at the beginning of the mode-line, to let you know that the buffer has been modified:

```
--**-Emacs: some_file.txt          (Fundamental)--Top-----------------------
```

These two asterisks are displayed as soon as you modify the buffer, and remain visible until you save the buffer. You can save the buffer multiple times during an editing session—the command to do so is just `C-x C-s` (hold down Control and hit "x" and "s" while it's down...okay, so you probably already figured that out!). It's deliberately easy to type, because saving your buffers is something best done early and often.

I'm going to list a few more commands now, along with the ones you've learned already, and you can practice them however you like. I'd suggest becoming familiar with them before going any further:

[1] In case you hadn't noticed yet, many of Emacs' movement commands consist of combining Control with a single mnemonic letter.

`C-f`	Move forward one character.
`C-b`	Move backward one character.
`C-n`	Go to next line.
`C-p`	Go to previous line.
`C-a`	Go to beginning of line.
`C-e`	Go to end of line.
`C-v`	Go to next page/screenful of text.
`C-l`	Redraw the screen, with current line in center.
`C-d`	Delete this character (practice this one).
`C-k`	Delete text from here to end of line.
`C-x C-s`	Save the buffer in its corresponding file.
Backspace	Delete preceding character (the one you just typed).

7.2 Editing Many Files at Once

Emacs can work on more than one file at a time. In fact, the only limit on how many buffers your Emacs can contain is the actual amount of memory available on the machine.[2] The command to bring a new file into an Emacs buffer is `C-x C-f`. When you type it, you will be prompted for a filename in the minibuffer:

```
Find file: ~/
```

The syntax here is the same one used to specify files from the shell prompt; slashes represent subdirectories, `~` means your home directory, etc. You also get **filename completion**, meaning that if you've typed enough of a filename at the prompt to identify the file uniquely, you can just hit Tab to complete it (or to show possible completions, if there are more than one). Space also has a role in filename completion in the minibuffer, similar to Tab, but I'll let you experiment to find out how the two differ. Once you have the full filename in the minibuffer, hit Return, and Emacs will bring up a buffer displaying that file. In Emacs, this process is known as **finding** a file. Go ahead and find some other unimportant text file now, and bring it into Emacs (do this from our original buffer `some_file.txt`). Now you have a new buffer; I'll pretend it's called `another_file.txt`, since I can't see your mode-line.

Your original buffer seems to have disappeared—you're probably wondering where it went. It's still inside Emacs, and you can switch back to it with `C-x b`. When you type this, you will see that the minibuffer prompts you for a buffer to switch to, and it names a default. The default is the buffer you'd get if you just hit Return at the prompt, without typing a buffer name. The default buffer to switch to is always the one most recently left, so that when you are doing a lot of work between two buffers, `C-x b` always defaults to the "other" buffer (which saves you from having to type the buffer name). Even if the default buffer is the one you want, however, you should try typing in its name anyway.

[2] This leads to one of the more Zen-like quotes in Emacs lore, in which an unimpressed user asks an Emacs fanatic "Who cares? Why would I ever want to have two hundred buffers open simultaneously?" To which the fanatic replies: "But isn't it nice to know that you can?"

Notice that you get the same sort of completion you got when finding a file: hitting Tab completes as much of a buffer name as it can, and so on. Whenever you are being prompted for something in the minibuffer, it's a good idea to see if Emacs is doing completion. Taking advantage of completion whenever it's offered will save you a lot of typing. Emacs usually does completion when you are choosing one item out of some predefined list.

Everything you learned about moving around and editing text in the first buffer applies to the new one. Go ahead and change some text in the new buffer, but don't save it (i.e. don't type `C-x C-s`). Let's assume that you want to discard your changes without saving them in the file. The command for that is `C-x k`, which "kills" the buffer. Type it now. First you will be asked which buffer to kill, but the default is the current buffer, and that's almost always the one you want to kill, so just hit Return. Then you will be asked if you *really* want to kill the buffer—Emacs always checks before killing a buffer that has unsaved changes in it.[3] Just type "yes" and hit Return, if you want to kill it.

Go ahead and practice loading in files, modifying them, saving them, and killing their buffers. Make sure you don't modify any important system files in a way that will cause trouble[4], of course, but do try to have at least five buffers open at once, so you can get the hang of switching between them.

7.3 Ending an Editing Session

When you are done with your work in Emacs, make sure that all buffers are saved that should be saved, and exit Emacs with `C-x C-c`. Sometimes `C-x C-c` will ask you a question or two in the minibuffer before it lets you leave—don't be alarmed, just answer them in the obvious ways. If you think that you might be returning to Emacs later, don't use `C-x C-c` at all; use `C-z`, which will suspend Emacs. You can return to it with the shell command "`fg`" later.[5] This is more efficient than stopping and starting Emacs multiple times, especially if you have edit the same files again later.

7.4 The Meta Key

You've already learned about one "modifier key" in Emacs, the Control key. There is a second one, called the **Meta** key, which is used almost as frequently. However, not all keyboards have their Meta key in the same place, and some don't have one at all. The first thing you need to do is find where your Meta key is located. Chances are, your keyboard's Alt keys are also Meta keys, if you are using an IBM PC or other another keyboard that has an Alt key.

[3]You can tell if a buffer has unsaved changes by looking at the beginning of the mode-line. If you see two asterisks in a row right near the left edge of the mode-line ("`**`"), then the buffer has been modified since it was last saved.

[4]If you are not the "root" user on the machine, you shouldn't be able to hurt the system anyway, but be careful just the same.

[5]If you are running Emacs under a windowing system, like X Windows, then just iconify the window. This is analogous to suspending it, though not quite the same.

The way to test this is to hold down a key that you think might be a Meta key and type "x". If you see a little prompt appear in the minibuffer (like this: `M-x`) then you've found it. To get rid of the prompt and go back to your Emacs buffer, type `C-g`.

If you didn't get a prompt, then there is still one solution. You can use the $\boxed{\text{Escape}}$ key as a Meta key. But instead of holding it down while you type the next letter, you have to tap it and release it quickly, and *then* type the letter. This method will work whether or not you have a real Meta key, so it's the safest way to go. Try tapping $\boxed{\text{Escape}}$ and then typing "x" now. You should get that tiny prompt again. Just use `C-g` to make it go away. `C-g` is the general way in Emacs to quit out of something you don't mean to be in. It usually beeps annoyingly at you to let you know that you have interrupted something, but that's fine, since that's what you intended to do if you typed `C-g`!

The notation `M-x` is analogous to `C-x` (substitute any character for "x"). If you have found a real Meta key, use that, otherwise just use the $\boxed{\text{Escape}}$ key. I will simply write `M-x`, since I don't know which alternative you are using.

7.5 Cutting, Pasting, Killing and Yanking

Emacs, like any good editor, allows you to cut and paste blocks of text. In order to do this, you need a way to define the start and end of the block. In Emacs, you do this by setting two locations in the buffer, known as **mark** and **point**. To set the mark, go to the place you want your block to begin and type `C-SPC` ("SPC" means $\boxed{\text{Space}}$, of course). You should see the message "Mark set" appear in the minibuffer.[6] The mark has now been set at that place. There will be no special highlighting indicating that fact, but you know where you put it, and that's all that matters.

What about **point**? Well, it turns out that you've been setting point every time you move the cursor, because "point" just refers to your current location in the buffer. In formal terms, point is the spot where text would be inserted if you were to type something. By setting the mark, and then moving to the end of the block of text, you have actually defined a block of text. This block is known as the **region**. The "**region**" always means the area between mark and point.

Merely defining the region does not make it available for pasting. You have to tell Emacs to copy it in order to be able to paste it. To copy the region, make sure that mark and point are set correctly, and type `M-w`. It has now been recorded by Emacs. In order to paste it somewhere else, just go there and type `C-y`. This is known as **yanking** the text into the buffer.

If you want to actually move the text of the region to somewhere else, type `C-w` instead of `M-w`. This will **kill** the region—all the text inside it will disappear. In fact, it has been saved in the same way as if you had used `M-w`. You can yank it back out with `C-y`, as always. The place Emacs saves all this text is known as the **kill-ring**. Some editors call it the "clipboard" or the "paste buffer".

There's another way to do cutting and pasting: whenever you use `C-k` to kill to the end of a line, the killed text is saved in the kill-ring. If you kill more than one line in a row, they are all saved in the kill-ring together, so that the next yank will paste in all the lines at once. Because of this

[6] Sometimes, on a few machines, `C-SPC` doesn't work. For these machines, you must use `C-@`.

feature, it is often faster to use repeated `C-k`'s to kill some text than it is to explicitly set mark and point and use `C-w`. However, either way will work. It's really a matter of personal preference how you do it.

7.6 Searching and Replacing

There are several ways to search for text in Emacs. Many of them are rather complex, and not worth going into here. The easiest and most entertaining way is to use **isearch**. "Isearch" stands for "incremental search". Suppose you want to search for the string "gadfly" in the following buffer:

```
I was growing afraid that we would run out of gasoline, when my passenger exclaimed
``Gadzooks!  There's a gadfly in here!''.
```

You would move to the beginning of the buffer, or at least to some point that you know is before the first occurence of the goal word, "gadfly", and type `C-s`. That puts you in isearch mode. Now start typing the word you are searching for, "gadfly". But as soon as you type the "g", you see that Emacs has jumped you to the first occurence of "g" in the buffer. If the above quote is the entire contents of the buffer, then that would be the first "g" of the word "growing". Now type the "a" of "gadfly", and Emacs leaps over to "gasoline", which contains the first occurence of a "ga". The "d" gets you to gadzooks, and finally, "f" gets you to "gadfly", without your having had to type the entire word.

What you are doing in an isearch is defining a string to search for. Each time you add a character to the end of the string, the number of matches is reduced, until eventually you have entered enough to define the string uniquely. Once you have found the match you are looking for, you can exit the search with $\boxed{\text{Return}}$ or any of the normal movement commands. If you think the string you're looking for is behind you in the buffer, then you should use `C-r`, which does an isearch backwards.

If you encounter a match, but it's not the one you were looking for, then hit `C-s` again while still in the search. This will move you forward to the next complete match, each time you hit it. If there is no next match, it will say that the search failed, but if you press `C-s` again at that point, the search will wrap around from the beginning of the buffer. The reverse holds true for `C-r` — it wraps around the end of the buffer.

Try bringing up a buffer of plain English text and doing and isearch for the string "`the`". First you'd type in as much as you wanted, then use repeated `C-s`'s to go to all instances of it. Notice that it will match words like "`them`" as well, since that also contains the substring "`the`". To search only for "`the`", you'd have to do add a space to the end of your search string. You can add new characters to the string at any point in the search, even after you've hit `C-s` repeatedly to find the next matches. You can also use $\boxed{\text{Backspace}}$ or $\boxed{\text{Delete}}$ to remove characters from the search string at any point in the search, and hitting $\boxed{\text{Return}}$ exits the search, leaving you at the last match.

Emacs also allows you to replace all instances of a string with some new string—this is known as **query-replace**. To invoke it, type `query-replace` and hit $\boxed{\text{Return}}$. Completion is done on the command name, so once you have typed "query-re", you can just hit $\boxed{\text{Tab}}$ to finish it. Say you wish to replace all instances of "gadfly" with "housefly". At the "`Query replace: `" prompt, type "gadfly", and hit Return. Then you will be prompted again, and you should enter "housefly". Emacs will then step through the buffer, stopping at every instance of the word "gadfly", and asking if you want to replace it. Just hit "`y`" or "`n`" at each instance, for "Yes" or "No", until it finishes. If this doesn't make sense as you read it, then try it out.

7.7 What's Really Going On Here?

Actually, all these **keybindings** you have been learning are shortcuts to Emacs functions. For example, `C-p` is a short way of telling Emacs to execute the internal function `previous-line`. However, all these internal functions can be called by name, using `M-x`. If you forgot that `previous-line` is bound to `C-p`, you could just type `M-x previous-line` Return, and it would move you up one line. Try this now, to understand how `M-x previous-line` and `C-p` are really the same thing.

