Chapter 3

Linux Tutorial

3.1 Introduction

New users of UNIX and Linux may be a bit intimidated by the size and apparent complexity of the system before them. There are many good books on using UNIX out there, for all levels of expertise from novice to expert. However, none of these books covers, specifically, an introduction to using Linux. While 95% of using Linux is exactly like using other UNIX systems, the most straightforward way to get going on your new system is with a tutorial tailored for Linux. Herein is such a tutorial.

This chapter does not go into a large amount of detail or cover many advanced topics. Instead, it is intended to get the new Linux user running, on both feet, so that he or she may then read a more general book about UNIX and understand the basic differences between other UNIX systems and Linux.

Very little is assumed here, except perhaps some familiarity with personal computer systems, and MS-DOS. However, even if you're not an MS-DOS user, you should be able to understand everything here. At first glance, UNIX looks a lot like MS-DOS (after all, parts of MS-DOS were modeled on the CP/M operating system, which in turn was modeled on UNIX). However, only the very superficial features of UNIX resemble MS-DOS in any way. Even if you're completely new to the PC world, this tutorial should be of help.

And, before we begin: *Don't be afraid to experiment.* The system won't bite you. You can't destroy anything by working on the system. UNIX has some amount of security built in, to prevent "normal" users (the role which you will now assume) from damaging files which are essential to the system. Even so, the absolute worst thing that can happen is that you'll delete all of your files—and you'll have to go back and re-install the system. So, at this point, you have nothing to lose.

3.2 Basic UNIX Concepts

UNIX is a multitasking, multiuser operating system. This means that there can be many people using one computer at the same time, running many different applications. (This differs from MS-

DOS, where only one person can use the system at any one time.) Under UNIX, for users to identify themselves to the system, they must **log in**, which entails two steps: Entering your **login name** (the name which the system identifies you as), and entering your **password**, which is your personal secret key to logging into your account. Because only you know your password, no one else can login to the system under your username.

On traditional UNIX systems, the system administrator will assign you a username and an initial password when you are given an account on the system. However, because you are the system administrator, you must set up your own account before you can login—see Section 3.2.1, below. For the following discussions, we'll use the imaginary username "`larry`".

In addition, each UNIX system has a **hostname** assigned to it. It is this hostname that gives your machine a name, gives it character and charm. The hostname is used to identify individual machines on a network, but even if your machine isn't networked, it should have a hostname. In Section 4.10.2 we'll cover setting your system's hostname. For our examples, below, the system's hostname is "`mousehouse`".

3.2.1 Creating an account

Before you can use the system, you must set up a user account for yourself. This is because it's usually not a good idea to use the `root` account for normal use. The `root` account should be reserved for running privileged commands and for maintaining the system, as discussed in Section 4.1.

In order to create an account for yourself, you need to login as `root` and use the `useradd` or `adduser` command. See Section 4.4 for information on this procedure.

3.2.2 Logging in

At login time, you'll see a prompt resembling the following on your screen:

```
mousehouse login:
```

Here, enter your username, and press the `Return` key. Our hero, `larry`, would type the following:

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

Now, enter your password. It won't be echoed to the screen when you login, so type carefully. If you mistype your password, you'll see the message

```
Login incorrect
```

and you'll have to try again.

Once you have correctly entered the username and password, you are officially logged into the system, and are free to roam.

3.2.3 Virtual consoles

The system's **console** is the monitor and keyboard connected directly to the system. (Because UNIX is a multiuser operating system, you may have other terminals connected to serial ports on your system, but these would not be the console.) Linux, like some other versions of UNIX, provides access to **virtual consoles** (or VC's), which allow you to have more than one login session from your console at a time.

To demonstrate this, login to your system (as demonstrated above). Now, press `alt-F2`. You should see the `login:` prompt again. You're looking at the second virtual console—you logged into the first. To switch back to the first VC, press `alt-F1`. *Voila!* You're back to your first login session.

A newly-installed Linux system probably allows you to access the first four VC's, using `alt-F1` through `alt-F4`. However, it is possible to enable up to 12 VC's—one for each function key on your keyboard. As you can see, use of VC's can be very powerful—you can be working on several different VC's at once.

While the use of VC's is somewhat limiting (after all, you can only be looking at one VC at a time), it should give you a feel for UNIX's multiuser capabilities. While you're working on VC #1, you can switch over to VC #2 and start working on something else.

3.2.4 Shells and commands

For most of your explorations in the world of UNIX, you'll be talking to the system through the use of a **shell**. A shell is just a program which takes user input (e.g., commands which you type) and translates them into instructions. This can be compared to the `COMMAND.COM` program under MS-DOS, which does essentially the same thing. The shell is just one interface to UNIX. There are many possible interfaces—such as the X Window System, which lets you run commands by using the mouse and keyboard in conjunction.

As soon as you login, the system starts the shell, and you can type commands to it. Here's a quick example. Here, Larry logs in, and is left sitting at the shell **prompt**.

```
mousehouse login: larry
Password: larry's password
Welcome to Mousehouse!

/home/larry#
```

"`/home/larry#`" is the shell's prompt, indicating that it's ready to take commands. (More on what the prompt itself means later.) Let's try telling the system to do something interesting:

```
/home/larry# make love
make: *** No way to make target `love'. Stop.
/home/larry#
```

Well, as it turns out `make` was the name of an actual program on the system, and the shell executed this program when given the command. (Unfortunately, the system was being unfriendly.)

This brings us to one burning question: What are commands? What happens when you type "`make love`"? The first word on the command line, "`make`", is the name of the command to be executed. Everything else on the command line is taken as arguments to this command. Examples:

```
/home/larry# cp foo bar
```

Here, the name of the command is "`cp`", and the arguments are "`foo`" and "`bar`".

When you type a command, the shell does several things. First of all, it looks at the command name, and checks to see if it is a command which is internal to the shell. (That is, a command which the shell knows how to execute itself. There are a number of these commands, and we'll go into them later.) The shell also checks to see if the command is an alias, or substitute name, for another command. If neither of these conditions apply, the shell looks for a program, on the disk, with the command's name. If it finds such a program, the shell runs it, giving the program the arguments specified on the command line.

In our example, the shell looks for the program called `make`, and runs it with the argument `love`. `Make` is a program often used to compile large programs, and it takes as arguments the name of a "target" to compile. In the case of "`make love`", we instructed `make` to compile the target `love`. Because `make` can't find a target by this name, it fails with a humorous error message, and we are returned to the shell prompt.

What happens if we type a command to a shell, and the shell can't find a program with the command name to run? Well, we can try it:

```
/home/larry# eat dirt
eat: command not found
/home/larry#
```

Quite simply, if the shell can't find a program with the name given on the command line (here, "`eat`"), it prints an error message which should be self-explanatory. You'll often see this error message if you mistype a command (for example, if you had typed "`mkae love`" instead of "`make love`").

3.2.5 Logging out

Before we delve much further, we should tell you how to log out of the system. At the shell prompt, use the command

```
/home/larry# exit
```

to logout. There are other ways of logging out as well, but this is the most foolproof one.

3.2.6 Changing your password

You should also be aware of how to change your password. The command `passwd` will prompt you for your old password, and your new password. It will ask you to reenter the new password for validation. Be careful not to forget your password—if you do, you will have to ask the system administrator to reset it for you. (If you're the system administrator, see Section 4.4.)

3.2.7 Files and directories

Under most operating systems (UNIX included), there is the concept of a **file**, which is just a bundle of information which is given a name (called a **filename**). Examples of files would be your history term paper, an e-mail message, or an actual program which can be executed. Essentially, anything which is saved on disk is saved in an individual file.

Files are identified by their filenames. For example, the file containing your history paper might be saved with the filename `history-paper`. These names usually identify the file and its contents in some form which is meaningful to you. There is no standard format for filenames as there is under MS-DOS and other operating systems; in general, filenames may contain any character (except `/`—see the discussion of pathnames, below), and are limited to 256 characters in length.

With the concept of files comes the concept of directories. A **directory** is just a collection of files. It can be thought of as a "folder" which contains many different files. Directories themselves are given names, with which you can identify them. Furthermore, directories are maintained in a tree-like structure; that is, directories may contain other directories.

A file may be referred to by its **pathname**, which is made up of the filename, preceded by the name of the directory which contains the file. For example, let's say that Larry has a directory called `papers`, which contains three files: `history-final`, `english-lit`, and `masters-thesis`. (Each of these three files contains information for three of Larry's ongoing projects.) To refer to the file `english-lit`, Larry can specify the file's pathname:

```
papers/english-lit
```

As you can see, the directory and file names are separated by a single slash (`/`). For this reason, filenames themselves cannot contain the `/` character. MS-DOS users will find this convention familiar, although in the MS-DOS world, the backslash (`\`) is used instead.

As mentioned, directories can be nested within each other as well. For example, let's say that Larry has another directory, within `papers`, called `notes`. This directory contains the files `math-notes` and `cheat-sheet`. The pathname of the file `cheat-sheet` would be

```
papers/notes/cheat-sheet
```

Therefore, the pathname really is a "path" which you take to locate a certain file. The directory above a given subdirectory is known as the **parent directory**. Here, the directory `papers` is the parent of the `notes` directory.

3.2.8 The directory tree

Most UNIX systems have a standard layout for files, so that system resources and programs can be easily located. This layout forms a directory tree, which starts at the "/" directory, also known as "the root directory". Directly underneath / are some important subdirectories: `/bin`, `/etc`, `/dev`, and `/usr`, among others. These directories in turn contain other directories which contain system configuration files, programs, and so on.

In particular, each user has a **home directory**, which is the directory set aside for that user to store his or her files. In the examples above, all of Larry's files (such as `cheat-sheet` and `history-final`) were contained in Larry's home directory. Usually, user home directories are contained under `/home`, and are named for the user who owns that directory. Therefore, Larry's home directory is `/home/larry`.

In Figure 3.2.8 a sample directory tree is represented. It should give you some idea of how the directory tree on your system is organized.

3.2.9 The current working directory

At any given time, commands that you type to the shell are given in terms of your **current working directory**. You can think of your working directory as the directory in which you are currently "located". When you first login, your working directory is set to your home directory—`/home/larry` in our case. Whenever you reference a file, you may refer to it in relationship to your current working directory, instead of specifying the full pathname of the file.

Here's an example. Larry has the directory `papers`, and `papers` contains the file `history-final`. If Larry wants to look at this file, he can use the command

`/home/larry#` *more /home/larry/papers/history-final*

The `more` command simply displays a file, one screen at a time. However, because Larry's current working directory is `/home/larry`, he can instead refer to the file *relative* to his current location. The command would be

`/home/larry#` *more papers/history-final*

Therefore, if you begin a filename (such as `papers/final`) with a character other than "/", the system assumes that you're referring to the file in terms relative to your current working directory. This is known as a **relative pathname**.

On the other hand, if you begin a filename with a "/", the system interprets this as a full pathname—that is, a pathname including the entire path to the file, starting from the root directory, /. This is known as an **absolute pathname**.

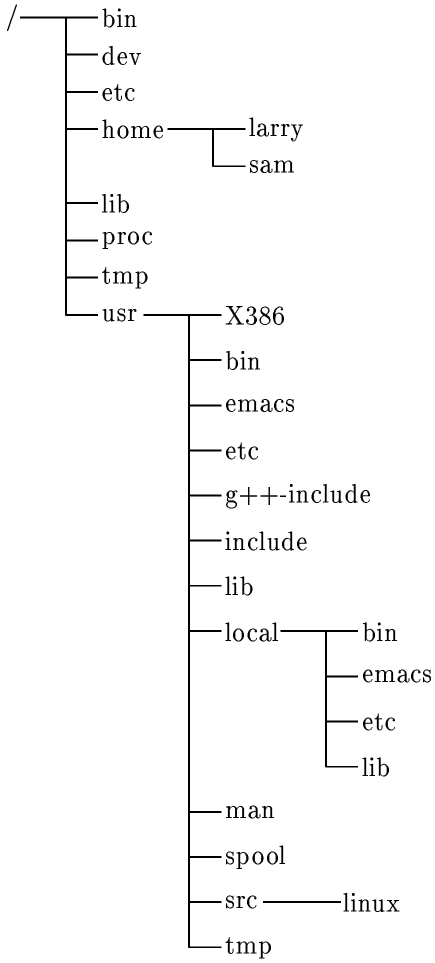

Figure 3.1: A typical (abridged) Unix directory tree.

3.2.10 Referring to home directories

Under both `tcsh` and `bash`,[1] your home directory can be referred to using the tilde character ("~"). For example, the command

> `/home/larry#` *more ~/papers/history-final*

is equivalent to

> `/home/larry#` *more /home/larry/papers/history-final*

The "~" character is simply replaced with the name of your home directory by the shell.

[1] `tcsh` and `bash` are two *shells* running under Linux. The shell is the program which reads user commands and executes them; most Linux systems enable either `tcsh` or `bash` for new user accounts.

In addition, you can specify other user's home directories with the tilde as well. The pathname "`~karl/letters`" translates to "`/home/karl/letters`" by the shell (if `/home/karl` is karl's home directory). The use of the tilde is simply a shortcut; there is no directory named "`~`"—it's just syntactic sugar provided by the shell.

3.3 First Steps into UNIX

Before we begin, it is important to note that all file and command names on a UNIX system are case-sensitive (unlike operating systems such as MS-DOS). For example, the command `make` is very different than `Make` or `MAKE`. The same hold for file and directory names.

3.3.1 Moving around

Now that we can login, and know how to refer to files using pathnames, how can we change our current working directory, to make life easier?

The command for moving around in the directory structure is `cd`, short for "change directory". You'll notice that many often-used Unix commands are two or three letters. The usage of the `cd` command is:

`cd` ⟨*directory*⟩

where ⟨*directory*⟩ is the name of the directory which you wish to change to.

As we said, when you login, you begin in your home directory. If Larry wanted to move down into the `papers` subdirectory, he'd use the command

```
/home/larry# cd papers
/home/larry/papers#
```

As you can see, Larry's prompt changes to reflect his current working directory (so he knows where he is). Now that he's in the `papers` directory, he can look at his history final with the command

```
/home/larry/papers# more history-final
```

Now, Larry is stuck in the `papers` subdirectory. To move back up to the parent directory, use the command

```
/home/larry/papers# cd ..
/home/larry#
```

(Note the space between the "`cd`" and the "`..`".) Every directory has an entry named "`..`" which refers to the parent directory. Similarly, every directory has an entry named "`.`" which refers to itself. Therefore, the command

```
/home/larry/papers# cd .
```

gets us nowhere.

You can also use absolute pathnames in the `cd` command. To `cd` into Karl's home directory, we can use the command

```
/home/larry/papers# cd /home/karl
/home/karl#
```

Also, using `cd` with no argument will return you to your own home directory.

```
/home/karl# cd
/home/larry#
```

3.3.2 Looking at the contents of directories

Now that you know how to move around directories you probably think, "So what?" The basic skill of moving around directories is fairly useless, so let's introduce a new command, `ls`. `ls` prints a listing of files and directories, by default from your current directory. For example:

```
/home/larry# ls
Mail
letters
papers
/home/larry#
```

Here we can see that Larry has three entries in his current directory: `Mail`, `letters`, and `papers`. This doesn't tell us much—are these directories or files? We can use the `-F` option on the `ls` command to tell us more.

```
/home/larry# ls -F
Mail/
letters/
papers/
/home/larry#
```

From the `/` appended to each filename, we know that these three entries are in fact subdirectories.

Using `ls -F` may also append "`*`" to the end of a filename. This indicates that the file is an **executable**, or a program which can be run. If nothing is appended to the filename using `ls -F`, the file is a "plain old file", that is, it's neither a directory, or an executable.

In general, each UNIX command may take a number of options in addition to other arguments. These options usually begin with a "`-`", as demonstrated above with `ls -F`. The `-F` option tells `ls` to give more information about the type of the files involved—in this case, printing a / after each directory name.

If you give `ls` a directory name, it will print the contents of that directory.

```
/home/larry# ls -F papers
english-lit
history-final
masters-thesis
notes/
/home/larry#
```

Or, for a more interesting listing, let's see what's in the system's `/etc` directory.

```
/home/larry# ls /etc

Images      ftpusers    lpc         rc.new      shells
adm         getty       magic       rc0.d       startcons
bcheckrc    gettydefs   motd        rc1.d       swapoff
brc         group       mount       rc2.d       swapon
brc~        inet        mtab        rc3.d       syslog.conf
csh.cshrc   init        mtools      rc4.d       syslog.pid
csh.login   init.d      pac         rc5.d       syslogd.reload
default     initrunlvl  passwd      rmt         termcap
disktab     inittab     printcap    rpc         umount
fdprm       inittab.old profile     rpcinfo     update
fstab       issue       psdatabase  securetty   utmp
ftpaccess   lilo        rc          services    wtmp
/home/larry#
```

(For those MS-DOS users out there, notice how the filenames can be longer than 8 characters, and can contain periods in any position. It is even possible to have more than one period in a filename.)

Let's `cd` up to the top of the directory tree, using "`cd ..`", and then down to another directory: `/usr/bin`.

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

You can also move into directories in multiple steps, as in `cd /usr/bin`.

Try moving around various directories, using `ls` and `cd`. In some cases, you may run into a foreboding "`Permission denied`" error message. This is simply the concept of UNIX security kicking in: in order to `ls` or to `cd` into a directory, you must have permission to do so. We'll talk more about this in Section 3.9.

3.3.3 Creating new directories

It's time to learn how to create directories. This involves the use of the `mkdir` command. Try the following:

```
/home/larry# mkdir foo
/home/larry# ls -F
Mail/
foo/
letters/
papers/
/home/larry# cd foo
/home/larry/foo# ls
/home/larry/foo#
```

Congrats! You've just made a new directory and moved into it. Since there aren't any files in this new directory, let's learn how to copy files from one place to another.

3.3.4 Copying files

Copying files is done with the command `cp`:

```
/home/larry/foo# cp /etc/termcap .
/home/larry/foo# cp /etc/shells .
/home/larry/foo# ls -F
shells     termcap
/home/larry/foo# cp shells bells
/home/larry/foo# ls -F
bells     shells     termcap
/home/larry/foo#
```

The `cp` command copies the files listed on the command line to the file or directory given as the last argument. Notice how we use the directory "`.`" to refer to the current directory.

3.3.5 Moving files

A new command named `mv` moves files, instead of copying them. The syntax is very straightforward.

```
/home/larry/foo# mv termcap sells
/home/larry/foo# ls -F
bells     sells     shells
/home/larry/foo#
```

Notice how `termcap` no longer exists, but in its place is the file `sells`. This can be used to rename files, as we have just done, but also to move a file to a completely new directory.

◇ **Note:** `mv` and `cp` will overwrite the destination file (if it already exists) without asking you. Be careful when you move a file into another directory: there may already be a file with the same name in that directory, which you'll overwrite!

3.3.6 Deleting files and directories

You now have an ugly rhyme developing with the use of the `ls` command. To delete a file, use the `rm` command. ("`rm`" stands for "remove").

```
/home/larry/foo# rm bells sells
/home/larry/foo# ls -F
shells
/home/larry/foo#
```

We're left with nothing but shells, but we won't complain. Note that `rm` by default won't prompt you before deleting a file—so be careful.

A related command to `rm` is `rmdir`. This command deletes a directory, but only if the directory is empty. If the directory contains any files or subdirectories, `rmdir` will complain.

3.3.7 Looking at files

The commands `more` and `cat` are used for viewing the contents of files. `more` displays a file, one screenful at a time, while `cat` displays the whole file at once.

To look at the file `shells`, we can use the command

```
/home/larry/foo# more shells
```

In case you're interested what `shells` contains, it's a list of valid shell programs on your system. On most systems, this includes `/bin/sh`, `/bin/bash`, and `/bin/csh`. We'll talk about these different types of shells later.

While using `more`, press `Space` to display the next page of text, and `b` to display the previous page. There are other commands available in `more` as well, these are just the basics. Pressing `q` will quit `more`.

Quit `more` and try `cat /etc/termcap`. The text will probably fly by much too quickly for you to read it. The name "`cat`" actually stands for "concatenate", which is the real use of the program. The `cat` command can be used to concatenate the contents of several files and save the result to another file. This will be discussed later.

3.3.8 Getting online help

Almost every UNIX system, Linux included, provides a facility known as "manual pages", or "man pages" for short. These man pages contain online documentation for all of the various system commands, resources, configuration files, and so on.

The command used to access man pages is `man`. For example, if you're interested in finding out about the other options of the `ls` command, you can type

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

and the man page for `ls` will be displayed.