The designer of Emacs started from the ground up, first defining a whole lot of internal functions, and then giving keybindings to the most commonly-used ones. Sometimes it's easier just to call a function explicitly with `M-x` than to remember what key it's bound to. The function `query-replace`, for example, is bound to `M-%` in some versions of Emacs. But who can remember such an odd keybinding? Unless you use query-replace extremely often, it's easier just to call it with `M-x`.

Most of the keys you type are letters, meant to be inserted into the text of the buffer. So each of those keys is **bound** to the function `self-insert-command`, which does nothing but insert that letter into the buffer. Combinations that use the Control key with a letter are generally bound to functions that do other things, like moving you around. For example, `C-v` is bound to a function called `scroll-up`, which scrolls the buffer up by one screenful (meaning that your position in the buffer moves *down*, of course).

If you ever actually wanted to insert a Control character into the buffer, then, how would you do it? After all, the Control characters are `ASCII` characters, although rarely used, and you might want them in a file. There is a way to prevent Control characters from being interpreted as commands by Emacs. The key `C-q`[7] is bound to a special function named `quoted-insert`. All `quoted-insert` does is read the next key and insert it literally into the buffer, without trying to interpret it as a command. This is how you can put Control characters into your files using Emacs. Naturally, the way to insert a C-q is to press `C-q` twice!

Emacs also has many functions that are not bound to any key. For example, if you're typing a long message, you don't want to have to hit return at the end of every line. You can have Emacs do it for you (you can have Emacs do anything for you)—the command to do so is called `auto-fill-mode`[8], but it's not bound to any keys by default. In order to invoke this command, you would type "`M-x auto-fill-mode`". "`M-x`" is the key used to call functions by name. You could even use it to call functions like `next-line` and `previous-line`, but that would be very inefficient, since they are already bound to `C-n` and `C-p`!

[7] We call `C-q` a "key", even though it is produced by holding down Control and pressing "q", because it is a single `ASCII` character.

[8] You might expect this to be called '`wrap-mode`', or '`auto-wrap-mode`' ... but you'd be wrong. Sorry!

By the way, if you look at your mode-line after invoking `auto-fill-mode`, you will notice that the word "Fill" has been added to the right side. As long as it's there, Emacs will fill (wrap) text automatically. You can turn it off by typing "`M-x auto-fill-mode`" again—it's a **toggle** command.

The inconvenience of typing long function names in the minibuffer is lessened because Emacs does completion on function names the same way it does on file names. Therefore, you should rarely find yourself typing in the whole function name, letter by letter. If you're not sure whether or not you can use completion, just hit Tab. It can't hurt: the worst thing that will happen is that you'll just get a tab character, and if you're lucky, it'll turn out that you can use completion.

7.8 Asking Emacs for Help

Emacs has extensive help facilities—so extensive, in fact, that we can only touch on them here. The most basic help features are accessed by typing `C-h` and then a single letter. For example, `C-h k` gets help on a key (it prompts you to type a key, then tells you what that key does). `C-h t` brings up a short Emacs tutorial. Most importantly, `C-h C-h C-h` gets you help on help, to tell you what's available once you have typed `C-h` the first time. If you know the name of an Emacs function (`save-buffer`, for example), but can't remember what key sequence invokes it, then use `C-h w`, for "`where-is`", and type in the name of the function. Or, if you want to know what a function does in detail, use `C-h f`, which prompts for a function name.

Remember, since Emacs does completion on function names, you don't really have to be sure what a function is called to ask for help on it. If you think you can guess the word it might start with, type that and hit Tab to see if it completes to anything. If not, back up and try something else. The same goes for file names: even if you can't remember quite what you named some file that you haven't accessed for three months, you can guess and use completion to find out if you're right. Get used to using completion as means of asking questions, not just as a way of saving keystrokes.

There are other characters you can type after `C-h`, and each one gets you help in a different way. The ones you will use most often are `C-h k`, `C-h w`, and `C-h f`. Once you are more familiar with Emacs, another one to try is `C-h a`, which prompts you for a string and then tells you about all the functions who have that string as part of their name (the "a" means for "apropos", or "about").

Another source of information is the **Info** documentation reader. Info is too complex a subject to go into here, but if you are interested in exploring it on your own, type `C-h i` and read the paragraph at the top of the screen. It will tell you how get more help.

7.9 Specializing Buffers: Modes

Emacs buffers have **modes** associated with them[9]. The reason for this is that your needs when writing a mail message are very different from your needs when, say, writing a program. Rather than try to come up with an editor that would meet every single need all the time (which would be

[9]To make matters worse, there are "Major Modes" and "Minor Modes", but you don't need to know about that.

impossible), the designer of Emacs[10] chose to have Emacs behave differently depending on what you are doing in each individual buffer. Thus, buffers have modes, each one designed for some specific activity. The main features that distinguish one mode from another are the keybindings, but there can be other differences as well.

The most basic mode is `fundamental` mode, which doesn't really have any special commands at all. In fact, here's what Emacs has to say about Fundamental Mode:

```
Fundamental Mode:

Major mode not specialized for anything in particular.
Other major modes are defined by comparison with this one.
```

I got that information like this: I typed `C-x b`, which is `switch-to-buffer`, and entered "foo" when it prompted me for a buffer name to switch to. Since there was previously no buffer named "`foo`", Emacs created one and switched me to it. It was in `fundamental-mode` by default, but it it hadn't been, I could have typed "`M-x fundamental-mode`" to make it so. All mode names have a command called `<modename>-mode` which puts the current buffer into that mode. Then, to find out more information about that major mode, I typed `C-h m`, which gets you help on the current major mode of the buffer you're in.

There's a slightly more useful mode called `text-mode`, which has the special commands `M-S`, for `center-paragraph`, and `M-s`, which invokes `center-line`. `M-S`, by the way, means exactly what you think it does: hold down both the Meta and the Shift key, and press "S".

Don't just take my word for this—go make a new buffer, put it into `text-mode`, and type `C-h m`. You may not understand everything Emacs tells you when you do that, but you should be able to get some useful information out of it.

Here is an introduction to some of the more commonly used modes. If you use them, make sure that you type `C-h m` sometime in each one, to find out more about each mode.

7.10 Programming Modes

7.10.1 C Mode

If you use Emacs for programming in the C language, you can get it to do all the indentation for you automatically. Files whose names end in "`.c` " or "`.h`" are automatically brought up in `c-mode`. This means that certain special editing commands, useful for writing C-programs, are available. In C-mode, Tab is bound to `c-indent-command`. This means that hitting the Tab key does not actually insert a tab character. Instead, if you hit Tab anywhere on a line, Emacs automatically indents that line correctly for its location in the program. This implies that Emacs knows something about C syntax, which it does (although nothing about semantics—it cannot insure that your program has no errors!)

[10] Well, there's no reason not to use his name. He is Richard Stallman, also sometimes referred to as "`rms`", because that's his login name.

In order to do this, it assumes that the previous line(s) are indented correctly. That means that if the preceding line is missing a parenthesis, semicolon, curly brace, or whatever, Emacs will indent the current line in a funny way. When you see it do that, you will know to look for a punctuation mistake on the line above.

You can use this feature to check that you have punctuated your programs correctly—instead of reading through the entire program looking for problems, just start indenting lines from the top down with $\boxed{\text{Tab}}$, and when something indents oddly, check the lines just before it. In other words, let Emacs do the work for you!

7.10.2 Scheme Mode

This is a major mode that won't do you any good unless you have a compiler or an interpreter for the Scheme programming language on your system. Having one is not as normal as having, say, a C compiler, but it's becoming more and more common, so I'll cover it too. Much of what is true for Scheme mode is true for Lisp mode as well, if you prefer to write in Lisp.

Well, to make matters painful, Emacs comes with two different Scheme modes, because people couldn't decide how they wanted it to work. The one I'm describing is called "`cmuscheme`", and later on, in the section on customizing Emacs, I'll talk about how there can be two different Scheme modes and what to do about it. For now, don't worry about it if things in your Emacs don't quite match up to what I say here. A customizable editor means an unpredictable editor, and there's no way around that!

You can run an interactive Scheme process in Emacs, with the command `M-x run-scheme`. This creates a buffer named "`*scheme*`", which has the usual Scheme prompt in it. You can type in Scheme expressions at the prompt, hit $\boxed{\text{Return}}$, and Scheme will evaluate them and display the answer. Thus, in order to interact with the Scheme process, you could just type all your function definitions and applications in at the prompt. Chances are you have previously-written Scheme source code in a file somewhere, and it would be easier to do your work in that file and send the definitions over to the Scheme process buffer as necessary.

If that source file ends in "`.ss`" or "`.scm`", it will automatically be brought up in **Scheme mode** when you find it with `C-x C-f`. If for some reason, it doesn't come up in Scheme mode, you can do it by hand with `M-x scheme-mode`. This `scheme-mode` is not the same thing as the buffer running the Scheme process; rather, the source code buffer's being in `scheme-mode` means that it has special commands for communicating with the process buffer.

If you put yourself inside a function definition in the Scheme source code buffer and type `C-c C-e`, then that definition will be "sent" to the process buffer — exactly as if you had typed it in yourself. `C-c M-e` sends the definition and then brings you to the process buffer to do some interactive work. `C-c C-l` loads a file of Scheme code (this one works from either the process buffer or the source code buffer). And like other programming language modes, hitting $\boxed{\text{Tab}}$ anywhere on a line of code correctly indents that line.

If you're at the prompt in the process buffer, you can use `M-p` and `M-n` to move through your previous commands (also known as the **input history**). So if you are debugging the function `'rotate'`, and have already applied it to arguments in the process buffer, like so:

```
    > (rotate '(a b c d e))
```

then you can get that command back by typing `M-p` at the prompt later on. There should be no need to retype long expressions at the Scheme prompt — get in the habit of using the input history and you'll save a lot of time.

Emacs knows about quite a few programming languages: C, C++, Lisp, and Scheme are just some. Generally, it knows how to indent them in intuitive ways.

7.10.3 Mail Mode

You can also edit and send mail in Emacs. To enter a mail buffer, type `C-x m`. You need to fill in the `To:` and `Subject:` fields, and then use C-n to get down below the separator line into the body of the message (which is empty when you first start out). Don't change or delete the separator line, or else Emacs will not be able to send your mail—it uses that line to distinguish the mail's headers, which tell it where to send the mail, from the actual contents of the message.

You can type whatever you want below the separator line. When you are ready to send the message, just type `C-c C-c`, and Emacs will send it and then make the mail buffer go away.

7.11 Being Even More Efficient

Experienced Emacs users are fanatical about efficiency. In fact, they will often end up wasting a lot of time searching for ways to be more efficient! While I don't want that to happen to you, there are some easy things you can do to become a better Emacs user. Sometimes experienced users make novices feel silly for not knowing all these tricks—for some reason, people become religious about using Emacs "correctly". I'd condemn that sort of elitism more if I weren't about to be guilty of it myself. Here we go:

When you're moving around, use the fastest means available. You know that `C-f` is `forward-char`—can you guess that `M-f` is `forward-word`? `C-b` is `backward-char`. Guess what `M-b` does? That's not all, though: you can move forward a sentence at a time with `M-e`, as long as you write your sentences so that there are always two spaces following the final period (otherwise Emacs can't tell where one sentence ends and the next one begins). `M-a` is `backward-sentence`.

If you find yourself using repeated `C-f`'s to get to the end of the line, be ashamed, and make sure that you use `C-e` instead, and `C-a` to go to the beginning of the line. If you use many `C-n`'s to move down screenfuls of text, be very ashamed, and use `C-v` forever after. If you are using repeated `C-p`'s to move up screenfuls, be embarrassed to show your face, and use `M-v` instead.

If you are nearing the end of a line and you realize that there's a mispelling or a word left out somewhere earlier in the line, *don't* use $\boxed{\text{Backspace}}$ or $\boxed{\text{Delete}}$ to get back to that spot. That would require retyping whole portions of perfectly good text. Instead, use combinations of `M-b`, `C-b`, and `C-f` to move to the precise location of the error, fix it, and then use `C-e` to move to the end of the line again.

When you have to type in a filename, don't ever type in the whole name. Just type in enough of it to identify it uniquely, and let Emacs' completion finish the job by hitting Tab or Space. Why waste keystrokes when you can waste CPU cycles instead?

If you are typing some kind of plain text, and somehow your auto-filling (or auto-wrapping) has gotten screwed up, use `M-q`, which is **`fill-paragraph`** in common text modes. This will "adjust" the paragraph you're in as if it had been wrapped line by line, but without your having to go mess around with it by hand. `M-q` will work from inside the paragraph, or from its very beginning or end.

When something awful happens, and Emacs seems to be behaving weirdly because you hit some keys by accident and don't know what they did, the solution is not to go hitting more keys randomly. Just use `C-g`, which quits out of whatever you're in, and lets out a noise (if your terminal is capable of that) to inform you that something was interrupted. Whatever you do, don't panic—Emacs will not reward it.

Sometimes it's helpful to use `C-x u`, (**undo**), which will try to "undo" the last change(s) you made. Emacs will guess at how much to undo; usually it guesses very intelligently. Calling it repeatedly will undo more and more, until Emacs can no longer remember what changes were made.

7.12 Customizing Emacs

Emacs is *so* big, and *so* complex, that it actually has its own programming language! I'm not kidding: to really customize Emacs to suit your needs, you have to write programs in this language. It's called Emacs Lisp, and it's a dialect of Lisp, so if you have previous experience in Lisp, it will seem quite friendly. If not, don't worry: I'm not going to go into a great deal of depth, because it's definitely best learned by doing. To really learn about programming Emacs, you should consult the Info pages on Emacs Lisp, and read a lot of Emacs Lisp source code.

Most of Emacs' functionality is defined in files of Emacs Lisp[11] code. Most of these files are distributed with Emacs and collectively are known as the "Emacs Lisp library". This library's location depends on how Emacs was installed on your system — common locations are `/usr/lib/emacs/lisp`, `/usr/lib/emacs/19.19/lisp/`, etc. The "`19.19`" is the version number of Emacs, and might be different on your system.

You don't need to poke around your filesystem looking for the lisp library, because Emacs has the information stored internally, in a variable called **load-path**. To find out the value of this variable, it is necessary to **evaluate** it; that is, to have Emacs' lisp interpreter get its value. There is a special mode for evaluating Lisp expressions in Emacs, called **lisp-interaction-mode**. Usually, there is a buffer called "`*scratch*`" that is already in this mode. If you can't find one, create a new buffer of any name, and type `M-x lisp-interaction-mode` inside it.

Now you have a workspace for interacting with the Emacs Lisp interpreter. Type this:

```
load-path
```

and then press `C-j` at the end of it. In lisp-interaction-mode, `C-j` is bound to `eval-print-last-sexp`.

[11] Sometimes unofficially called "Elisp".

An "**sexp**" is an "**s-expression**", which means a balanced group of parentheses, including none. Well, that's simplifying it a little, but you'll get a feel for what they are as you work with Emacs Lisp. Anyway, evaluating `load-path` should get you something like this:

```
load-path C-j
("/usr/lib/emacs/site-lisp/vm-5.35" "/home/kfogel/elithp"
"/usr/lib/emacs/site-lisp" "/usr/lib/emacs/19.19/lisp")
```

It won't look the same on every system, of course, since it is dependant on how Emacs was installed. The above example comes from my 386 PC running Linux. As the above indicates, `load-path` is a list of strings. Each string names a directory that might contain Emacs Lisp files. When Emacs needs to load a file of Lisp code, it goes looking for it in each of these directories, in order. If a directory is named but does not actually exist on the filesystem, Emacs just ignores it.

When Emacs starts up, it automatically tries to load the file `.emacs` in your home directory. Therefore, if you want to make personal customizations to Emacs, you should put them in `.emacs`. The most common customizations are keybindings, so here's how to do them:

```
(global-set-key "\C-cl" 'goto-line)
```

`global-set-key` is a function of two arguments: the key to be bound, and the function to bind it to. The word "`global`" means that this keybinding will be in effect in all major modes (there is another function, `local-set-key`, that binds a key in a single buffer). Above, I have bound `C-c l` to the function `goto-line`. The key is described using a string. The special syntax "`\C-<char>`" means the Control key held down while the key `<char>` is pressed. Likewise, "`\M-<char>`" indicates the Meta key.

All very well, but how did I know that the function's name was "`goto-line`"? I may know that I want to bind `C-c l` to some function that prompts for a line number and then moves the cursor to that line, but how did I find out that function's name?

This is where Emacs' online help facilities come in. Once you have decided what kind of function you are looking for, you can use Emacs to track down its exact name. Here's one quick and dirty way to do it: since Emacs gives completion on function names, just type `C-h f` (which is `describe-function`, remember), and then hit Tab without typing anything. This asks Emacs to do completion on the empty string — in other words, the completion will match every single function! It may take a moment to build the completion list, since Emacs has so many internal functions, but it will display as much of it as fits on the screen when it's ready.

At that point, hit `C-g` to quit out of `describe-function`. There will be a buffer called "`*Completions*`", which contains the completion list you just generated. Switch to that buffer. Now you can use `C-s`, `isearch`, to search for likely functions. For example, it's a safe assumption that a function which prompts for a line number and then goes to that line will contain the string "`line`" in its name. Therefore, just start searching for the string "`line`", and you'll find what you're looking for eventually.

If you want another method, you can use `C-h a`, `command-apropos`, to show all functions whose names match the given string. The output of `command-apropos` is a little harder to sort through than just searching a completion list, in my opinion, but you may find that you feel differently. Try both methods and see what you think.

There is always the possibility that Emacs does not have any predefined function to do what you're looking for. In this situation, you have to write the function yourself. I'm not going to talk about how to do that — you should look at the Emacs Lisp library for examples of function definitions, and read the Info pages on Emacs Lisp. If you happen to know a local Emacs guru, ask her how to do it. Defining your own Emacs functions is not a big deal — to give you an idea, I have written 131 of them in the last year or so. It takes a little practice, but the learning curve is not steep at all.

Another thing people often do in their `.emacs` is set certain variables to preferred values. For example, put this in your `.emacs` and then start up a new Emacs:

```
(setq inhibit-startup-message t)
```

Emacs checks the value of the variable `inhibit-startup-message` to decide whether or not to display certain information about version and lack of warranty when it starts up. The Lisp expression above uses the command `setq` to set that variable to the value '`t`', which is a special Lisp value that means **true**. The opposite of '`t`' is '`nil`', which is the designated **false** value in Emacs Lisp. Here are two things that are in my `.emacs` that you might find useful:

```
(setq case-fold-search nil) ; gives case-insensitivity in searching
;; make C programs indent the way I like them to:
(setq c-indent-level 2)
```

The first expression causes searches (including `isearch`) to be case-insensitive; that is, the search will match upper- or lower-case versions of a character even though the search string contains only the lower-case version. The second expression sets the default indentation for C language statements to be a little smaller than it is normally — this is just a personal preference; I find that it makes C code more readable.

The comment character in Lisp is "`;`". Emacs ignores anything following one, unless it appears inside a literal string, like so:

```
;; these two lines are ignored by the Lisp interpreter, but the
;; s-expression following them will be evaluated in full:
(setq some-literal-string "An awkward pause; for no purpose.")
```

It's a good idea to comment your changes to Lisp files, because six months later you will have no memory of what you were thinking when you modified them. If the comment appears on a line by itself, precede it with two semicolons. This aids Emacs in indenting Lisp files correctly.

You can find out about internal Emacs variables the same ways you find out about functions. Use `C-h v`, `describe-variable` to make a completion list, or use `C-h C-a`, `apropos`. `Apropos` differs from `C-h a`, `command-apropos`, in that it shows functions and variables instead of just functions.

The default extension for Emacs Lisp files is "`.el`", as in "`c-mode.el`". However, to make Lisp code run faster, Emacs allows it to be **byte-compiled**, and these files of compiled Lisp code end in "`.elc`" instead of "`.el`". The exception to this is your `.emacs` file, which does not need the `.el` extension because Emacs knows to search for it on startup.

To load a file of Lisp code interactively, use the command `M-x load-file`. It will prompt you for the name of the file. To load Lisp files from inside other Lisp files, do this:

```
(load "c-mode") ; force Emacs to load the stuff in c-mode.el or .elc
```

Emacs will first add the `.elc` extension to the filename and try to find it somewhere in the `load-path`. If it fails, it tries it with the `.el` extension; failing that, it uses the literal string as passed to `load`. You can byte-compile a file with the command `M-x byte-compile-file`, but if you modify the file often, it's probably not worth it. You should never byte-compile your `.emacs`, though, nor even give it a `.el` extension.

After your `.emacs` has been loaded, Emacs searches for a file named `default.el` to load. Usually it's located in a directory in load-path called `site-lisp` or `local-elisp` or something (see the example `load-path` I gave a while ago). People who maintain Emacs on multi-user systems use default.el to make changes that will affect everyone's Emacs, since everybody's Emacs loads it after their personal `.emacs`. `Default.el` should not be byte-compiled either, since it tends to be modified fairly often.

If a person's `.emacs` contains any errors, Emacs will not attempt to load `default.el`, but instead will just stop, flashing a message saying "`Error in init file.`" or something. If you see this message, there's probably something wrong with your `.emacs`.

There is one more kind of expression that often goes in a `.emacs`. The Emacs Lisp library sometimes offers multiple packages for doing the same thing in different ways. This means that you have to specify which one you want to use (or you'll get the default package, which is not always the best one for all purposes). One area in which this happens is Emacs' Scheme interaction features. There are two different Scheme interfaces distributed with Emacs (in version 19 at least): `xscheme` and `cmuscheme`.

```
prompt> ls /usr/lib/emacs/19.19/lisp/*scheme*
/usr/lib/emacs/19.19/lisp/cmuscheme.el
/usr/lib/emacs/19.19/lisp/cmuscheme.elc
/usr/lib/emacs/19.19/lisp/scheme.el
/usr/lib/emacs/19.19/lisp/scheme.elc
/usr/lib/emacs/19.19/lisp/xscheme.el
/usr/lib/emacs/19.19/lisp/xscheme.elc
```

I happen to like the interface offered by `cmuscheme` much better than that offered by `xscheme`, but the one Emacs will use by default is `xscheme`. How can I cause Emacs to act in accordance with my preference? I put this in my `.emacs`:

```
;; notice how the expression can be broken across two lines.  Lisp
;; ignores whitespace, generally:
(autoload 'run-scheme "cmuscheme"
"Run an inferior Scheme, the way I like it." t)
```

The function `autoload` takes the name of a function (quoted with "`'`", for reasons having to do with how Lisp works) and tells Emacs that this function is defined in a certain file. The file is the second argument, a string (without the "`.el`" or "`.elc`" extension) indicating the name of the file to search for in the `load-path`.

The remaining arguments are optional, but necessary in this case: the third argument is a documentation string for the function, so that if you call `describe-function` on it, you get some useful information. The fourth argument tells Emacs that this autoloadable function can be called interactively (that is, by using `M-x`). This is very important in this case, because one should be able to type `M-x run-scheme` to start a scheme process running under Emacs.