Unfortunately, most of the man pages out there are written for those who already have some idea of what the command or resource does. For this reason, man pages usually only contain the hardcore technical details of the command, without a lot of tutorial. However, man pages can be an invaluable resource for jogging your memory if you forget the syntax of a command. Man pages will also tell you a lot about the commands which we won't tell you in this book.

I suggest that you try `man` for the commands we've already gone over, and whenever I introduce a new command. You'll notice some of these commands won't have man pages. This could be for several reasons. For one, the man pages haven't been written yet (the Linux Documentation Project is responsible for man pages under Linux as well. We are gradually accumulating most of the man pages available for the system). Secondly, the the command might be an internal shell command, or an alias (as discussed in Section 3.2.4), in which case it would not have a man page of its own. One example is `cd`, which is a shell internal command. The shell actually processes the `cd`—there is no separate program which contains this command.

3.4 Summary of Basic Commands

This section introduces some of the most useful basic commands on a UNIX system, including those covered in the last section.

Note that options usually begin with a "`-`", and in most cases multiple one-letter options may be combined using a single "`-`". For example, instead of using the command `ls -l -F`, it is adequate to use `ls -lF`.

Instead of listing all of the options available for each of these commands, we'll only talk about those which are useful or important at this time. In fact, most of these commands have a large number of options (most of which you'll never use). You can use `man` to see the manual pages for each command, which list all of the available options.

Also note that many of these commands take a list of files or directories as arguments, denoted by "⟨*file1*⟩ ... ⟨*fileN*⟩". For example, the `cp` command takes as arguments a list of files to copy, followed by the destination file or directory. When copying more than one file, the destination must be a directory.

`cd` Change the current working directory.
Syntax: `cd` ⟨*directory*⟩
⟨*directory*⟩ is the directory to change to. ("`.`" refers to the current directory, "`..`" the parent directory.)
Example: `cd ../foo` sets the current directory to `../foo`.

`ls` Displays information about the named files and directories.
Syntax: `ls` ⟨*file1*⟩ ⟨*file2*⟩ ... ⟨*fileN*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the filenames or directories to list. Options: There are more options than you want to think about. The most commonly used are `-F` (used to display some information about the type of the file), and `-l` (gives a "long" listing including file size, owner, permissions, and so on. This will be covered in detail later.)
Example: `ls -lF /home/larry` will display the contents of the directory `/home/larry`.

`cp` Copies file(s) to another file or directory.
Syntax: `cp` ⟨*file1*⟩ ⟨*file2*⟩ ... ⟨*fileN*⟩ ⟨*destination*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the files to copy, and ⟨*destination*⟩ is the destination file or directory.
Example: `cp ../frog joe` copies the file `../frog` to the file or directory `joe`.

`mv` Moves file(s) to another file or directory. This command does the equivalent of a copy followed by the deletion of the original. This can be used to rename files, as in the MS-DOS command `RENAME`.
Syntax: `mv` ⟨*file1*⟩ ⟨*file2*⟩ ... ⟨*fileN*⟩ ⟨*destination*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the files to move, and ⟨*destination*⟩ is the destination file or directory.
Example: `mv ../frog joe` moves the file `../frog` to the file or directory `joe`.

`rm` Deletes files. Note that when files are deleted under UNIX, they are unrecoverable (unlike MS-DOS, where you can usually "undelete" the file).
Syntax: `rm` ⟨*file1*⟩ ⟨*file2*⟩ ... ⟨*fileN*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the filenames to delete.
Options: `-i` will prompt for confirmation before deleting the file.
Example: `rm -i /home/larry/joe /home/larry/frog` deletes the files `joe` and `frog` in `/home/larry`.

`mkdir` Creates new directories.
Syntax: `mkdir` ⟨*dir1*⟩ ⟨*dir2*⟩ ... ⟨*dirN*⟩
Where ⟨*dir1*⟩ through ⟨*dirN*⟩ are the directories to create.
Example: `mkdir /home/larry/test` creates the directory `test` under `/home/larry`.

`rmdir` This command deletes empty directories. When using `rmdir`, your current working directory must not be within the directory to be deleted.

Syntax: `rmdir` ⟨*dir1*⟩ ⟨*dir2*⟩ ...⟨*dirN*⟩
Where ⟨*dir1*⟩ through ⟨*dirN*⟩ are the directories to delete.
Example: `rmdir /home/larry/papers` deletes the directory `/home/larry/papers`, if it is empty.

`man` Displays the manual page for the given command or resource (that is, any system utility which isn't a command, such as a library function.) Syntax: `man` ⟨*command*⟩
Where ⟨*command*⟩ is the name of the command or resource to get help on.
Example: `man ls` gives help on the `ls` command.

`more` Displays the contents of the named files, one screenful at a time.
Syntax: `more` ⟨*file1*⟩ ⟨*file2*⟩ ...⟨*fileN*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the files to display.
Example: `more papers/history-final` displays the file `papers/history-final`.

`cat` Officially used to concatenate files, `cat` is also used to display the entire contents of a file at once.
Syntax: `cat` ⟨*file1*⟩ ⟨*file2*⟩ ...⟨*fileN*⟩
Where ⟨*file1*⟩ through ⟨*fileN*⟩ are the files to display.
Example: `cat letters/from-mdw` displays the file `letters/from-mdw`.

`echo` Simply echoes the given arguments.
Syntax: `echo` ⟨*arg1*⟩ ⟨*arg2*⟩ ...⟨*argN*⟩
Where ⟨*arg1*⟩ through ⟨*argN*⟩ are the arguments to echo.
Example: `echo "Hello world"` displays the string "`Hello world`".

`grep` Display all of the lines in the named file(s) matching the given pattern.
Syntax: `grep` ⟨*pattern*⟩ ⟨*file1*⟩ ⟨*file2*⟩ ...⟨*fileN*⟩
Where ⟨*pattern*⟩ is a regular expression pattern, and ⟨*file1*⟩ through ⟨*fileN*⟩ are the files to search.
Example: `grep loomer /etc/hosts` will display all lines in the file `/etc/hosts` which contain the pattern "`loomer`".

3.5 Exploring the File System

The **file system** is the collection of files and the hierarchy of directories on your system. I promised before to escort you around the filesystem and the time has come.

You have the skills and the knowledge to make sense out of what I'm saying, and you have a roadmap. (Refer to Figure 3.2.8 on page 80).

First, change to the root directory (`cd /`), and do an `ls -F`. You'll probably see these directories[2]: `bin`, `dev`, `etc`, `home`, `install`, `lib`, `mnt`, `proc`, `root`, `tmp`, `user`, `usr`, and `var`.

[2] You may see others, and you might not see all of them. Don't worry. Every release of Linux differs in some respects.

Let's take a look at each of these directories.

`/bin` — `/bin` is short for "binaries", or executables. This is where many essential system programs reside. Use the command "`ls -F /bin`" to list the files here. If you look down the list you may see a few commands that you recognize, such as `cp`, `ls`, and `mv`. These are the actual programs for these commands. When you use the `cp` command, you're running the program `/bin/cp`.

Using `ls -F`, you'll see that most (if not all) of the files in `/bin` have an asterisk ("`*`") appended to their filenames. This indicates that the files are executables, as described in Section 3.3.2.

`/dev` — Next on our stop is `/dev`. Take a look, again with `ls -F`.

The "files" in `/dev` are known as **device drivers**—they are used to access system devices and resources, such as disk drives, modems, memory, and so on. For example, just as you can read data from a file, you can read input from the mouse by accessing `/dev/mouse`.

The filenames beginning with `fd` are floppy disk devices. `fd0` is the first floppy disk drive, `fd1` the second. Now, the astute among you will notice that there are more floppy disk devices then just the two I've listed above: they represent specific types of floppy disks. For example, `fd1H1440` will access high-density, 3.5" diskettes in drive 1.

Here is a list of some of the most commonly used device files. Note that even though you may not have some of the devices listed below, the chances are that you'll have entries in `/dev` for them anyway.

- `/dev/console` refers to the system's console—that is, the monitor connected directly to your system.
- The various `/dev/ttyS` and `/dev/cua` devices are used for accessing serial ports. For example, `/dev/ttyS0` refers to "`COM1`" under MS-DOS. The `/dev/cua` devices are "callout" devices, which are used in conjunction with a modem.
- The device names beginning with `hd` access hard drives. `/dev/hda` refers to the *whole* first hard disk, while `hda1` refers to the first *partition* on `/dev/hda`.
- The device names beginning with `sd` are SCSI drives. If you have a SCSI hard drive, instead of accessing it through `/dev/hda`, you would access `/dev/sda`. SCSI tapes are accessed via `st` devices, and SCSI CD-ROM via `sr` devices.
- The device names beginning with `lp` access parallel ports. `/dev/lp0` refers to "`LPT1`" in the MS-DOS world.
- `/dev/null` is used as a "black hole"—any data sent to this device is gone forever. Why is this useful? Well, if you wanted to suppress the output of a command appearing on your screen, you could send that output to `/dev/null`. We'll talk more about this later.

- The device names beginning with `/dev/tty` refer to the "virtual consoles" on your system (accessed via by pressing `alt-F1`, `alt-F2`, and so on). `/dev/tty1` refers to the first VC, `/dev/tty2` refers to the second, and so on.
- The device names beginning with `/dev/pty` are "pseudo-terminals". They are used to provide a "terminal" to remote login sessions. For example, if your machine is on a network, incoming `telnet` logins would use one of the `/dev/pty` devices.

`/etc` — `/etc` contains a number of miscellaneous system configuration files. These include `/etc/passwd` (the user database), `/etc/rc` (the system initialization script), and so on.

`/sbin` — `sbin` is used for storing essential system binaries, to be used by the system administrator.

`/home` — `/home` contains user's home directories. For example, `/home/larry` is the home directory for the user "`larry`". On a newly-installed system, there may not be any users in this directory.

`/lib` — `/lib` contains **shared library images**. These files contain code which many programs share in common. Instead of each program containing its own copy of these shared routines, they are all stored in one common place, in `/lib`. This makes executable files smaller, and saves space on your system.

`/proc` — `/proc` is a "virtual filesystem", the files in which are stored in memory, not on the drive. They refer to the various **processes** running on the system, and allow you to get information about what programs and processes are running at any given time. We'll go into more detail in Section 3.11.1.

`/tmp` — Many programs have a need to generate some information and store it in a temporary file. The canonical location for these files is in `/tmp`.

`/usr` — `/usr` is a very important directory. It contains a number of subdirectories which in turn contain some of the most important and useful programs and configuration files used on the system.

The various directories described above are essential for the system to operate, but most of the things found in `/usr` are optional for the system. However, it is those optional things which make the system useful and interesting. Without `/usr`, you'd more or less have a boring system, only with programs like `cp` and `ls`. `/usr` contains most of the larger software packages and the configuration files which accompany them.

`/usr/X386` — `/usr/X386` contains The X Window System, if you installed it. The X Window System is a large, powerful graphical environment which provides a large number of graphical utilities and programs, displayed in "windows" on your screen. If you're at all familiar with the Microsoft Windows or Macintosh environments, X Windows will look very familiar. The `/usr/X386` directory contains all of the X Windows executables, configuration files, and support files. This will be covered in more detail in Section 5.1.

`/usr/bin` — `/usr/bin` is the real warehouse for software on any UNIX system. It contains most of the executables for programs not found in other places, such as `/bin`.

`/usr/etc` — Just as `/etc` contained miscellaneous system programs and configuration files, `/usr/etc` contains even more of these utilities and files. In general, the files found in `/usr/etc` are not essential to the system, unlike those found in `/etc`, which are.

`/usr/include` — `/usr/include` contains **include files** for the C compiler. These files (most of which end in `.h`, for "header") declare data structure names, subroutines, and constants used when writing programs in C. Those files found in `/usr/include/sys` are generally used when programming on the UNIX system level. If you are familiar with the C programming language, here you'll find header files such as `stdio.h`, which declares functions such as `printf()`.

`/usr/g++-include`

`/usr/g++-include` contains include files for the C++ compiler (much like `/usr/include`).

`/usr/lib` — `/usr/lib` contains the "stub" and "static" library equivalents to the files found in `/lib`. When compiling a program, the program is "linked" with the libraries found in `/usr/lib`, which then directs the program to look in `/lib` when it needs the actual code in the library. In addition, various other programs store configuration files in `/usr/lib`.

`/usr/local` — `/usr/local` is a lot like `/usr`—it contains various programs and files not essential to the system, but which make the system fun and exciting. In general, those programs found in `/usr/local` are specialized for your system specifically—that is, `/usr/local` differs greatly between UNIX systems.

Here, you'll find large software packages such as TeX (a document formatting system) and Emacs (a large and powerful editor), if you installed them.

`/usr/man` — This directory contains the actual man pages. There are two subdirectories for every man page "section" (use the command `man man` for details). For example, `/usr/man/man1` contains the source (that is, the unformatted original) for man pages in section 1, and `/usr/man/cat1` contains the formatted man pages for section 1.

`/usr/src` — `/usr/src` contains the source code (the uncompiled program) for various programs on your system. The most important thing here is `/usr/src/linux`, which contains the source code for the Linux kernel.

`/var` — `/var` holds directories that often change in size or tend to grow. Many of those directories used to reside in `/usr`, but since we are trying to keep it relatively unchangeable, the directories that change often have been moved to `/var`. Some of those directories are:

`/var/adm` — `/var/adm` contains various files of interest to the system administrator, specifically system logs, which record any errors or problems with the system. Other files record logins to the system, as well as failed login attempts. This will be covered in Chapter 4.

`/var/spool` — `/var/spool` contains files which are to be “spooled” to another program. For example, if your machine is connected to a network, incoming mail will be stored in `/var/spool/mail`, until you read it or delete it. Outgoing or incoming news articles may be found in `/var/spool/news`, and so on.

3.6 Types of shells

As I have mentioned too many times before, UNIX is a multitasking, multiuser operating system. Multitasking is *very* useful, and once you get used to it, you'll use it all of the time. Before long, you'll be able to run programs in the “background”, switch between multiple tasks, and “pipeline” programs together to achieve complicated results with a single command.

Many of the features we'll be covering in this section are features provided by the shell itself. Be careful not to confuse UNIX (the actual operating system) with the shell—the shell is just an interface to the underlying system. The shell provides a great deal of functionality on top of UNIX itself.

The shell is not only an interpreter for your interactive commands, which you type at the prompt. It is also a powerful programming language, which allows you to write **shell scripts**, to “batch” several shell commands together in a file. MS-DOS users will recognize the similarity to “batch files”. Use of shell scripts is a very powerful tool, which will allow you to automate and expand your usage of UNIX. See Section 3.13.1 for more information.

There are several types of shells in the UNIX world. The two major types are the “Bourne shell” and the “C shell”. The Bourne shell uses a command syntax like the original shell on early UNIX systems, such as System III. The name of the Bourne shell on most UNIX systems is `/bin/sh` (where `sh` stands for “shell”). The C shell (not to be confused with sea shell) uses a different syntax, somewhat like the programming language C, and on most UNIX systems is named `/bin/csh`.

Under Linux, there are several variations of these shells available. The two most commonly used are the Bourne Again Shell, or “Bash” (`/bin/bash`), and Tcsh (`/bin/tcsh`). Bash is a form of the Bourne shell with many of the advanced features found in the C shell. Because Bash supports a superset of the Bourne shell syntax, any shell scripts written in the standard Bourne shell should work with Bash. For those who prefer to use the C shell syntax, Linux supports Tcsh, which is an expanded version of the original C shell.

The type of shell that you decide to use is mostly a religious issue. Some folks prefer the Bourne shell syntax with the advanced features of Bash, and some prefer the more structured C shell syntax. As far as normal commands, such as `cp` and `ls`, are concerned, the type of shell you're using doesn't matter—the syntax is the same. Only when you start to write shell scripts or use some of the advanced features of the shell do the differences between shell types begin to matter.

As we're discussing some of the features of the shell, below, we'll note those differences between Bourne and C shells. However, for the purposes of this manual, most of those differences are minimal. (If you're really curious at this point, read the man pages for `bash` and `tcsh`).

3.7 Wildcards

A key feature of most Unix shells is the ability to reference more than one filename using special characters. These so-called **wildcards** allow you to refer to, say, all filenames which contain the character "`n`".

The wildcard "`*`" refers to any character or string of characters in a filename. For example, when you use the character "`*`" in a filename, the shell replaces it with all possible substitutions from filenames in the directory which you're referencing.

Here's a quick example. Let's suppose that Larry has the files `frog`, `joe`, and `stuff` in his current directory.

```
/home/larry# ls
frog     joe     stuff
/home/larry#
```

To access all files with the letter "o" in the filename, we can use the command

```
/home/larry# ls *o*
frog     joe
/home/larry#
```

As you can see, the use of the "`*`" wildcard was replaced with all substitutions which matched the wildcard from filenames in the current directory.

The use of "`*`" by itself simply matches all filenames, because all characters match the wildcard.

```
/home/larry# ls *
frog     joe     stuff
/home/larry#
```

Here are a few more examples.

```
/home/larry# ls f*
frog
/home/larry# ls *ff
stuff
/home/larry# ls *f*
```

```
frog    stuff
/home/larry# ls s*f
stuff
/home/larry#
```

The process of changing a "*" into filenames is called **wildcard expansion** and is done by the shell. This is important: the individual commands, such as `ls`, *never* see the "*" in their list of parameters. The shell expands the wildcard to include all of the filenames which match. So, the command

```
/home/larry# ls *o*
```

is expanded by the shell to actually be

```
/home/larry# ls frog joe
```

One important note about the "*" wildcard. Using this wildcard will *not* match filenames which begin with a single period ("`.`"). These files are treated as "hidden" files—while they are not really hidden, they don't show up on normal `ls` listings, and aren't touched by the use of the "*" wildcard.

Here's an example. We already mentioned that each directory has two special entries in it: "`.`" refers to the current directory, and "`..`" refers to the parent directory. However, when you use `ls`, these two entries don't show up.

```
/home/larry# ls
frog    joe    stuff
/home/larry#
```

If you use the `-a` switch with `ls`, however, you can display filenames which begin with "`.`". Observe:

```
/home/larry# ls -a
.    ..    .bash_profile    .bashrc    frog    joe    stuff
/home/larry#
```

Now we can see the two special entries, "`.`" and "`..`", as well as two other "hidden" files—`.bash_profile` and `.bashrc`. These two files are startup files used by `bash` when larry logs in. More on them in Section 3.13.3.

Note that when we use the "*" wildcard, none of the filenames beginning with "`.`" are displayed.

```
/home/larry# ls *
frog    joe    stuff
/home/larry#
```

This is a safety feature: if the "*" wildcard matched filenames beginning with "`.`", it would also match the directory names "`.`" and "`..`". This can be dangerous when using certain commands.

Another wildcard is "`?`". The "`?`" wildcard will only expand a single character. Thus, "`ls ?`" will display all one character filenames, and "`ls termca?`" would display "`termcap`" but *not* "`termcap.backup`". Here's another example:

```
/home/larry# ls j?e
joe
/home/larry# ls f??g
frog
/home/larry# ls ????f
stuff
/home/larry#
```

As you can see, wildcards allow you to specify many files at one time. In the simple command summary, in Section 3.4, we said that the `cp` and `mv` commands actually can copy or move multiple files at one time. For example,

```
/home/larry# cp /etc/s* /home/larry
```

will copy all filenames in `/etc` beginning with "`s`" to the directory `/home/larry`. Therefore, the format of the `cp` command is really

`cp` ⟨*file1*⟩ ⟨*file2*⟩ ⟨*file3*⟩ ...⟨*fileN*⟩ ⟨*destination*⟩

where ⟨*file1*⟩ through ⟨*fileN*⟩ is a list of filenames to copy, and ⟨*destination*⟩ is the destination file or directory to copy them to. `mv` has an identical syntax.

Note that if you are copying or moving more than one file, the ⟨*destination*⟩ must be a directory. You can only copy or move a *single* file to another file.

3.8 UNIX Plumbing

3.8.1 Standard input and output

Many UNIX commands get input from what is known as **standard input** and send their output to **standard output** (often abbreviated as "stdin" and "stdout"). Your shell sets things up so that standard input is your keyboard, and standard output is the screen.

Here's an example using the command `cat`. Normally, `cat` reads data from all of the filenames given on the command line and sends this data directly to stdout. Therefore, using the command

```
/home/larry/papers# cat history-final masters-thesis
```

will display the contents of the file `history-final` followed by `masters-thesis`.