Now that `run-scheme` has been defined as an autoloadable function, what happens when I type `M-x run-scheme`? Emacs looks at the function `run-scheme`, sees that it's set to be autoloaded, and loads the file named by the autoload (in this case, "`cmuscheme`"). The byte-compiled file `cmuscheme.elc` exists, so Emacs will load that. That file *must* define the function `run-scheme`, or there will be an autoload error. Luckily, it does define `run-scheme`, so everything goes smoothly, and I get my preferred Scheme interface[12].

An `autoload` is a like a promise to Emacs that, when the time comes, it can find the specified function in the file you tell it to look in. In return, you get some control over what gets loaded. Also, autoloads help cut down on Emacs' size in memory, by not loading certain features until they are asked for. Many commands are not really defined as functions when Emacs starts up. Rather, they are simply set to autoload from a certain file. If you never invoke the command, it never gets loaded. This space saving is actually vital to the functioning of Emacs: if it loaded every available file in the Lisp library, Emacs would take twenty minutes just to start up, and once it was done, it might occupy most of the available memory on your machine. Don't worry, you don't have to set all these autoloads in your `.emacs`; they were taken care of when Emacs was built.

7.13 Finding Out More

I have not told you everything there is to know about Emacs. In fact, I don't think I have even told you 1% of what there is to know about Emacs. While you know enough to get by, there are still lots of time-saving tricks and conveniences that you ought to find out about. The best way to do this is to wait until you find yourself needing something, and then look for a function that does it.

The importance of being comfortable with Emacs' online help facilities cannot be emphasized enough. For example, suppose you want to be able to insert the contents of some file into a buffer that is already working on a different file, so that the buffer contains both of them. Well, if you were to guess that there is a command called `insert-file`, you'd be right. To check your educated guess, type `C-h f`. At the prompt in the minibuffer, enter the name of a function that you want help on. Since you know that there is completion on function names, and you can guess that the command you are looking for begins with "insert", you type insert and hit $\boxed{\text{Tab}}$. This shows you all the function names that begin with "insert", and "insert-file" is one of them.

[12]By the way, `cmuscheme` was the interface I was talking about earlier, in the section on working with Scheme, so if you want to use any of the stuff from that tutorial, you need to make sure that you run `cmuscheme`.

So you complete the function name and read about how it works, and then use `M-x insert-file`. If you're wondering whether it's also bound to a key, you type `C-h w insert-file` $\boxed{\text{Return}}$, and find out. The more you know about Emacs' help facilities, the more easily you can ask Emacs questions about itself. The ability to do so, combined with a spirit of exploration and a willingness to learn new ways of doing things, can end up saving you a lot of keystrokes.

To order a copy of the Emacs user's manual and/or the Emacs Lisp Programming manual, write to:

Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
USA

Both of these manuals are distributed electronically with Emacs, in a form readable by using the Info documentation reader (`C-h i`), but you may find it easier to deal with treeware than with the online versions. Also, their prices are quite reasonable, and the money goes to a good cause — quality free software! At some point, you should type `C-h C-c` to read the copyright conditions for Emacs. It's more interesting than you might think, and will help clarify the concept of free software. If you think the term "free software" just means that the program doesn't cost anything, please do read that copyright as soon as you have time!

Chapter 8

I Gotta Be Me!

If God had known we'd need foresight, she would have given it to us.

8.1 Shell Customization

One of the distinguishing things about the Unix philosophy is that the system's designers did not attempt to predict every need that users might have; instead, they tried to make it easy for each individual user to tailor the environment to their own particular needs. This is mainly done through **configuration files**[1].

The most important configuration files are the ones used by the shell. Linux's default shell is `bash`, and that's the shell this manual will cover.

The most important configuration file for ordinary users is `.bash_profile`[2]. Each user has their own `.bash_profile`—it lives in your home directory—and it is used to customize your shell. (The shell, remember, is the go-between that allows you to communicate with the operating system itself, so it's quite natural to think of it as your "environment" and to direct customization efforts at it.) The commands in `.bash_profile` are read by `bash` when it starts up[3], and whatever it found there will be in effect for the rest of your "login session". Ex-MS-DOS users can think of

[1]Also known variously as "**init files**", "**rc files**" (for "*run control*"), or even "**dot files**" (because the filenames often begin with "`.`", so that the files aren't displayed in a normal `ls`).

[2]In, fact, due to some confusing Unix lossage, there are times when `bash` will look for a `.bash_profile`, and times when it will look for a `.bashrc`. To avoid inconsistency, just make `.bashrc` always be an exact copy of `.bash_profile`. You can do this through repeated use of the `cp` command, or you can make a link, with the command `ln ~/.bash_profile ~/.bashrc`. [Larry, am I leading them astray by doing this? Having looked over the Bash man page, I don't see any reason to initiate them into the mysteries of login vs. interactive shells, but I am not an expert in this area.]

[3]In fact, if you were using the C Shell instead of Bash, the init file would be `.cshrc`. Sigh..., one of the cutest facts about this is that the two shells use *slightly* different command languages, so that users who use both (on different systems, say) are constantly tripping themselves up. You needn't worry, however; the default shell in LINUXis `bash`, and if you know enough to be changing your default shell to C Shell, then you know enough not to be reading this section.

the `.bash_profile` as a sort of individualized AUTOEXEC.BAT (since it only affects one user, not everyone on the system).

8.1.1 Aliasing

What are some of the things you might want to customize? Here's something that I think about 90% of Bash users have put in their `.bash_profile`:

```
alias ll="ls -l"
```

That command defined a shell **alias** called `ll` that "expands" to the normal shell command "`ls -l`" when invoked by the user. So, assuming that Bash has read that command in from your `.bash_profile`, you can just type `ll` to get the effect of "`ls -l`" in only half the keystrokes. What happens is that when you type `ll` and hit Return, Bash intercepts it, because it's watching for aliases, replaces it with "ls -l", and runs that instead. There is no actual program called `ll` on the system (no binary file waiting to be executed, that is), but the shell translated the alias into a valid program. Clear? Good. Here are a zillion aliases from my own `.bash_profile`:

```
alias ls="ls -CF"
alias ll="ls -l"
alias la="ls -a"
alias rr="rm -r"
alias ro="rm *~; rm .*~"
alias rd="rmdir"
alias md="mkdir"
alias pu=pushd
alias po=popd
alias ds=dirs
alias b="~/.b"
alias to="telnet cs.oberlin.edu"
alias ta="telnet altair.mcs.anl.gov"
alias tg="telnet wombat.gnu.ai.mit.edu"
alias tko="talk kold@cs.oberlin.edu"
alias tjo="talk jimb@cs.oberlin.edu"
alias tji="talk jimb@totoro.bio.indiana.edu"
alias mroe="more"
alias moer="more"
alias email="emacs -f vm"
alias ed2="emacs -d floss:0 -fg \"grey95\" -bg \"grey50\""
```

You might have noticed a few odd things about them. First of all, I leave off the quotes in a few of the aliases, for example `rd`. Strictly speaking, quotes aren't always necessary. If you're just aliasing a single long command to give it a name that's easier to type, you can do without the quotes:

```
alias rf=refrobnicate
```

However, it never hurts to have quotes either, so don't let me get you into any bad habits. You should certainly use them if you're going to be aliasing a command with options and/or arguments:

```
alias rf="refrobnicate -verbose -prolix -wordy -o foo.out"
```

Also, the final alias has some funky quoting going on:

```
alias ed2="emacs -d floss:0 -fg \backslash"grey95\backslash" -bg \backslash"grey50\backslash""
```

As you probably guessed, I wanted to pass double-quotes in the options themselves, so I had to quote those with a backslash to prevent Bash from thinking that they signaled the end of the alias.

Finally, I have actually aliased two common typing mistakes, "mroe" and "moer", to the command I meant to type, `more`. Aliases do not interfere with your passing arguments to a program:

```
prompt> mroe hurd.txt
```

does invoke the `more` program on the file `hurd.txt`.

In fact, knowing how to make your own aliases is probably at least half of all the shell customization you'll ever do. Experiment a little, find out what long commands you find yourself typing frequently, and make aliases for them (and then don't forget to use the aliases!) You'll find that it makes working at a shell prompt a much more pleasant experience.

8.1.2 Environment Variables

The other major thing one does in a `.bash_profile` is set **environment variables**. And what are environment variables? Let's go at it from the other direction: suppose you are reading the documentation for the program `fruggle`, and you run across these sentences:

> Fruggle normally looks for its configuration file, `.frugglerc`, in the user's home directory. However, if the environment variable `FRUGGLEPATH` is set to a different filename, it will look there instead.

Hmmm, what did that mean? Well, every program executes in an **environment**, and that environment is defined by the shell that called the program[4]. The environment could be said to exist "within" the shell. Programmers have a special routine for querying the environment, and the `fruggle` program makes use of this routine. It checks the value of the environment variable `FRUGGLEPATH`. If that variable turns out to be undefined, then it will just use the file `.frugglerc` in your home directory. If it is defined, however, `fruggle` will use the variable's value (which had better be the name of a file that `fruggle` can use) instead of the default `.frugglerc`.

Enough yakking, here's how you can change your environment in Bash:

[4]Now you see why shells are so important. Imagine if you had to pass a whole environment by hand every time you called a program! It could get tiresome really fast...

```
prompt> export PGPPATH=/home/kfogel/secrets/pgp
```

You may think of the `export` command as meaning "Please export this variable out to the environment where I will be calling programs, so that its value is visible to them." There are actually reasons to call it `export`, as you'll see later.

This particular variable is used by Phil Zimmerman's infamous public-key encryption program, `pgp`. By default, `pgp` uses your home directory as a place to find certain files that it needs (ones containing encryption keys, ahem), and also as a place to store temporary files that it creates when it's running. By setting variable `PGPPATH` to this value, I have told it to use the directory `/home/kfogel/secrets/pgp` instead (I had to read the `pgp` manual to find out the exact name of the variable and what it does, but in fact it's a common convention to use the name of the program in capital letters, prepended to the suffix "PATH").

How can you check the value of an environment variable? Like this:

```
prompt> echo $PGPPATH
/home/kfogel/.pgp
prompt>
```

Notice the "$"; you prefix an environment variable with a dollar sign in order to extract the variable's value. Had you typed it without the dollar sign, `echo` would have simply echoed its argument(s):

```
prompt> echo PGPPATH
PGPPATH
prompt>
```

The "$" is used to *evaluate* environment variables, but it only does so in the context of the shell—that is, when the shell is interpreting. When is the shell interpreting? Well, when you are typing commands at the prompt, or when Bash is reading commands from a file like `.bash_profile`, it can be said to be "interpreting" the commands.

There are four variables defined automatically when you log in (meaning that you don't have to set them in your `.bash_profile`, because they have already been set by the time Bash gets around to reading its init file): **HOME**, **TERM**, **SHELL**, and **USER**. Let's check their values:

```
prompt> echo $HOME
/home/kfogel
prompt> echo $TERM
vt100
prompt> echo $SHELL
/bin/sh
prompt> echo $USER
kfogel
```

In order, these values are: the location of your home directory, your **terminal type**, the program that is running as your shell (/bin/sh, or maybe /bin/bash; in Linux they are the same thing anyway), and finally, your username. Of these, the only one whose purpose should be a mystery to you is the TERM variable. A brief history lesson is in order, though it means a digression...

The operating system, you see, needs to know certain facts about your console, in order to perform basic functions like writing a character to the screen, moving the cursor to the next line, etc. In the early days of computing, manufacturers were constantly adding new features to their terminals: first reverse-video, then maybe European character sets, eventually even primitive drawing functions (remember, these were the days before windowing systems and mice). However, all these nice features represented a problem to programmers: how were they to know what the system's terminal could support and what it couldn't, if it was changing from day to day? What eventually happened was that the Digital Equipment Corporation's VT-100 terminal became a sort of lowest-common-denominator standard. It supports all standard characters, some basic cursor movement commands, reverse-video, and a few other things. Programs counted on being able to find a vt100 terminal, and terminal manufacturers were expected to support a vt100 compatibility mode, even if their terminal had features above and beyond the standard vt100. This standard is still used today, even though most terminals can support much more, and anyway most users do their work in a graphical windowing system. The lesson: portability is more important than fancy features. Budding programmers take note.

So this is what the TERM variable is used for: programs check its value to make sure that they have a vt100, or something close to it, before they do anything that requires writing to the screen. Under LINUX, TERM's value is sometimes console, which means, I believe, a vt100-like terminal with some extra features. [Larry, or someone, please correct if that is not accurate.] Often, simple terminal problems (like garbage characters) can be fixed by issuing this command:

```
prompt> export TERM=vt100
```

There is another variable, named simply PATH, whose value is crucial to the proper functioning of the shell. Here's mine:

```
prompt> echo $PATH
/home/kfogel/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/TeX/bin
prompt>
```

Your PATH is a colon-separated list of the directories the shell should search for programs, when you type the name of a program to run. When I type ls and hit $\boxed{\text{Return}}$, for example, the Bash first looks in /home/kfogel/bin, a directory I made for storing programs that I wrote. However, I didn't write ls (in fact, I think it might have been written before I was born!). Failing to find it in /home/kfogel/bin, Bash looks next in /bin—and there it has a hit! /bin/ls does exist and is executable, so Bash stops searching for a program named ls and runs it. There might well have been another ls sitting in the directory /usr/bin, but Bash would never run it unless I asked for it by specifying an explicit pathname:

```
prompt> /usr/bin/ls
```

The PATH variable exists so that we don't have to type in complete pathnames for every command. When you type a command, Bash looks for it in the directories named in PATH, in order, and runs it if it finds it. If it doesn't find it, you get a rude error:

```
prompt> clubly
clubly: command not found
```

Notice that my PATH does not have the current directory, "`.`", in it. If it did, it might look like this:

```
prompt> echo $PATH
.:/home/kfogel/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/TeX/bin
prompt>
```

This is a matter of some debate in Unix-circles (which you are now a member of, whether you like it or not). The problem is that having the current directory in your path can be a security hole. Suppose that you `cd` into a directory where somebody has left a "Trojan Horse" program called `ls`, and you do an `ls`, as would be natural on entering a new directory. Since the current directory, "`.`", came first in your PATH, the shell would have found this version of `ls` and executed it. Whatever mischief they might have put into that program, you have just gone ahead and executed (and that could be quite a lot of mischief indeed). The person did not need root privileges to do this; they only needed write permission on the directory where the "false" `ls` was located. It might even have been their home directory, if they knew that you would be poking around in there at some point.

On your own system, it's highly unlikely that people are leaving traps for each other. All the users are probably friends or colleagues of yours. However, on a large multi-user system (like many university computers), there could be plenty of unfriendly programmers whom you've never met. Whether or not you want to take your chances by having "`.`" in your path depends on your situation; I'm not going to be dogmatic about it either way, I just want you to be aware of the risks involved[5]. Multi-user systems really are communities, where people can do things to one another in all sorts of unforseen ways.

The actual way that I set my PATH involves most of what you've learned so far about environment variables. Here is what is actually in my .bash_profile:

```
export PATH=.:${HOME}/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/TeX/bin
```

Here, I am taking advantage of the fact that the HOME variable is set before Bash reads my `.bash_profile`, by using its value in setting my PATH. The curly braces ("{...}") are a further level of quoting; they delimit the extent of what the "$" is to evaluate, so that the shell doesn't get

[5]Remember that you can always execute programs in the current directory by being explicit about it, i.e.: "`./foo`" .

confused by the text immediately following it ("`/bin`" in this case). Here is another example of the effect they have:

```
prompt> echo ${HOME}foo
/home/kfogelfoo
prompt>
```

Without the curly braces, I would get nothing

```
prompt> echo $HOMEfoo

prompt>
```

because there is no environment variable named "HOMEfoo" set.

The file `/etc/profile` serves as a kind of global `.bash_profile` that is common to all users. Having one centralized file like that makes it easier for the system administrator to add a new directory to everyone's `PATH` or something, without them all having to do it individually. Therefore, it might be best to put this in your .bash_profile:

```
export PATH=${PATH}:.:${HOME}/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/TeX/bin
```

so as not to lose any of the `PATH` directories defined in `/etc/profile`. (The reasons you didn't see me doing that in my own `.bash_profile` before are not worth going into here; 99% of the time, you'll want the `PATH` defined in `/etc/profile` to be part of your own `PATH`, so you'd best be careful to include it.)

You can also control what your prompt looks like. This is done by setting the value of the environment variable **PS1**. Personally, I want a prompt that shows me the path to the current working directory—here's how I do it in my `.bash_profile`:

```
export PS1='$PWD>'
```

Whew! As you can see, there are actually *two* variables being used here. The one being set is `PS1`, and it is being set to the value of `PWD`, which can be thought of as either "Print Working Directory" or "Path to Working Directory". But the evaluation of `PWD` takes place inside single quotes. The single quotes serve to evaluate the expression inside them, which itself evaluates the variable `PWD`. If you just did `export PS1=$PWD`, your prompt would constantly display the path to the current directory *at the time that* `PS1` *was set*, instead of constantly updating it as you change directories. Well, that's sort of confusing, and not really all that important. Just keep in mind that you need the quotes if you want the current directory displayed in your prompt.

There's a lot more to configuring your .bash_profile, and not enough room to explain it here. You can read the Bash man page for more, and also the Info pages on it (if you have them installed), and ask questions of experienced Bash users. Here is a complete `.bash_profile` for you to study; it's fairly standard and probably can't get you into any trouble, but don't be intimidated if you don't understand everything in it:

```
# some random stuff:
ulimit -c unlimited
export history_control=ignoredups
export PS1='$PWD>'
umask 022

# application-specific paths:
export MANPATH=/usr/local/man:/usr/man
export INFOPATH=/usr/local/info
export PGPPATH=${HOME}/.pgp

# make the main PATH:
homepath=${HOME}:~/bin
stdpath=/bin:/usr/bin:/usr/local/bin:/usr/ucb/:/etc:/usr/etc:/usr/games
pubpath=/usr/public/bin:/usr/gnusoft/bin:/usr/local/contribs/bin
softpath=/usr/bin/X11:/usr/local/bin/X11:/usr/TeX/bin
export PATH=.:${homepath}:${stdpath}:${pubpath}:${softpath}
# Technically, the curly braces were not necessary, because the colons
# were valid delimiters; nevertheless, the curly braces are a good
# habit to get into, and they can't hurt.  There is absolutely no need
# to be this fancy with the PATH, it's just for showboating.

# aliases
alias ls="ls -CF"
alias fg1="fg %1"
alias fg2="fg %2"
alias tba="talk sussman@tern.mcs.anl.gov"
alias tko="talk kold@cs.oberlin.edu"
alias tji="talk jimb@totoro.bio.indiana.edu"
alias mroe="more"
alias moer="more"
alias email="emacs -f vm"
alias pu=pushd
alias po=popd
alias b="~/.b"
alias ds=dirs
alias ro="rm *~; rm .*~"
alias rd="rmdir"
alias ll="ls -l"
alias la="ls -a"
alias rr="rm -r"
alias md="mkdir"
alias ed2="emacs -d floss:0 -fg \"grey95\" -bg \"grey50\""

function gco
{
  gcc -o $1 $1.c -g
}
```

8.2 X Windows Init Files

Most people prefer to do their work inside a graphical environment, and for Unix machines, that usually means using X Windows. If you're accustomed to the Macintosh or to Microsoft Windows, the X Window System may take a little getting used to, especially in how it is customized.

With a Mac or MS-Windows, you customize the environment from *within* the environment: if you want to change your background, for example, you do by clicking on the new color in some special graphical setup program. In X Windows, system defaults are controlled by text files, which you edit directly—in other words, you'd type the actual color name into a file in order to set your background to that color.

There is no denying that this method just isn't as slick as some commercial windowing systems. I think this tendency to remain text-based, even in a graphical environment, has to do with the fact that X Windows was created by a bunch of programmers who simply weren't trying to write software that their grandparents could use. This tendency may change in future versions of X Windows (at least I hope it will), but for now, you just have to learn to deal with more text files. It does at least give you very flexible and precise control over your configuration.

Here are the most important files for configuring X Windows:

`.xinitrc`	A script run by X Windows when it starts up.
`.twmrc`	Read by an X Windows window manager, `twm`.
`.fvwmrc`	Read by an X Windows window manager, `fvwm`.

All of these files should be located in your home directory, if they exist at all.

The `.xinitrc` is a simple shell script that gets run when X Windows is invoked. It can do anything any other shell script can do, but of course it makes the most sense to use it for starting up various X Windows programs and setting window system parameters. The last command in the `.xinitrc` is usually the name of a window manager to run, for example `/usr/bin/X11/twm` (short for the "Twm Window Manager", in case anyone ever asks you).

What sort of thing might you want to put in a `.xinitrc` file? Perhaps some calls to the `xsetroot` program, to make your root (background) window and mouse cursor look the way you want them to look. Calls to `xmodmap`, which tells the server[6] how to interpret the signals from your keyboard. Any other programs you want started every time you run X Windows (for example, `xclock`).

Here is some of my `.xinitrc`; yours will almost certainly look different, so this is meant only as

[6] The "server" just means the main X Windows process on your machine, the one with which all other X programs must communicate in order to use the display. These other programs are known as "clients", and the whole deal is called a "client-server" system.

an example:

```
#!/bin/sh
# The first line tells the operating system which shell to use in
# interpreting this script.  The script itself ought to be marked as
# executable; you can make it so with "chmod +x ~/.xinitrc".

# xmodmap is a program for telling the X server how to interpret your
# keyboard's signals.  It is *definitely* worth learning about. You
# can do "man xmodmap", "xmodmap -help", "xmodmap -grammar", and more.
# I don't guarantee that the expressions below will mean anything on
# your system (I don't even guarantee that they mean anything on
# mine):
xmodmap -e 'clear Lock'
xmodmap -e 'keycode 176 = Control_R'
xmodmap -e 'add control = Control_R'
xmodmap -e 'clear Mod2'
xmodmap -e 'add Mod1 = Alt_L Alt_R'

# xset is a program for setting some other parameters of the X server:
xset m 3 2 &        # mouse parameters
xset s 600 5 &      # screen saver prefs
xset s noblank &    # ditto
xset fp+ /home/kfogel/x/fonts # for cxterm
# To find out more, do "xset -help".

# Tell the X server to superimpose fish.cursor over fish.mask, and use
# the resulting pattern as my mouse cursor:
xsetroot -cursor /home/lab/kfogel/x/fish.cursor /home/lab/kfogel/x/fish.mask &

# a pleasing background pattern and color:
xsetroot -bitmap /home/lab/kfogel/x/pyramid.xbm -bg tan

# todo: xrdb here?  What about .Xdefaults file?

# You should do "man xsetroot", or "xsetroot -help" for more
# information on the program used above.

# A client program, the imposing circular color-clock by Jim Blandy:
/usr/local/bin/circles &

# Maybe you'd like to know have a clock on your screen at all times?
/usr/bin/X11/xclock -digital &

# Allow client X programs running at occs.cs.oberlin.edu to display
# themselves here, do the same thing for juju.mcs.anl.gov:
xhost occs.cs.oberlin.edu
xhost juju.mcs.anl.gov

# You could simply tell the X server to allow clients running on any
# other host (a host being a remote machine) to display here, but this
# is a security hole -- those clients might be run by someone else,
# and watch your keystrokes as you type your password or something!
# However, if you wanted to do it anyway, you could use a "+" to stand
# for all possible hostnames, instead of a specific hostname, like
# this:
# xhost +

# And finally, run the window manager:
/usr/bin/X11/twm
# Some people prefer other window managers.  I use twm, but fvwm is
# often distributed with Linux too:
```

It seems to work fine either way; I'm wondering if there's any point recommending an "&" or not. -Karl]

Notice that some commands are run in the background (i.e.: they are followed with a "&"), while others aren't. The distinction is that some programs will start when you start X Windows and keep going until you exit—these get put in the background. Others execute once and then exit immediately. `xsetroot` is one such; it just sets the root window or cursor or whatever, and then exits.