However, if no filenames are given to `cat` as parameters, it instead reads data from stdin, and sends it back to stdout. Here's an example.

```
/home/larry/papers# cat
Hello there.
Hello there.
Bye.
Bye.
ctrl-D
/home/larry/papers#
```

As you can see, each line that the user types (displayed in italics) is immediately echoed back by the `cat` command. When reading from standard input, commands know that the input is "finished" when they receive an EOT (end-of-text) signal. In general, this is generated by pressing `ctrl-D`.

Here's another example. The command `sort` reads in lines of text (again, from stdin, unless files are given on the command line), and sends the sorted output to stdout. Try the following.

```
/home/larry/papers# sort
bananas
carrots
apples
ctrl-D
apples
bananas
carrots
/home/larry/papers#
```

Now we can alphabetize our shopping list... isn't UNIX useful?

3.8.2 Redirecting input and output

Now, let's say that we wanted to send the output of `sort` to a file, to save our shopping list elsewhere. The shell allows us to **redirect** standard output to a filename, using the ">" symbol. Here's how it works.

```
/home/larry/papers# sort > shopping-list
bananas
carrots
apples
ctrl-D
/home/larry/papers#
```

As you can see, the result of the `sort` command isn't displayed, instead it's saved to the file `shopping-list`. Let's look at this file.

```
/home/larry/papers# cat shopping-list
apples
```

```
bananas
carrots
/home/larry/papers#
```

Now we can sort our shopping list, and save it, too! But let's suppose that we were storing our unsorted, original shopping list in the file `items`. One way of sorting the information and saving it to a file would be to give `sort` the name of the file to read, in lieu of standard input, and redirect standard output as we did above. As so:

```
/home/larry/papers# sort items > shopping-list
/home/larry/papers# cat shopping-list
apples
bananas
carrots
/home/larry/papers#
```

However, there's another way of doing this. Not only can we redirect standard output, but we can redirect standard *input* as well, using the "<" symbol.

```
/home/larry/papers# sort < items
apples
bananas
carrots
/home/larry/papers#
```

Technically, `sort < items` is equivalent to `sort items`, but the former allows us to demonstrate the point: `sort < items` behaves as if the data in the file `items` was typed to standard input. The shell handles the redirection. `sort` wasn't given the name of the file (`items`) to read; as far as `sort` is concerned, it was still reading from standard input as if you had typed the data from your keyboard.

This introduces the concept of a **filter**. A filter is a program which reads data from standard input, processes it in some way, and sends the processed data to standard output. Using redirection, standard input and/or standard output can be referenced from files. `sort` is a simple filter: it sorts the incoming data and sends the result to standard output. `cat` is even simpler: it doesn't do anything with the incoming data, it simply outputs whatever was given to it.

3.8.3 Using pipes

We've already demonstrated how to use `sort` as a filter. However, these examples assumed that you had data in a file somewhere, or were willing to type the data to standard input yourself. What if the data you wanted to sort came from the output of another command, such as `ls`? For example, using the `-r` option with `sort` sorts the data in reverse-alphabetical order. If you wanted to list the files in your current directory in reverse order, one way to do it would be:

```
/home/larry/papers# ls
english-list
```

```
history-final
masters-thesis
notes
/home/larry/papers# ls > file-list
/home/larry/papers# sort -r file-list
notes
masters-thesis
history-final
english-list
/home/larry/papers#
```

Here, we saved the output of `ls` in a file, and then ran `sort -r` on that file. But this is unwieldy and causes us to use a temporary file to save the data from `ls`.

The solution is to use **pipelining**. Pipelining is another feature of the shell which allows you to connect a string of commands in a "pipe", where the stdout of the first command is sent directly to the stdin of the second command, and so on. Here, we wish to send the stdout of `ls` to the stdin of `sort`. The "`|`" symbol is used to create a pipe:

```
/home/larry/papers# ls | sort -r
notes
masters-thesis
history-final
english-list
/home/larry/papers#
```

This command is much shorter, and obviously easier to type.

Another useful example—using the command

```
/home/larry/papers# ls /usr/bin
```

is going to display a long list a files, most of which will fly past the screen too quickly for you to read them. Instead, let's use `more` to display the list of files in `/usr/bin`.

```
/home/larry/papers# ls /usr/bin | more
```

Now you can page down the list of files at your own leisure.

But the fun doesn't stop here! We can pipe more than two commands together. The command `head` is a filter which displays the first lines from an input stream (here, input from a pipe). If we wanted to display the last filename in alphabetical order in the current directory, we can use:

```
/home/larry/papers# ls | sort -r | head -1
notes
/home/larry/papers#
```

where `head -1` simply displays the first line of input that it receives (in this case, the stream of reverse-sorted data from `ls`).

3.8.4 Non-destructive redirection

Using ">" to redirect output to a file is destructive: in other words, the command

`/home/larry/papers#` *ls > file-list*

overwrites the contents of the file `file-list`. If, instead, you redirect with the symbol ">>", the output will be appended to the named file, instead of overwriting it.

`/home/larry/papers#` *ls >> file-list*

will append the output of the `ls` command to `file-list`.

Just keep in mind that redirection and using pipes are features provided by the shell—the shell provides this handy syntax using ">" and ">>" and "|". It has nothing to do with the commands themselves, but the shell.

3.9 File Permissions

3.9.1 Concepts of file permissions

Because there are multiple users on a UNIX system, in order to protect individual user's files from tampering by other users, UNIX provides a mechanism known as **file permissions**. This mechanism allows files and directories to be "owned" by a particular user. As an example, because Larry created the files in his home directory, Larry owns those files, and has access to them.

UNIX also allows files to be shared between users and groups of users. If Larry so desired, he could cut off access to his files, such that no other user could access them. However, on most systems the default is to allow other users to read your files, but not modify or delete them in any way.

As explained above, every file is owned by a particular user. However, files are also owned by a particular **group**, which is a system-defined group of users. Every user is placed into at least one group when that user is created. However, the system administrator may also grant the user access to more than one group.

Groups are usually defined by the type of users which access the machine. For example, on a university UNIX system, users may be placed into the groups `student`, `staff`, `faculty` or `guest`. There are also a few system-defined groups (such as `bin` and `admin`) which are used by the system itself to control access to resources—very rarely do actual users belong to these system groups.

Permissions fall into three main divisions: read, write, and execute. These permissions may be granted to three classes of users: the owner of the file, the group to which the file belongs, and to all users, regardless of group.

Read permission allows a user to read the contents of the file, or in the case of directories, to list the contents of the directory (using `ls`). Write permission allows the user to write to and modify the file. For directories, write permission allows the user to create new files or delete files within that directory. Finally, execute permission allows the user to run the file as a program or shell script (if the file happens to be a program or shell script, that is). For directories, having execute permission allows the user to `cd` into the directory in question.

3.9.2 Interpreting file permissions

Let's look at an example to demonstrate file permissions. Using the `ls` command with the `-l` option will display a "long" listing of the file, including file permissions.

```
/home/larry/foo# ls -l stuff
-rw-r--r--   1 larry    users        505 Mar 13 19:05 stuff
/home/larry/foo#
```

The first field printed in the listing represents the file permissions. The third field is the owner of the file (`larry`), and the fourth field is the group to which the file belongs (`users`). Obviously, the last field is the name of the file (`stuff`), and we'll cover the other fields later.

This file is owned by `larry`, and belongs to the group `users`. Let's look at the file permissions. The string `-rw-r--r--` lists, in order, the permissions granted to the file's owner, the file's group, and everybody else.

The first character of the permissions string ("`-`") represents the type of file. A "`-`" just means that this is a regular file (as opposed to a directory or device driver). The next three letters ("`rw-`") represent the permissions granted to the file's owner, `larry`. The "`r`" stands for "read" and the "`w`" stands for "write". Thus, `larry` has read and write permission to the file `stuff`.

As we mentioned, besides read and write permission, there is also "execute" permission—represented by an "`x`". However, there is a "`-`" here in place of the "`x`", so Larry doesn't have execute permission on this file. This is fine, the file `stuff` isn't a program of any kind. Of course, because Larry owns the file, he may grant himself execute permission for the file if he so desires. This will be covered shortly.

The next three characters, `r--`, represent the group's permissions on the file. The group which owns this file is `users`. Because only an "`r`" appears here, any user which belongs to the group `users` may read this file.

The last three characters, also `r--`, represent the permissions granted to every other user on the system (other than the owner of the file and those in the group `users`). Again, because only an "`r`" is present, other users may read the file, but not write to it or execute it.

Here are some other examples of group permissions.

`-rwxr-xr-x` The owner of the file may read, write, and execute the file. Users in the file's group, and all other users, may read and execute the file.

`-rw-------` The owner of the file may read and write the file. No other user can access the file.

`-rwxrwxrwx` All users may read, write, and execute the file.

3.9.3 Dependencies

It is important to note that the permissions granted to a file also depend on the permissions of the directory in which the file is located. For example, even if a file is set to `-rwxrwxrwx`, other users cannot access the file unless they have read and execute access to the directory in which the file is located. For example, if Larry wanted to restrict access to all of his files, he could simply set the permissions on his home directory `/home/larry` to `-rwx------`. In this way, no other user has access to his directory, and all files and directories within it. Larry doesn't need to worry about the individual permissions on each of his files.

In other words, to access a file at all, you must have execute access to all directories along the file's pathname, and read (or execute) access to the file itself.

Usually, users on a UNIX system are very open with their files. The usual set of permissions given to files is `-rw-r--r--`, which will allow other users to read the file, but not change it in any way. The usual set of permissions given to directories is `-rwxr-xr-x`, which will allow other users to look through your directories, but not create or delete files within them.

However, many users wish to keep other users out of their files. Setting the permissions of a file to `-rw-------` will not allow any other user to access the file. Likewise, setting the permissions of a directory to `-rwx------` will keep other users out of the directory in question.

3.9.4 Changing permissions

The command `chmod` is used to set the permissions on a file. Only the owner of a file may change the permissions on that file. The syntax of `chmod` is:

`chmod {a,u,g,o}{+,-}{r,w,x}` ⟨*filenames*⟩

Briefly, you supply one or more of **a**ll, **u**ser, **g**roup, or **o**ther. Then you specify whether you are adding rights (+) or taking them away (-). Finally, you specify one or more of **r**ead, **w**rite, and e**x**ecute. Some examples of legal commands are:

`chmod a+r stuff`
: Gives all users read access to the file.