Once the window manager has started, it will read its own init file, which controls things like how your menus are set up, which positions windows are brought up at, icon control, and other earth-shakingly important issues. If you use `twm`, then this file is `.twmrc` in your home directory. If you use `fvwm`, then it's `.fvwmrc`, etc. I'll deal with only those two, since they're the window managers you'll be most likely to encounter with Linux.

The `.twmrc` is not a shell script—it's actually written in a language specially made for `twm`, believe it or not![7] The main thing people like to play with in their `.twmrc` is window style (colors

[7] This is one of the harsh facts about init files: they generally each have their own idiosyncratic command language. This means that users get very good at learning command languages quickly. I suppose that it would have been nice if early Unix programmers had agreed on some standard init file format, so that we wouldn't have to learn new syntaxes all the time, but to be fair it's hard to predict what kinds of information programs will need.

and such), and making cool menus, so here's an example `.twmrc` that does that:

```
# Set colors for the various parts of windows.  This has a great
# impact on the "feel" of your environment.
Color
{
    BorderColor "OrangeRed"
    BorderTileForeground "Black"
    BorderTileBackground "Black"
    TitleForeground "black"
    TitleBackground "gold"
    MenuForeground "black"
    MenuBackground "LightGrey"
    MenuTitleForeground "LightGrey"
    MenuTitleBackground "LightSlateGrey"
    MenuShadowColor "black"
    IconForeground "DimGray"
    IconBackground "Gold"
    IconBorderColor "OrangeRed"
    IconManagerForeground "black"
    IconManagerBackground "honeydew"
}

# I hope you don't have a monochrome system, but if you do...
Monochrome
{
    BorderColor "black"
    BorderTileForeground "black"
    BorderTileBackground "white"
    TitleForeground "black"
    TitleBackground "white"
}

# I created beifang.bmp with the program "bitmap".  Here I tell twm to
# use it as the default highlight pattern on windows' title bars:
Pixmaps
{
    TitleHighlight "/home/kfogel/x/beifang.bmp"
}

# Don't worry about this stuff, it's only for power users :-)
BorderWidth     2
TitleFont       "-adobe-new century schoolbook-bold-r-normal--14-140-75-75-p-87-iso8859-1"
MenuFont        "6x13"
IconFont        "lucidasans-italic-14"
ResizeFont      "fixed"
Zoom 50
RandomPlacement

# These programs will not get a window titlebar by default:
NoTitle
{
  "stamp"
  "xload"
  "xclock"
  "xlogo"
  "xbiff"
  "xeyes"
  "oclock"
  "xoid"
}
```

that some decent example `.twmrc` files came with your X Windows. Take a look in the directory `/usr/lib/X11/twm/` or `/usr/X11/lib/X11/twm` and see what's there.

One bug to watch out for with `.twmrc` files is forgetting to put the & after a command on a menu. If you notice that X Windows just freezes when you run certain commands, chances are that this is the cause. Break out of X with Control-Alt-Backspace, edit your `.twmrc`, and try again.

If you are using `fvwm`, the directory `/usr/lib/X11/fvwm/` (or `/usr/X11/lib/X11/fvwm/`) has some good example config files in it, as well.

[Folks: I don't know anything about fvwm, although I might be able to grok something from the example config files. Then again, so could the reader :-). Also, given the decent but small system.twmrc in the above-mentioned directory, I wonder if it's worth it for me to provide that lengthy example with my own .twmrc. It's in for now, but I don't know whether we want to leave it there or not. -Karl]

8.3 Other Init Files

Some other initialization files of note are:

`.emacs`	Read by the Emacs text editor when it starts up.
`.netrc`	Gives default login names and passwords for ftp.
`.rhosts`	Makes your account remotely accessible.
`.forward`	For automatic mail forwarding.

8.3.1 The Emacs Init File

If you use `emacs` as your primary editor, then the `.emacs` file is quite important. It is dealt with at length in Chapter 7.

8.3.2 FTP Defaults

Your .netrc file allows you to have certain **ftp** defaults set before you run **ftp**. Here is a small sample **.netrc**:

```
    machine floss.life.uiuc.edu login kfogel password fishSticks
    machine darwin.life.uiuc.edu login kfogel password fishSticks
    machine geta.life.uiuc.edu login kfogel password fishSticks
    machine phylo.life.uiuc.edu login kfogel password fishSticks
    machine ninja.life.uiuc.edu login kfogel password fishSticks
    machine indy.life.uiuc.edu login kfogel password fishSticks

    machine clone.mcs.anl.gov login fogel password doorm@
    machine osprey.mcs.anl.gov login fogel password doorm@
    machine tern.mcs.anl.gov login fogel password doorm@
    machine altair.mcs.anl.gov login fogel password doorm@
    machine dalek.mcs.anl.gov login fogel password doorm@
    machine juju.mcs.anl.gov login fogel password doorm@

    machine sunsite.unc.edu login anonymous password kfogel@cs.oberlin.edu
```

Each line of your **.netrc** specifies a machine name, a login name to use by default for that machine, and a password. This is a great convenience if you do a lot of **ftp**-ing and are tired of constantly typing in your username and password at various sites. The **ftp** program will try to log you in automatically using the information found in your **.netrc** file, if you **ftp** to one of the machines listed in the file.

You can tell **ftp** to ignore your .netrc and not attempt auto-login by invoking it with the **-n** option: "**ftp -n**".

You must make sure that your **.netrc** file is readable *only* by you. Use the **chmod** program to set the file's read permissions. If other people can read it, that means they can find out your password at various other sites. This is about as big a security hole as one can have; to encourage you to be careful, **ftp** and other programs that look for the **.netrc** file will actually refuse to work if the read permissions on the file are bad.

There's more to the **.netrc** file than what I've said; when you get a chance, do "**man .netrc**" or "**man ftp**".

8.3.3 Allowing Easy Remote Access to Your Account

If you have an **.rhosts** file in your home directory, it will allow you to run programs on this machine remotely. That is, you might be logged in on the machine **cs.oberlin.edu**, but with a correctly configured **.rhosts** file on floss.life.uiuc.edu, you could run a program on **floss.life.uiuc.edu** and have the output go to **cs.oberlin.edu**, without ever having to log in or type a password.

A `.rhosts` file looks like this:

```
frobnozz.cs.knowledge.edu jsmith
aphrodite.classics.hahvaahd.edu wphilps
frobbo.hoola.com trixie
```

The format is fairly straightforward: a machine name, followed by username. Suppose that that example is in fact my `.rhosts` file on `floss.life.uiuc.edu`. That would mean that I could run programs on floss, with output going to any of the machines listed, as long as I were also logged in as the corresponding user given for that machine when I tried to do it.

The exact mechanism by which one runs a remote program is usually the `rsh` program. It stands for "remote shell", and what it does is start up a shell on a remote machine and execute a specified command. For example:

```
frobbo$ whoami
trixie
frobbo$ rsh floss.life.uiuc.edu "ls ~"
foo.txt    mbox   url.ps    snax.txt
frobbo$ rsh floss.life.uiuc.edu "more ~/snax.txt"
[snax.txt comes paging by here]
```

User trixie at floss.life.uiuc.edu, who had the example `.rhosts` shown previously, explicitly allows trixie at frobbo.hoola.com to run programs as trixie from floss.

You don't have to have the same username on all machines to make a `.rhosts` work right. Use the "`-l`" option to `rsh`, to tell the remote machine what username you'd like to use for logging in. If that username exists on the remote machine, and has a `.rhosts` file with your current (i.e.: local) machine and username in it, then your `rsh` will succeed.

```
frobbo$ whoami
trixie
frobbo$ rsh -l kfogel floss.life.uiuc.edu "ls ~"
[Insert a listing of my directory on floss here]
```

This will work if user `kfogel` on `floss.life.uiuc.edu` has a `.rhosts` file which allows `trixie` from `frobbo.hoopla.com` to run programs in his account. Whether or not they are the same person is irrelevant: the only important things are the usernames, the machine names, and the entry in kfogel's `.rhosts` file on floss. Note that trixie's `.rhosts` file on frobbo doesn't enter into it, only the one on the remote machine matters.

There are other combinations that can go in a `.rhosts` file—for example, you can leave off the username following a remote machine name, to allow any user from that machine to run programs as you on the local machine! This is, of course, a security risk: someone could remotely run a program that removes your files, just by virtue of having an account on a certain machine. If you're going to do things like leave off the username, then you ought to make sure that your `.rhosts` file is readable by you and no one else.

8.3.4 Mail Forwarding

You can also have a `.forward` file, which is not strictly speaking an "init file". If it contains an email address, then all mail to you will be forwarded to that address instead. This is useful when you have accounts on many different systems, but only want to read mail at one location.

There is a host of other possible initialization files. The exact number will vary from system to system, and is dependent on the software installed on that system. One way to learn more is to look at files in your home directory whose names begin with "`.`". These files are not all guaranteed to be init files, but it's a good bet that most of them are.

8.4 Seeing Some Examples

The ultimate example I can give you is a running Linux system. So, if you have Internet access, feel free to telnet to **`floss.life.uiuc.edu`**. Log in as "guest", password "explorer", and poke around. Most of the example files given here can be found in **`/home/kfogel`**, but there are other user directories as well. You are free to copy anything that you can read. Please be careful: floss is not a terribly secure box, and you can almost certainly gain root access if you try hard enough. I prefer to rely on trust, rather than constant vigilance, to maintain security.

Chapter 9

Funny Commands

Well, most people who had to do with the UNIX commands exposed in this chapter will not agree with this title. "What the heck! You have just shown me that the Linux interface is very standard, and now we have a bunch of commands, each one working in a completely different way. I will never remember all those options, and you are saying that they are *funny*?" Yes, you have just seen an example of hackers' humor. Besides, look at it from the bright side: there is no MS-DOS equivalent of these commands. If you need them, you have to purchase them, and you never know how their interface will be. Here they are a useful – and inexpensive – add-on, so enjoy!

The set of commands dwelled on in this chapter covers `find`, which lets the user search in the directory tree for specified groups of files; `tar`, useful to create some archive to be shipped or just saved; `dd`, the low-level copier; and `sort`, which ... yes, sorts files. A last proviso: these commands are by no means standardized, and while a core of common options could be found on all $*$IX systems, the (GNU) version which is explained below, and which you can find in your Linux system, has usually many more capabilities. So if you plan to use other UNIX-like operating systems, please don't forget to check their man page in the target system to learn the maybe not-so-little differences.

9.1 `find`, the file searcher

9.1.1 Generalities

Among the various commands seen so far, there were some which let the user recursively go down the directory tree in order to perform some action: the canonical examples are `ls -R` and `rm -R`. Good. `find` is *the* recursive command. Whenever you are thinking "Well, I have to do so-and-so on all those kind of files in my own partition", you have better think about using `find`. In a certain sense the fact that `find` finds files is just a side effect: its real occupation is to evaluate

The basic structure of the command is as follows:

`find` *path* [...] *expression* [...]

This at least on the GNU version; other version do not allow to specify more than one path, and besides it is very uncommon the need to do such a thing. The rough explanation of the command syntax is rather simple: you say from where you want to start the search (the *path* part; with GNU find you can omit this and it will be taken as default the current directory `.`), and which kind of search you want to perform (the *expression* part).