`chmod +r stuff`
: Same as above—if none of `a`, `u`, `g`, or `o` is specified, `a` is assumed.

`chmod og-x stuff`
: Remove execute permission from users other than the owner.

`chmod u+rwx stuff`
: Allow the owner of the file to read, write, and execute the file.

`chmod o-rwx stuff`
: Remove read, write, and execute permission from users other than the owner and users in the file's group.

3.10 Managing file links

Links allow you to give a single file multiple names. Files are actually identified to the system by their **inode number**, which is just the unique filesystem identifier for the file[3]. A directory is actually a listing of inode numbers with their corresponding filenames. Each filename in a directory is a **link** to a particular inode.

3.10.1 Hard links

The `ln` command is used to create multiple links for one file. For example, let's say that you have the file `foo` in a directory. Using `ls -i`, we can look at the inode number for this file.

```
# ls -i foo
22192 foo
#
```

Here, the file `foo` has an inode number of 22192 in the filesystem. We can create another link to `foo`, named `bar`:

```
# ln foo bar
```

With `ls -i`, we see that the two files have the same inode.

```
# ls -i foo bar
22192 bar   22192 foo
#
```

Now, accessing either `foo` or `bar` will access the same file. If you make changes to `foo`, those changes will be made to `bar` as well. For all purposes, `foo` and `bar` are the same file.

These links are known as *hard links* because they directly create a link to an inode. Note that you can only hard-link files on the same filesystem; symbolic links (see below) don't have this restriction.

When you delete a file with `rm`, you are actually only deleting one link to a file. If you use the command

```
# rm foo
```

then only the link named `foo` is deleted; `bar` will still exist. A file is only actually deleted on the system when it has no links to it. Usually, files have only one link, so using the `rm` command deletes the file. However, if a file has multiple links to it, using `rm` will only delete a single link; in order to delete the file, you must delete all links to the file.

The command `ls -l` will display the number of links to a file (among other information).

[3] The command `ls -i` will display file inode numbers.

```
# ls -l foo bar
-rw-r--r--   2 root     root           12 Aug  5 16:51 bar
-rw-r--r--   2 root     root           12 Aug  5 16:50 foo
#
```

The second column in the listing, "2", specifies the number of links to the file.

As it turns out, a directory is actually just a file containing information about link-to-inode translations. Also, every directory has at least two hard links in it: "`.`" (a link pointing to itself), and "`..`" (a link pointing to the parent directory). The root directory (`/`) "`..`" link just points back to `/`.

3.10.2 Symbolic links

Symbolic links are another type of link, which are somewhat different than hard links. A symbolic link allows you to give a file another name, but it doesn't link the file by inode.

The command `ln -s` will create a symbolic link to a file. For example, if we use the command

```
# ln -s foo bar
```

we will create the symbolic link `bar` pointing to the file `foo`. If we use `ls -i`, we will see that the two files have different inodes, indeed.

```
# ls -i foo bar
22195 bar   22192 foo
#
```

However, using `ls -l`, we see that the file `bar` is a symlink pointing to `foo`.

```
# ls -l foo bar
lrwxrwxrwx   1 root     root            3 Aug  5 16:51 bar -> foo
-rw-r--r--   1 root     root           12 Aug  5 16:50 foo
#
```

The permission bits on a symbolic link are not used (they always appear as `rwxrwxrwx`). Instead, the permissions on the symbolic link are determined by the permissions on the target of the symbolic link (in our example, the file `foo`).

Functionally, hard links and symbolic links are similar, but there are some differences. For one thing, you can create a symbolic link to a file which doesn't exist; the same is not true for hard links. Symbolic links are processed by the kernel differently than hard links are, which is just a technical difference but sometimes an important one. Symbolic links are helpful because they identify what file they point to; with hard links, there is no easy way to determine which files are linked to the same inode.

Links are used in many places on the Linux system. Symbolic links are especially important to the shared library images in `/lib`. See Section 4.7.2 for more information.

3.11 Job Control

3.11.1 Jobs and processes

Job control is a feature provided by many shells (Bash and Tcsh included) which allows you to control multiple running commands, or **jobs**, at once. Before we can delve much further, we need to talk about **processes**.

Every time you run a program, you start what is known as a *process*—which is just a fancy name for a running program. The command `ps` displays a list of currently running processes. Here's an example:

```
/home/larry# ps

  PID TT STAT  TIME COMMAND
   24  3 S     0:03 (bash)
  161  3 R     0:00 ps

/home/larry#
```

The `PID` listed in the first column is the **process ID**, a unique number given to every running process. The last column, `COMMAND`, is the name of the running command. Here, we're only looking at the processes which Larry is currently running[4]. These are `bash` (Larry's shell), and the `ps` command itself. As you can see, `bash` is running concurrently with the `ps` command. `bash` executed `ps` when Larry typed the command. After `ps` is finished running (after the table of processes is displayed), control is returned to the `bash` process, which displays the prompt, ready for another command.

A running process is known as a *job* to the shell. The terms *process* and *job* are interchangeable. However, a process is usually referred to as a "job" when used in conjunction with **job control**—a feature of the shell which allows you to switch between several independent jobs.

In most cases users are only running a single job at a time—that being whatever command they last typed to the shell. However, using job control, you can run several jobs at once, switching between them as needed. How might this be useful? Let's say that you're editing a text file and need to suddenly interrupt your editing and do something else. With job control, you can temporarily suspend the editor, and back at the shell prompt start to work on something else. When you're done, you can start the editor back up, and be back where you started, as if you never left the editor. This is just one example. There are many practical uses for job control.

3.11.2 Foreground and background

Jobs can either be in the **foreground** or in the **background**. There can only be one job in the foreground at any one time. The foreground job is the job which you interact with—it receives input from the keyboard and sends output to your screen. (Unless, of course, you have redirected input

[4] There are many other processes running on the system as well—"ps -aux" lists them all.

or output, as described in Section 3.8). On the other hand, jobs in the background do not receive input from the terminal—in general, they run along quietly without need for interaction.

Some jobs take a long time to finish, and don't do anything interesting while they are running. Compiling programs is one such job, as is compressing a large file. There's no reason why you should sit around being bored while these jobs complete their tasks; you can just run them in the background. While the jobs are running in the background, you are free to run other programs.

Jobs may also be **suspended**. A suspended job is a job that is not currently running, but is temporarily stopped. After you suspend a job, you can tell the job to continue, in the foreground or the background as needed. Resuming a suspended job will not change the state of the job in any way—the job will continue to run where it left off.

Note that suspending a job is not equal to *interrupting* a job. When you interrupt a running process (by hitting your interrupt key, which is usually `ctrl-C`)[5], it kills the process, for good. Once the job is killed, there's no hope of resuming it; you'll have to re-run the command. Also note that some programs trap the interrupt, so that hitting `ctrl-C` won't immediately kill the job. This is to allow the program to perform any necessary cleanup operations before exiting. In fact, some programs simply don't allow you to kill them with an interrupt at all.

3.11.3 Backgrounding and killing jobs

Let's begin with a simple example. The command `yes` is a seemingly useless command which sends an endless stream of y's to standard output. (This is actually useful. If you piped the output of `yes` to another command which asked a series of yes and no questions, the stream of y's would confirm all of the questions.)

Try it out.

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

The y's will continue *ad infinitum*. You can kill the process by hitting your interrupt key, which is usually `ctrl-C`. So that we don't have to put up with the annoying stream of y's, let's redirect the standard output of `yes` to `/dev/null`. As you may remember, `/dev/null` acts as a "black hole" for data. Any data sent to it will disappear. This is a very effective method of quieting an otherwise verbose program.

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

[5] The interrupt key can be set using the `stty` command. The default on most systems is `ctrl-C`, but we can't guarantee the same for your system.

Ah, much better. Nothing is printed, but the shell prompt doesn't come back. This is because `yes` is still running, and is sending those inane `y`'s to `/dev/null`. Again, to kill the job, hit the interrupt key.

Let's suppose that we wanted the `yes` command to continue to run, but wanted to get our shell prompt back to work on other things. We can put `yes` into the background, which will allow it to run, but without need for interaction.

One way to put a process in the background is to append an "`&`" character to the end of the command.

```
/home/larry# yes > /dev/null &
[1] 164
/home/larry#
```

As you can see, we have our shell prompt back. But what is this "`[1] 164`"? And is the `yes` command really running?

The "`[1]`" represents the **job number** for the `yes` process. The shell assigns a job number to every running job. Because `yes` is the one and only job that we're currently running, it is assigned job number `1`. The "`164`" is the process ID, or PID, number given by the system to the job. Either number may be used to refer to the job, as we'll see later.

You now have the `yes` process running in the background, continuously sending a stream of `y`'s to `/dev/null`. To check on the status of this process, use the shell internal command `jobs`.

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

Sure enough, there it is. You could also use the `ps` command as demonstrated above to check on the status of the job.

To terminate the job, use the command `kill`. This command takes either a job number or a process ID number as an argument. This was job number 1, so using the command

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

will kill the job. When identifying the job with the job number, you must prefix the number with a percent ("`%`") character.

Now that we've killed the job, we can use `jobs` again to check on it:

```
/home/larry# jobs

[1]+  Terminated              yes >/dev/null

/home/larry#
```

The job is in fact dead, and if we use the `jobs` command again nothing should be printed.

You can also kill the job using the process ID (PID) number, which is printed along with the job ID when you start the job. In our example, the process ID is 164, so the command

```
/home/larry# kill 164
```

is equivalent to

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

You don't need to use the "%" when referring to a job by its process ID.

3.11.4 Stopping and restarting jobs

There is another way to put a job into the background. You can start the job normally (in the foreground), **stop** the job, and then restart it in the background.

First, start the `yes` process in the foreground, as you normally would:

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

Again, because `yes` is running in the foreground, you shouldn't get your shell prompt back.

Now, instead of interrupting the job with `ctrl-C`, we'll *suspend* the job. Suspending a job doesn't kill it: it only temporarily stops the job until you restart it. To do this, you hit the suspend key, which is usually `ctrl-Z`.

```
/home/larry# yes > /dev/null
ctrl-Z
[1]+  Stopped                 yes >/dev/null
/home/larry#
```

While the job is suspended, it's simply not running. No CPU time is used for the job. However, you can restart the job, which will cause the job to run again as if nothing ever happened. It will continue to run where it left off.

To restart the job in the foreground, use the command `fg` (for "foreground").