The standard behavior of the command is a little tricky, so it's worth to note it. Let's suppose that in your home directory there is a directory called garbage, containing a file foobar. You happily type `find . -name foobar` (which as you can guess searches for files named foobar), and you obtain ...nothing else than the prompt again. The trouble lies in the fact that `find` is by default a silent command; it just returns 0 if the search was completed (with or without finding anything) or a non-zero value if there had been some problem. This does not happen with the version you can find on Linux, but it is useful to remember it anyway.

9.1.2 Expressions

The *expression* part can be divided itself in four different groups of keywords: *options*, *tests*, *actions*, and *operators*. Each of them can return a true/false value, together with a side effect. The difference among the groups is shown below.

options affect the overall operation of find, rather than the processing of a single file. An example is -follow, which instructs `find` to follow symbolic links instead of just stating the inode. They always return true.

tests are real tests (for example, -empty checks whether the file is empty), and can return true or false.

actions have also a side effect the name of the considered file. They can return true or false too.

operators do not really return a value (they can conventionally be considered as true), and are used to build compress expression. An example is -or, which takes the logical OR of the two subexpressions on its side. Notice that when juxtaposing expression, a -and is implied.

Note that `find` relies upon the shell to have the command line parsed; it means that all keyword must be embedded in white space and especially that a lot of nice characters have to be escaped, otherwise they would be mangled by the shell itself. Each escaping way (backslash, single and double quotes) is OK; in the examples the single character keywords will be usually quoted with backslash, because it is the simplest way (at least in my opinion. But it's me who is writing these notes!)

9.1.3 Options

Here there is the list of all options known by GNU version of `find`. Remember that they always return true.

- -daystart measures elapsed time not from 24 hours ago but from last midnight. A true hacker probably won't understand the utility of such an option, but a worker who programs from eight to five does appreciate it.
- -depth processes each directory's contents before the directory itself. To say the truth, I don't know many uses of this, apart form an emulation of `rm -F` command (of course you cannot delete a directory before all files in it are deleted too ...
- -follow deferences (that is, follows) symbolic links. It implies option -noleaf; see below.
- -noleaf turns off an optimization which says "A directory contains two fewer subdirectories than their hard link count". If the world were perfect, all directories would be referenced with their name on the father directory, as `.` on itself – thus the value two above –

  possible exceptions: a non-UNIX NFS-mounted filesystem, and symbolic links. Life is hard, sometimes.
- -maxdepth *levels*, -mindepth *levels*, where *levels* is a non-negative integer, respectively say that at most or at least *levels* levels of directories should be searched. A couple of examples is mandatory: `-maxdepth 0` indicates that it the command should be performed just on the arguments in the command line, i.e., without recursively going down the directory tree; `-mindepth 1` inhibits the processing of the command for the arguments in the command line, while all other files down are considered.
- -version just prints the current version of the program.
- -xdev, which is a misleading name, instructs `find` **not** to cross device, i.e. changing filesystem. It is very useful when you have to search for something in the root filesystem; in many machines it is a rather small partition, but a `find /` would otherwise search the whole structure!

9.1.4 Tests

The first two tests are very simple to understand: -false always return false, while -true always return true. Other tests which do not need the specification of a value are -empty, which returns true whether the file is empty, and the couple -nouser / -nogroup, which return true in the case that no entry in `/etc/passwd` or `/etc/group` match the user/group id of the file owner. This is a common thing which happens in a multiuser system; a user is deleted, but files owned by her remain in the strangest part of the filesystems, and due to Murphy's laws take a lot of space.

Of course, it is possible to search for a specific user or group. The tests are -uid *nn* and -gid *nn*. Unfortunately it is not possibile to give directly the user name, but it is necessary to use the numeric id, *nn*.

allowed to use the forms $+nn$, which means "a value strictly greater than nn", and $-nn$, which means "a value strictly less than nn". This is rather silly in the case of UIDs, but it will turn handy with other tests.

Another useful option is -type *c*, which returns true if the file is of type *c*. The mnemonics for the possible choices are the same found in `ls`; so we have **b** when the file is a block special; **c** when the file is character special; **d** for directories; **p** for named pipes; **l** for symbolic links, and **s** for sockets. Regular files are indicated with **f**. A related test is `-xtype`, which is similar to -type except in the case of symbolic links. If -follow has not been given, the file pointed at is checked, instead of the link itself. Completely unrelated is the test `-fstype` *type*. In this case, the filesystem type is checked. I think that the information is got from file `/etc/mtab`, the one stating the mounting filesystems; I am certain that types nfs, tmp, msdos and ext2 are recognized.

Tests `-inum` *nn* and `-links` *nn* check whether the file has inode number *nn*, or *nn* links, while `-size` *nn* is true if the file has *nn* 512-bytes blocks allocated. (well, not precisely: for sparse files unallocated blocks are counted too). As nowadays the result of `ls -s` is not always measured in 512-bytes chunks (Linux for example uses 1k as the unit), it is possible to append to *nn* the character *b*, which means to count in butes, or *k*, to count in kilobytes.

Permission bits are checked through the test `-perm` *mode*. If *mode* has no leading sign, then the permission bits of the file must exactly match them. A leading − means that all permission bits must be set, but makes no assumption for the other; a leading + is satisfied just if any of the bits are set. Oops! I forgot saying that the mode is written in octal or symbolically, like you use them in `chmod`.

Next group of tests is related to the time in which a file has been last used. This comes handy when a user has filled his space, as usually there are many files he did not use since ages, and whose meaning he has forgot. The trouble is to locate them, and `find` is the only hope in sight. `-atime` *nn* is true if the file was last accessed *nn* days ago, `-ctime` *nn* if the file status was last changed *nn* days ago – for example, with a `chmod` – and `-mtime` *nn* if the file was last modified *nn* days ago. Sometimes you need a more precise timestamp; the test `-newer` *file* is satisfied if the file considered has been modified later than *file*. So, you just have to use `touch` with the desidered date, and you're done. GNU find add the tests `-anewer` and `-cnewer` which behave similarly; and the tests `-amin`, `-cmin` and `-mmin` which count time in minutes instead than 24-hours periods.

Last but not the least, the test I use more often. `-name` *pattern* is true if the file name exactly matches *pattern*, which is more or less the one you would use in a standard `ls`. Why 'more or less'? Because of course you have to remember that all the parameters are processed by the shell, and those lovely metacharacters are expanded. So, a test like `-name foo*` won't return what you want, and you should either write `-name foo` or `-name "foo*"`. This is probably one of the most common mistakes made by careless users, so write it in BIG letters on your screen. Another problem is that, like with `ls`, leading dots are not recognized. To cope with this, you can use test `-path` *pattern* which does not worry about dot and slashes when comparing the path of the considered file with *pattern*.

9.1.5 Actions

I have said that actions are those which actually do something. Well, `-prune` rather does not do something, i.e. descending the directory tree (unless -depth is given). It is usally find together with -fstype, to choose among the various filesystems which should be checked.

The other actions can be divided into two broad categories;

- Actions which *print* something. The most obvious of these – and indeed, the default action of `find` – is -print which just print the name of the file(s) matching the other conditions in the command line, and returns true. A simple variants of -print is -fprint *file*, which uses *file* instead of standard output, -ls lists the current file in the same format as `ls -dils`; -printf *format* behaves more or less like C function printf(), so that you can specify how the output should be formatted, and -fprintf *file format* does the same, but writing on *file*. These action too return true.

- Actions which *execute* something. Their syntax is a little odd and they are used widely, so please look at them.

  -exec *command* `\;` the command is executed, and the action returns true if its final status is 0, that is regular execution of it. The reason for the `\;` is rather logical: `find` does not know where the command ends, and the trick to put the exec action at the end of the command is not applicable. Well, the best way to signal the end of the command is to use the character used to do this by the shell itself, that is '`;`', but of course a semicolon all alone on the command line would be eaten by the shell and never sent to `find`, so it has to be escaped. The second thing to remember is how to specify the name of the current file within *command*, as probably you did all the trouble to build the expression to do something, and not just to print `date`. This is done by means of the string `{}`. Some old versions of `find` require that it must be embedded in white space – not very handy if you needed for example the whole path and not just the file name – but with GNU find could be anywhere in the string composing *command*. And shouldn't it be escaped or quoted, you surely are asking? Amazingly, I never had to do this neither under tcsh nor under bash (sh does not consider `{` and `}` as special characters, so it is not much of a problem). My idea is that the shells "know" that `{}` is not an option making sense, so they do not try to expand them, luckily for `find` which can obtain it untouched.

  -ok *command* `\;` behaves like -exec, with the difference that for each selected file the user is asked to confirm the command; if the answer starts with `y` or `Y`, it is executed, otherwise not, and the action returns false.

9.1.6 Operators

There are a number of operators; here there is a list, in order of decreasing precedence.

\(*expr* \)
: forces the precedence order. The parentheses must of course be quoted, as they are meaningful for the shell too.

! *expr*
-not *expr*
: change the truth value of expression, that is if *expr* is true, it becomes false. The exclamation mark needn't be escaped, because it is followed by a white space.

expr1 expr2
expr1 -a *expr2*
expr1 -and *expr2*

all correspond to the logical AND operation, which in the first and most common case is implied. *expr2* is not evaluated, if *expr1* is false.

expr1 `-o` *expr2*
expr1 `-or` *expr2*
correspond to the logical OR operation. *expr2* is not evaluated, if *expr1* is true.

expr1 , *expr2*
is the list statement; both *expr1* and *expr2* are evaluated (together with all side effects, of course!), and the final value of the expression is that of *expr2*.

9.1.7 Examples

Yes, `find` has just too many options, I know. But there are a lot of cooked instances which are worth to remember, because they are usen very often. Let's see some of them.

```
% find . -name foo\* -print
```

finds all file names starting with `foo`. If the string is embedded in the name, probably it is more sensitive to write something like `"*foo*"`, rather than `foo`.

```
% find /usr/include -xtype f -exec grep foobar \
          /dev/null {} \;
```

is a grep executed recursively starting from directory /usr/include. In this case, we are interested both in regular file and in symbolic links which point to regular files, hence the -xtype test. Many times it is simpler to avoid specyfing it, especially if we are rather sure no binary file contains the wanted string. And why the /dev/null in the command? It's a trick to force grep to write the file name where a match has been found. The command grep is applied to each file in a different invocation, and so it doesn't think it is necessary to output the file name. But now there are *two* files, i.e. the current one and `/dev/null`! Another possibility should be to pipe the command to `xargs` and let it perform the grep. I just tried it, and completely smashed my filesystem (together with these notes which I am tring to recover by hand `:-(`).

```
% find /  -atime +1 -fstype ext2 -name core \
             -exec rm {} \;
```

is a classical job for crontab. It deletes all file named `core` in filesystems of type ext2 which have not been accessed in the last 24 hours. It is possible that someone wants to use the core file to perform a post mortem dump, but nobody could remember what he was doing after 24 hours...

```
% find /home -xdev -size +500k -ls > piggies
```

is useful to see who has those files who clog the filesystem. Note the use of -xdev; as we are interested in just one filesystem, it is not necessary to descend other filesystems mounted under `/home`.

9.1.8 A last word

Keep in mind that `find` is a very time consuming command, as it has to access each and every inode of the system in order to perform its operation. It is therefore wise to combine how many operations you need in a unique invocation of `find`, especially in the 'housekeeping' jobs usually ran via a crontab job. A enlightening example is the following: let's suppose that we want to delete files ending in `.BAK` and change the protection of all directories to 771 and that of all files ending in `.sh` to 755. And maybe we are mounting NFS filesystems on a dial-up link, and we'd like not to check for files there. Why writing three different commands? The most effective way to accomplish the task is this:

```
% find . \( -fstype nfs -prune \) -o \
         \( -type d       -a -exec chmod 771 {} \; \) -o \
         \( -name "*.BAK" -a -exec /bin/rm {}   \; \) -o \
         \( -name "*.sh"  -a -exec chmod 755 {} \; \)
```

It seems ugly (and with much abuse of backslashes!), but looking closely at it reveals that the underlying logic is rather straightforward. Remember that what is really performed is a true/false evaluation; the embedded command is just a side effect. But this means that it is performed only if `find` must evaluate the exec part of the expression, that is only if the left side of the subexpression evaluates to true. So, if for example the file considered at the moment is a directory then the first exec is evaluated and the permission of the inode is changed to 771; otherwise it forgets all and steps to the next subexpression. Probably it's easier to see it in practice than to writing it down; but after a while, it will become a natural thing.