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

The shell prints the name of the command again so you're aware of which job you just put into the foreground. Stop the job again, with `ctrl-Z`. This time, use the command `bg` to put the job into the background. This will cause the command to run just as if you started the command with "&" as in the last section.

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

And we have our prompt back. `jobs` should report that `yes` is indeed running, and we can kill the job with `kill` as we did before.

How can we stop the job again? Using `ctrl-Z` won't work, because the job is in the background. The answer is to put the job in the foreground, with `fg`, and then stop it. As it turns out you can use `fg` on either stopped jobs or jobs in the background.

There is a big difference between a job in the background and a job which is stopped. A stopped job is not running—it's not using any CPU time, and it's not doing any work (the job still occupies system memory, although it may be swapped out to disk). A job in the background is running, and using memory, as well as completing some task while you do other work. However, a job in the background may try to display text on to your terminal, which can be annoying if you're trying to work on something else. For example, if you used the command

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

without redirecting stdout to `/dev/null`, a stream of `y`'s would be printed to your screen, without any way of interrupting it (you can't use `ctrl-C` to interrupt jobs in the background). In order to stop the endless `y`'s, you'd have to use the `fg` command, to bring the job to the foreground, and then use `ctrl-C` to kill it.

Another note. The `fg` and `bg` commands normally foreground or background the job which was last stopped (indicated by a "+" next to the job number when you use the command `jobs`). If you are running multiple jobs at once, you can foreground or background a specific job by giving the job ID as an argument to `fg` or `bg`, as in

```
/home/larry# fg %2
```

(to foreground job number 2), or

```
/home/larry# bg %3
```

(to background job number 3). You can't use process ID numbers with `fg` or `bg`.

Furthermore, using the job number alone, as in

```
/home/larry# %2
```

is equivalent to

```
/home/larry# fg %2
```

Just remember that using job control is a feature of the shell. The commands `fg`, `bg` and `jobs` are internal to the shell. If for some reason you use a shell which does not support job control, don't expect to find these commands available.

In addition, there are some aspects of job control which differ between Bash and Tcsh. In fact, some shells don't provide job control at all—however, most shells available for Linux support job control.

3.12 Using the `vi` Editor

A **text editor** is simply a program used to edit files which contain text, such as a letter, C program, or a system configuration file. While there are many such editors available for Linux, the only editor which you are guaranteed to find on any UNIX system is `vi`— the "visual editor". `vi` is not the easiest editor to use, nor is it very self-explanatory. However, because it is so common in the UNIX world, and at times you may be required to use it, it deserves some documentation here.

Your choice of an editor is mostly a question of personal taste and style. Many users prefer the baroque, self-explanatory and powerful **Emacs**—an editor with more features than any other single program in the UNIX world. For example, Emacs has its own built-in dialect of the LISP programming language, and has many extensions (one of which is an "Eliza"-like AI program). However, because Emacs and all of its support files are relatively large, you may not have access to it on many systems. `vi`, on the other hand, is small and powerful, but more difficult to use. However, once you know your way around `vi`, it's actually very easy. It's just the learning curve which is sometimes difficult to cross.

This section is a coherent introduction to `vi`—we won't discuss all of its features, just the ones you need to know to get you started. You can refer to the man page for `vi` if you're interested in learning about more of this editor's features. Or, you can read the book *Learning the* `vi` *Editor* from O'Reilly and Associates. See Appendix A for information.

3.12.1 Concepts

While using `vi`, at any one time you are in one of three modes of operation. These modes are known as *command mode*, *insert mode*, and *last line mode*.

When you start up `vi`, you are in *command mode*. This mode allows you to use certain commands to edit files or to change to other modes. For example, typing "`x`" while in command mode deletes the character underneath the cursor. The arrow keys move the cursor around the file which you're editing. Generally, the commands used in command mode are one or two characters long.

You actually insert or edit text within *insert mode*. When using `vi`, you'll probably spend most of your time within this mode. You start insert mode by using a command such as "`i`" (for "insert") from command mode. While in insert mode, you are inserting text into the document from your current cursor location. To end insert mode and return to command mode, press $\boxed{\texttt{esc}}$.

Last line mode is a special mode used to give certain extended commands to `vi`. While typing these commands, they appear on the last line of the screen (hence the name). For example, when you type "`:`" from command mode, you jump into last line mode, and can use commands such as "`wq`" (to write the file and quit `vi`), or "`q!`" (to quit `vi` without saving changes). Last line mode is generally used for `vi` commands which are longer than one character. In last line mode, you enter a single-line command and press `enter` to execute it.

3.12.2 Starting vi

The best way to understand these concepts is to actually fire up `vi` and edit a file. In the example "screens" below, we're only going to show a few lines of text, as if the screen was only six lines high (instead of twenty-four).

The syntax for `vi` is

vi ⟨*filename*⟩

where ⟨*filename*⟩ is the name of the file that you wish to edit.

Start up `vi` by typing

`/home/larry#` *vi test*

which will edit the file `test`. You should see something like

```
~
~
~
~
~
~
"test" [New file]
```

The column of "~" characters indicates that you are the end of the file.

3.12.3 Inserting text

You are now in command mode; in order to insert text into the file, press `i` (which will place you into insert mode), and begin typing.

```
Now is the time for all good men to come to the aid of the party.
~
~
~
~
~
```

While inserting text, you may type as many lines as you wish (pressing `return` after each, of course), and you may correct mistakes using the backspace key.

To end insert mode, and return to command mode, press `esc`.

While in command mode, you can use the arrow keys to move around the file. Here, because we only have one line of text, trying to use the up- or down-arrow keys will probably cause `vi` to beep at you.

There are several ways to insert text, other than using the `i` command. For example, the `a` command inserts text beginning *after* the current cursor position, instead of on the current cursor position. For example, use the left arrow key to move the cursor between the words "good" and "men".

```
Now is the time for all good_men to come to the aid of the party.
~
~
~
~
~
```

Press `a`, to start insert mode, type "`wo`", and then hit `esc` to return to command mode.

```
Now is the time for all good women to come to the aid of the party.
~
~
~
~
~
```

To begin inserting text at the line below the current one, use the `o` command. For example, press `o` and type another line or two:

```
Now is the time for all good women to come to the aid of the party.
Afterwards, we'll go out for pizza and beer.
~
~
~
~
```

Just remember that at any time you're either in command mode (where commands such as `i`, `a`, or `o` are valid), or in insert mode (where you're inserting text, followed by `esc` to return to command mode), or last line mode (where you're entering extended commands, as discussed below).

3.12.4 Deleting text

From command mode, the `x` command deletes the character under the cursor. If you press `x` five times, you'll end up with:

```
Now is the time for all good women to come to the aid of the party.
Afterwards, we'll go out for pizza and_
~
~
~
~
```

Now press `a`, insert some text, followed by `esc`:

```
Now is the time for all good women to come to the aid of the party.
Afterwards, we'll go out for pizza and Diet Coke.
~
~
~
~
```

You can delete entire lines using the command `dd` (that is, press `d` twice in a row). If your cursor is on the second line, and you type `dd`,

```
Now is the time for all good women to come to the aid of the party.
~
~
~
~
~
```

To delete the word which the cursor is on, use the `dw` command. Place the cursor on the word "good", and type `dw`.

```
Now is the time for all women to come to the aid of the party.
~
~
~
~
~
```

3.12.5 Changing text

You can replace sections of text using the `R` command. Place the cursor on the first letter in "party", press `R`, and type the word "hungry".

```
Now is the time for all women to come to the aid of the hungry.
~
~
~
~
~
```

Using `R` to edit text is much like the `i` and `a` commands, but `R` overwrites text, instead of inserting it.

The `r` command replaces the single character under the cursor. For example, move the cursor to the beginning of the word "Now", and type `r` followed by `C`, you'll have:

```
Cow is the time for all women to come to the aid of the hungry.
~
~
~
~
~
```

The "~" command changes the case of the letter under the cursor from upper- to lower-case, and vise versa, For example, if you place the cursor on the "o" in "Cow", above, and repeatedly press ~, you'll end up with:

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE HUNGRY.
~
~
~
~
~
```

3.12.6 Moving commands

You already know how to use the arrow keys to move around the document. In addition, you can use the `h`, `j`, `k`, and `l` commands to move the cursor left, down, up, and right, respectively. This comes in handy when (for some reason) your arrow keys aren't working correctly.

The `w` command moves the cursor to the beginning of the next word; the `b` moves it to the beginning of the previous word.

The `0` (that's a zero) command moves the cursor to the beginning of the current line, and the `$` command moves it to the end of the line.

When editing large files, you'll want to move forwards or backwards through the file a screenful at a time. Pressing ctrl-F moves the cursor one screenful forward, and ctrl-B moves it a screenful back.

In order to move the cursor to the end of the file, type `G`. You can also move to an arbitrary line; for example, typing the command `10G` would move the cursor to line 10 in the file. To move to the beginning of the file, use `1G`.

You can couple moving commands with other commands, such as deletion. For example, the command `d$` will delete everything from the cursor to the end of the line; `dG` will delete everything from the cursor to the end of the file, and so on.

3.12.7 Saving files and quitting vi

To quit `vi` without making changes to the file, use the command `:q!`. When you type the ":", the cursor will move to the last line on the screen; you'll be in last line mode.

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE HUNGRY.
~
~
~
~
~
:_
```

In last line mode, certain extended commands are available. One of them is `q!`, which quits `vi` without saving. The command `:wq` saves the file and then exits `vi`. The command `ZZ` (from command mode, without the ":") is equivalent to `:wq`. Remember that you must press `enter` after a command entered in last line mode.

To save the file without quitting vi, just use `:w`.

3.12.8 Editing another file

To edit another file, use the `:e` command. For example, to stop editing `test`, and edit the file `foo` instead, use the command

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE HUNGRY.
~
~
~
~
~
:e foo_
```

If you use `:e` without saving the file first, you'll get the error message

```
No write since last change (":edit!" overrides)
```

which simply means that `vi` doesn't want to edit another file until you save the first one. At this point, you can use `:w` to save the original file, and then use `:e`, or you can use the command

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE HUNGRY.
~
~
~
~
~
:e!  foo_
```

The "`!`" tells `vi` that you really mean it—edit the new file without saving changes to the first.

3.12.9 Including other files

If you use the :r command, you can include the contents of another file in the current file. For example, the command

```
:r foo.txt
```

would insert the contents of the file foo.txt in the text at the current cursor location.