9.2 `tar`, the tape archiver

9.2.1 Introduction

9.2.2 Main options

9.2.3 Modifiers

9.2.4 Examples

9.3 `dd`, the data duplicator

Legend says that back in the mists of time, when the first UNIX was created, its developers needed a low level command to copy data between devices. As they were in a hurry, they decided to borrow the syntax used by IBM-360 machines, and to develop later an interface consistent with that of the other commands. Time passed, and all were so used with the odd way of using `dd` that it stuck. I don't know whether it is true, but it is a nice story to tell.

9.3.1 Options

To say the truth, `dd` it's not completely unlike the other Unix command: it is indeed a **filter**, that is it reads by default from the standard input and writes to the standard output. So if you just type dd at the terminal it remains quiet, waiting for input, and a ctrl-C is the only sensitive thing to type.

The syntax of the command is as follows:

```
dd [if=file] [of=file] [ibs=bytes] [obs=bytes]
   [bs=bytes] [cbs=bytes] [skip=blocks] [seek=blocks]
   [count=blocks] [conv={ascii,ebcdic,ibm,block,
       unblock,lcase,ucase,swab,noerror,notrunc,sync}]
```

so all options are of the form *option=value*. No space is allowed either before or after the equal sign; this used to be annoying, because the shell did not expand a filename in this situation, but the version of bash present in Linux is rather smart, so you don't have to worry about that. It is important also to remember that all numbered values (`bytes` and `blocks` above) can be followed by a multiplier. The possible choices are **b** for block, which multiplies by 512, **k** for kilobytes (1024), **w** for word (2), and **xm** multiplies by **m**.

The meaning of options if explained below.

- if=*filein* and of=*fileout* instruct `dd` to respectively read from *filein* and write to *fileout*. In the latter case, the output file is truncated to the value given to `seek`, or if the keyword is not present, to 0 (that is deleted), before performing the operation. But look below at option `notrunc`.
- ibs=*nn* and obs=*nn* specify how much bytes should be read or write at a time. I think that the default is 1 block, i.e. 512 bytes, but I am not very sure about it: certainly it works that way with plain files. These parameters are very important when using special devices as input or output; for example, reading from the net should set `ibs` at 10k, while a high density 3.5" floppy has as its natural block size 18k. Failing to set these values could result not only in longer time to perform the command, but even in timeout errors, so be careful.
- bs=*nn* both reads and writes *nn* bytes at a time. It overrides `ibs` and `obs` keywords.
- cbs=*nn* sets the conversion buffers to *nn* bytes. This buffer is used when translating from ASCII to EBCDIC, or from an unblocked device to a blocked one. For example, files created under VMS have often a block size of 512, so you have to set `cbs` to 1b when reading a foreign VMS tape. Hope that you don't have to mess with these things!
- skip=*nbl* and seek=*nbl* tell the program to skip *nbl* blocks respectively at the beginning of input and at the beginning of output. Of course the latter case makes sense if conversion `notrunc` is given, see below. Each block's size is the value of `ibs` (`obs`). Beware: if you did not set `ibs` and write `skip=1b` you are actually skipping 512×512 bytes, that is 256KB. It was not precisely what you wanted, wasn't it?

- count=*nbl* means to copy only *nbl* blocks from input, each of the size given by `ibs`. This option, together with the previous, turns useful if for example you have a corrupted file and you want to recover how much it is possible from it. You just skip the unreadable part and get what remains.

- conv=*conversion*,[*conversion*...] convert the file as specified by its argument. Possible conversions are ascii, which converts from EBCDIC to ASCII; ebcdic and ibm, which both perform an inverse conversion (yes, there is not a unique conversion from EBCDIC to ASCII! The first is the standard one, but the second works better when printing files on a IBM printer); block, which pads newline-terminated records to the size of cbs, replacing newline with trailing spaces; unblock, which performs the opposite (eliminates trailing spaces, and replaces them with newline); lcase and ucase, to convert test to lowercase and uppercase; swab, which swaps every pair of input bytes (for example, to use a file containing short integers written on a 680x0 machine in an Intel-based machine you need such a conversion); noerror, to continue processing after read errors; sync, which pads input block to the size of `ibs` with trailing NULs.

9.3.2 Examples

The canonical example is the one you have probably bumped at when you tried to create the first Linux diskette: how to write to a floppy without a MS-DOS filesystem. The solution is simple:

```
% dd if=disk.img of=/dev/fd0 obs=18k count=80
```

I decided not to use `ibs` because I don't know which is the better block size for a hard disk, but in this case no harm would have been if instead of `obs` I use `bs` – it could even be a trifle quicker. Notice the explicitation of the number of sectors to write (18KB is the occupation of a sector, so `count` is set to 80) and the use of the low-level name of the floppy device.

Another useful application of dd is related to the network backup. Let's suppose that we are on machine *alpha* and that on machine *beta* there is the tape unit `/dev/rst0` with a tar file we are interested in getting. We have the same rights on both machines, but there is no space on *beta* to dump the tar file. In this case, we could write

```
% rsh beta 'dd if=/dev/rst0 ibs=8k obs=20k' | tar xvBf -
```

to do in a single pass the whole operation. In this case, we have used the facilities of `rsh` to perform the reading from the tape. Input and output sizes are set to the default for these operations, that is 8KB for reading from a tape and 20KB for writing to ethernet; from the point of view of the other side of the tar, there is the same flow of bytes which could be got from the tape, except the fact that it arrives in a rather erratic way, and the option B is necessary.

I forgot: I don't think at all that dd is an acronym for “data duplicator”, but at least this is a nice way to remember its meaning ...

9.4 `sort`, the data sorter

9.4.1 Introduction

9.4.2 Options

9.4.3 Examples

Chapter 10

Errors, Mistakes, Bugs, and Other Unpleasantries

Unix was never designed to keep people from doing stupid things, because that policy would also keep them from doing clever things.
Doug Gwyn

10.1 Avoiding Errors

Many users report frustration with the Unix operating system at one time or another, frequently because of their own doing. A feature of the Unix operating system that many users' love when they're working well and hate after a late-night session is how very few commands ask for confirmation. When a user is awake and functioning, they rarely think about this, and it is an assest since it let's them work smoother.

However, there are some disadvantages. `rm` and `mv` never ask for confirmation and this frequently leads to problems. Thus, let's go through a small list that might help you avoid total disaster:

- Keep backups! This applies especially to the one user system—all system adminstrators should make regular backups of their system! Once a week is good enough to salvage many files. See the *The* LINUX *System Adminstrator's Guide*for more information.
- Individual user's should keep there own backups, if possible. If you use more than one system regularly, try to keep updated copies of all your files on each of the systems. If you have access to a floppy drive, you might want to make backups onto floppies of your critical material. At worst, keep additional copies of your most important material lying around your account *in a seperate directory*!
- Think about commands, especially "destructive" ones like `mv`, `rm`, and `cp` before you act. You also have to be careful with redirection (`>`)—it'll overwrite your files when you aren't paying attention. Even the most harmless of commands can become sinister:

```
/home/larry/report# cp report-1992 report-1993 backups
```

can easily become disaster:

```
/home/larry/report# cp report-1992 report-1993
```

- The author also recommends, from his personal experience, not to do file maintanence late at night. Does you directory structure look a little messy at 1:32am? Let it stay—a little mess never hurt a computer.
- Keep track of your present directory. Sometimes, the prompt you're using doesn't display what directory you are working in, and danger strikes. It is a sad thing to read a post on `comp.unix.admin`[1] about a `root` user who was in `/` instead of `/tmp`! For example:

```
mousehouse> pwd
/etc
mousehouse> ls /tmp
passwd
mousehouse> rm passwd
```

10.2 Not Your Fault

Unfortunately for the programmers of the world, not all problems are caused by user-error. Unix and Linux are complicated systems, and all known versions have bugs. Sometimes these bugs are hard to find and only appear under certain circumstances.

First of all, what is a bug? An example of a bug is if you ask the computer to compute "5+3" and it tells you "7". Although that's a trivial example of what can go wrong, most bugs in computer programs involve arithmetic in some extremely strange way.

10.2.1 When Is There a Bug

If the computer gives a wrong answer (verify that the answer is wrong!) or crashes, it is a bug. If any one program crashes or gives an operating system error message, it is a bug.

If a command never finishes running can be a bug, but you must make sure that you didn't tell it to take a long time doing whatever you wanted it to do. Ask for assistance if you didn't know what the command did.

Some messages will alert you of bugs. Some messages are not bugs. Check Section 3.3 and any other documentation to make sure they aren't normal informational messages. For instance, messages like "disk full" or "lp0 on fire" aren't software problems, but something wrong with your hardware—not enough disk space, or a bad printer.

If you can't find anything about a program, it is a bug in the documentation, and you should contact the author of that program and offer to write it yourself. If something is incorrect in existing

[1] A discussion group in Usenet, which talks about administring Unix computers.

documentation[2], it is a bug with that manual. If something appears incomplete or unclear in the manual, that is a bug.

If you can't beat `gnuchess` at chess, it is a flaw with your chess algorithm, but not necessarily a bug with your brain.

10.2.2 Reporting a bug

After you are sure you found a bug, it is important to make sure that your information gets to the right place. Try to find what program is causing the bug—if you can't find it, perhaps you could ask for help in `comp.os.linux.help` or `comp.unix.misc`. Once you find the program, try to read the manual page to see who wrote it.

The preferred method of sending bug reports in the Linux world is via electronic mail. If you don't have access to electronic mail, you might want to contact whoever you got Linux from—eventually, you're bound to encounter someone who either has electronic mail, or sells Linux commercially and therefore wants to remove as many bugs as possible. Remember, though, that no one is under any obligation to fix any bugs unless you have a contract!

When you send a bug report in, include all the information you can think of. This includes:

- A description of what you think is incorrect. For instance, "I get 5 when I compute 2+2" or "It says `segmentation violation -- core dumped`." It is important to say exactly what is happening so the maintainer can fix *your* bug!
- Include any relevant environment variables.
- The version of your kernel (see the file `/proc/version`) and your system libraries (see the directory `/lib`—if you can't decipher it, send a listing of `/lib`).
- How you ran the program in question, or, if it was a kernel bug, what you were doing at the time.
- **All** peripheral information. For instance, the command `w` may not be displaying the current process for certain users. Don't just say, "`w` doesn't work when for a certain user". The bug could occur because the user's name is eight characters long, or when he is logging in over the network. Instead say, "`w` doesn't display the current process for use `greenfie` when he logs in over the network."
- And remember, be polite. Most people work on free software for the fun of it, and because they have big hearts. Don't ruin it for them—the Linux community has already disillusioned too many developers, and it's still early in Linux's life!

[2]Especially this one!

Appendix A

The GNU General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES

Appendix B

The GNU Library General Public License

Bibliography

[1] Almesberger, Werner. *LILO: Generic Boot Loader for Linux.* Available electronically: `tsx-11.mit.edu`. July 3, 1993.

[2] Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc. 1986.

[3] Lamport, Leslie. LaTeX*: A Document Preparation System.* Reading, Massachusetts: Addison-Wesley Publishing Company. 1986.

[4] Stallman, Richard M. *GNU Emacs Manual*, eight edition. Cambridge, Massachusetts: Free Software Foundation. 1993.

Index