3.12.10 Running shell commands

You can also run shell commands from within vi. The :r! command works like :r, but instead of reading a file, it inserts the output of the given command into the buffer at the current cursor location. For example, if you use the command

```
:r! ls -F
```

you'll end up with

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE HUNGRY.
letters/
misc/
papers/
~
~
```

You can also "shell out" of vi, in other words, run a command from within vi, and return to the editor when you're done. For example, if you use the command

```
:! ls -F
```

the ls -F command will be executed, and the results displayed on the screen, but not inserted into the file which you're editing. If you use the command

```
:shell
```

vi will start an instance of the shell, allowing you to temporarily put vi "on hold" while you execute other commands. Just logout of the shell (using the exit command) to return to vi.

3.12.11 Getting help

vi doesn't provide much in the way of interactive help (most UNIX programs don't), but you can always read the man page for vi. vi is a visual front-end to the ex editor; it is ex which handles many of the last-line mode commands in vi. So, in addition to reading the man page for vi, see ex as well.

3.13 Customizing your Environment

The shell provides many mechanisms to customize your work environment. As we've mentioned before, the shell is more than a command interpreter—it is also a powerful programming language. While writing shell scripts is an extensive subject, we'd like to introduce you to some of the ways that you can simplify your work on a UNIX system by using these advanced features of the shell.

As we have mentioned before, different shells use different syntaxes when executing shell scripts. For example, Tcsh uses a C-like syntax, while Bourne shells use another type of syntax. In this section, we won't be running into many of the differences between the two, but we will assume that shell scripts are executed using the Bourne shell syntax.

3.13.1 Shell scripts

Let's say that you use a series of commands often, and would like to shorten the amount of required typing by grouping all of them together into a single "command". For example, the commands

```
/home/larry# cat chapter1 chapter2 chapter3 > book
/home/larry# wc -l book
/home/larry# lp book
```

would concatenate the files `chapter1`, `chapter2`, and `chapter3` and place the result in the file `book`. Then, a count of the number of lines in `book` would be displayed, and finally `book` would be printed with the `lp` command.

Instead of typing all of these commands, you could group them into a **shell script**. We described shell scripts briefly in Section 3.13.1. The shell script used to run all of these commands would look like

```
#!/bin/sh
# A shell script to create and print the book

cat chapter1 chapter2 chapter3 > book
wc -l book
lp book
```

If this script was saved in the file `makebook`, you could simply use the command

```
/home/larry# makebook
```

to run all of the commands in the script. Shell scripts are just plain text files; you can create them with an editor such as `emacs` or `vi` [6].

Let's look at this shell script. The first line, "`#!/bin/sh`", identifies the file as a shell script, and tells the shell how to execute the script. It instructs the shell to pass the script to `/bin/sh`

[6] `vi` is covered in Section 3.12.

for execution, where `/bin/sh` is the shell program itself. Why is this important? On most UNIX systems, `/bin/sh` is a Bourne-type shell, such as Bash. By forcing the shell script to run using `/bin/sh`, we are ensuring that the script will run under a Bourne-syntax shell (instead of, say, a C shell). This will cause your script to run using the Bourne syntax even if you use Tcsh (or another C shell) as your login shell.

The second line is a *comment*. Comments begin with the character "`#`" and continue to the end of the line. Comments are ignored by the shell—they are commonly used to identify the shell script to the programmer.

The rest of the lines in the script are just commands, as you would type them to the shell directly. In effect, the shell reads each line of the script and runs that line as if you had typed it at the shell prompt.

Permissions are important for shell scripts. If you create a shell script, you must make sure that you have execute permission on the script in order to run it[7]. The command

```
/home/larry# chmod u+x makebook
```

can be used to give yourself execute permission on the shell script `makebook`.

3.13.2 Shell variables and the environment

The shell allows you to define **variables**, as most programming languages do. A variable is just a piece of data which is given the name.

◇ Note that Tcsh, as well as other C-type shells, use a different mechanism for setting variables than is described here. This discussion assumes the use of a Bourne shell, such as Bash (which you're probably using). See the Tcsh man page for details.

When you assign a value to a variable (using the "`=`" operator), you can access the variable by prepending a "`$`" to the variable name, as demonstrated below.

```
/home/larry# foo="hello there"
```

The variable `foo` is given the value "`hello there`". You can now refer to this value by the variable name, prefixed with a "`$`" character. The command

```
/home/larry# echo $foo
hello there
/home/larry#
```

produces the same results as

```
/home/larry# echo "hello there"
hello there
/home/larry#
```

[7] When you create text files, the default permissions usually don't include execute permission.

These variables are internal to the shell. This means that only the shell can access these variables. This can be useful in shell scripts; if you need to keep track of a filename, for example, you can store it in a variable, as above. Using the command `set` will display a list of all defined shell variables.

However, the shell allows you to **export** variables to the **environment**. The environment is the set of variables which all commands that you execute have access to. Once you define a variable inside the shell, exporting it makes that variable part of the environment as well. The `export` command is used to export a variable to the environment.

◇ Again, here we differ between Bash and Tcsh. If you're using Tcsh, another syntax is used for setting environment variables (the `setenv` command is used). See the Tcsh man page for more information.

The environment is very important to the UNIX system. It allows you to configure certain commands just by setting variables which the commands know about.

Here's a quick example. The environment variable `PAGER` is used by the `man` command. It specifies the command to use to display man pages one screenful at a time. If you set `PAGER` to be the name of a command, it will use that command to display the man pages, instead of `more` (which is the default).

Set `PAGER` to "`cat`". This will cause output from `man` to be displayed to the screen all at once, without breaking it up into pages.

```
/home/larry# PAGER="cat"
```

Now, export `PAGER` to the environment.

```
/home/larry# export PAGER
```

Try the command `man ls`. The man page should fly past your screen without pausing for you.

Now, if we set `PAGER` to "`more`", the `more` command will be used to display the man page.

```
/home/larry# PAGER="more"
```

Note that we don't have to use the `export` command after we change the value of `PAGER`. We only need to export a variable once; any changes made to it thereafter will automatically be propagated to the environment.

The man pages for a particular command will tell you if the command uses any environment variables; for example, the `man` man page explains that `PAGER` is used to specify the pager command. Some commands share environment variables; for example, many commands use the `EDITOR` environment variable to specify the default editor to use when one is needed.

The environment is also used to keep track of important information about your login session. An example is the `HOME` environment variable, which contains the name of your home directory.

```
/home/larry/papers# echo $HOME
/home/larry
```

Another interesting environment variable is `PS1`, which defines the main shell prompt. For example,

```
/home/larry# PS1="Your command, please: "
Your command, please:
```

To set the prompt back to our usual (which contains the current working directory followed by a "#" symbol),

```
Your command, please: PS1="\w# "
/home/larry#
```

The `bash` man page describes the syntax used for setting the prompt.

3.13.2.1 The PATH environment variable

When you use the `ls` command, how does the shell find the `ls` executable itself? In fact, `ls` is found in `/bin/ls` on most systems. The shell uses the environment variable `PATH` to locate executable files for commands which you type.

For example, your `PATH` variable may be set to:

```
/bin:/usr/bin:/usr/local/bin:.
```

This is a list of directories for the shell to search, each directory separated by a ":". When you use the command `ls`, the shell first looks for `/bin/ls`, then `/usr/bin/ls`, and so on.

Note that the `PATH` has nothing to do with finding regular files. For example, if you use the command

```
/home/larry# cp foo bar
```

The shell does not use `PATH` to locate the files `foo` and `bar`—those filenames are assumed to be complete. The shell only uses `PATH` to locate the `cp` executable.

This saves you a lot of time; it means that you don't have to remember where all of the command executables are stored. On many systems, executables are scattered about in many places, such as `/usr/bin`, `/bin`, or `/usr/local/bin`. Instead of giving the command's full pathname (such as `/usr/bin/cp`), you can simply set `PATH` to the list of directories that you want the shell to automatically search.

Notice that `PATH` contains ".", which is the current working directory. This allows you to create a shell script or program and run it as a command from your current directory, without having to specify it directly (as in `./makebook`). If a directory isn't on your `PATH`, then the shell will not search it for commands to run—this includes the current directory.

3.13.3 Shell initialization scripts

In addition to shell scripts that you create, there are a number of scripts that the shell itself uses for certain purposes. The most important of these are your **initialization scripts**, scripts automatically executed by the shell when you login.

The initialization scripts themselves are simply shell scripts, as described above. However, they are very useful in setting up your environment by executing commands automatically when you login. For example, if you always use the `mail` command to check your mail when you login, you place the command in your initialization script so it will be executed automatically.

Both Bash and Tcsh distinguish between a **login shell** and other invocations of the shell. A login shell is a shell invoked at login time; usually, it's the only shell which you'll use. However, if you "shell out" of another program, such as `vi`, you start another instance of the shell, which isn't your login shell. In addition, whenever you run a shell script, you automatically start another instance of the shell to execute the script.

The initialization files used by Bash are: `/etc/profile` (set up by the system administrator, executed by all Bash users at login time), `$HOME/.bash_profile` (executed by a login Bash session), and `$HOME/.bashrc` (executed by all non-login instances of Bash). If `.bash_profile` is not present, `.profile` is used instead.

Tcsh uses the following initialization scripts: `/etc/csh.login` (executed by all Tcsh users at login time), `$HOME/.tcshrc` (executed a login time and by all new instances of Tcsh), and `$HOME/.login` (executed at login time, following `.tcshrc`). If `.tcshrc` is not present, `.cshrc` is used instead.

To fully understand the function of these files, you'll need to learn more about the shell itself. Shell programming is a complicated subject, far beyond the scope of this book. See the man pages for `bash` and/or `tcsh` to learn more about customizing your shell environment.

3.14 So You Want to Strike Out on Your Own?

Hopefully we have provided enough information to give you a basic idea of how to use the system. Keep in mind that most of the interesting and important aspects of Linux aren't covered here—these are the very basics. With this foundation, before long you'll be up and running complicated applications and fulfilling the potential of your system. If things don't seem exciting at first, don't despair—there is much to be learned.

One indispensable tool for learning about the system is to read the man pages. While many of the man pages may appear confusing at first, if you dig beneath the surface there is a wealth of information contained therein.

We also suggest reading a complete book on using a UNIX system. There is much more to UNIX than meets the eye—unfortunately, most of it is beyond the scope of this book. Some good UNIX books to look at are listed in Appendix A.