Unix

Contents

1. Operating Systems

An operating system's primary purpose is to support programs that actually do the work you're insterested in. For instance, you may be using an editor so you can create a document. This editor could not do its work without help from the operating system - it needs this help for interacting with your terminal, your files, and the rest of the computer.

If all the operating system does is support your applications, why do you need a whole book just to talk about the operating system? There are lots of routine maintenance activities (apart from your major programs) that you also need to do. In the case of Linux, the operating system also contains a lot of "mini-applications" to help you do your work more efficently. Knowing the operating system can be helpful when you're not working in one huge application.

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

The original design philosophy for Unix was to distribute functionality into small parts, the programs. That way, you can relatively easily achieve new functionality and new features by combining the small parts (programs) in new ways. And if new utilities appear (and they do), you can integrate them into your old toolbox. Unfortunately, programs grow larger and more feature-packed on Unix as well these days, but some of the flexibility and interoperability is there to stay. For example, to write this document, I could be using the following programs actively; `fvwm` to manage my "windows", `emacs` to edit the text, `LaTeX` to format it, `xdvi` to preview it, `dvips` to prepare it for printing and then `lpr` to print it. If I got a new, better `dvi` previewer tommorow, I could use it instead of `xdvi` without changing the rest of my setup.

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

The key part of an operating system is called the "**kernel**." In many operating systems, like Unix, OS/2, or VMS, the kernel supplies functions for running programs to use, and schedules them to be run. It basically says program A can get so much time, program B can get this much time, etc. One school of thought says that kernels should be very small, and not supply a lot of resources, depending on programs to pick up the work. This allows the kernel to be small and fast, but may make programs bigger. Kernels designed like this are called micro-kernels. Another group of people believe that kernels that offer more services to applications are better and make more efficent operating systems. Most versions of Unix are designed like this. While it may seem at first that all micro-kernels should be smaller than all macro-kernels, the terms "micro" and "macro" really aren't referring to size of the kernel but a philosophy of operating system design.

2. Unix History

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

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

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

Unix was slow to catch on outside of academic institutions but soon was popular with businesses as well. The Unix of today is different from the Unix of 1970. It has two major versions: System V, from **U**nix **S**ystem **L**aboratories (**USL**), a subsiderary of Novell, and **BSD**, **B**erkeley **S**oftware **D**istribution. The USL version is now up to its forth release, or SVR4, while BSD's latest version is 4.4. However, there are many different versions of Unix besides these two. Most versions of Unix are developed by software companies and derive from one of the two groupings. Recently, the versions of Unix that are actually used incorporate features from both of them.

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

Please note the difference between **Unix** and **UNIX**. When I say "Unix" I am talking about Unix versions in generally, whether or not USL is involved in them. "UNIX" is the current version of Unix from USL. The distinction is because UNIX is a trademark of X/Open. (Officially, anybody can create a UNIX operating system, as long as it passes tests from X/Open. Since the tests haven't been created yet and are likely to cost money, Linux is currently not a "real" UNIX.)

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

2.1 Linux History

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

Linux began with a project to explore the 386 chip. One of Linus's earlier projects was a program that would switch between printing `AAAA` and `BBBB`. This later evolved to Linux. Linux has been copyrighted under the terms of the **GNU** **G**eneral **P**ublic **L**icense (**GPL**). This is a license written by the **F**ree **S**oftware **F**oundation (**FSF**) that is designed to prevent people from restricting the distribution of software. In brief it says that although you can charge as much as you'd like for giving a copy away, you can't prevent the person you sold it to from giving it away for free. It also means that the source code must also be available. This is useful for programmers. The license also says that anyone who modifies the program must also make his version freely redistributable.

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

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

Of course, Torvalds isn't the only big name in Linux's development. The following people also deserve to be recognized: H. J. Lu - has maintained gcc and the C Library, two items needed for programming.

3. Starting to Use Your Computer

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

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

3.1 Power to the Computer

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

```
No-system disk or disk error
```

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

If there isn't a floppy disk in the drive, the BIOS looks for a **m**aster **b**oot **r**ecord (**MBR**) on the hard disk. It will start executing the code found there, which loads the operating system. On systems, **LILO**, the **LI**nux **LO**ader, can occupy the MBR position, and will load. For now, we'll assume that happens and that starts to load. (Your particular distribution may handle booting from the hard disk differently. Check with the documentation included in that distribution.

3.2 Takes Over

Before reading this section, you should know that nothing in it is needed to actually use Linux. It is only here for your own enjoyment and interest, but if you find it boring or overly technical, skip over it!

After the BIOS passes control to LILO, LILO passes control to Linux. (This is assuming you have configured to boot by default. It is also possible for LILO to call DOS or some other PC operating system.) The first thing that does once it starts executing is to change to protected mode. The 80386 CPU that controls your computer has two modes (for our purposes) called **real mode** and **protected mode**. DOS runs in real mode, as does the BIOS. However, for more advanced operating systems, it is necessary to run in protected mode. Therefore, when Linux boots, it discardes the BIOS.

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

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

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

- The file system might be checked. What is a file system, you might ask? A **file system** is the layout of files on the hard disk. It let's Unix know which parts of the disk are already used, and which aren't. Unfortunately, due to various factors such as power losses, what the file system information thinks is going on in the rest of the disk and the actually layout of the rest of the disk are in conflict. A special program, called `fsck`, can find these situations and hopefully correct them.
- Special routing programs for **networks** are run.
- **Temporary files** left by some programs may be **deleted**.
- The **system clock** can be correctly **updated**. This is trickier then one might think, since Unix, by default, wants the time in GMT and your CMOS clock, a battery powered clock in your computer, is probably set on local time.

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

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

2.3 The User Acts

This section actually contains information needed to know how to use Linux. The first thing you have to do to use a Unix machine is to identify yourself. This process, knowing as "**logging in**", is Unix's way of knowing that users are authorized to use the system. It asks for an account name and password. An **account name** is normally similar to your regular name; you should have already received one from your system administrator, or created your own if you are the system administrator. (Information on doing this should be available in Installation and Getting Started or The System Adminstrator's Guide.)

You should see, after all the boot-up procedures are done, something like the following:

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

However, it's possible that what the system presents you with does not look like this. Instead of a boring text mode screen, it is graphical. However, it will still ask you to login, and will function mostly the same way. If this is the case on your system, you are going to be using The X Window System. This means that you will be presented with a windowing system. However, logging in will be similar. If you are using X, look for a giant X is the margin.

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

`mousehouse` is, by the way, the "**name**" **of the machine** I'm working on. It is possible that when you installed, you were prompted for some very witty name.

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

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

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

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

```
/home/larry#
```

If you've already determined you're using X Windows, you'll probably see a prompt like the one above in a "**window**" somewhere on the screen. (A "window" is simply a rectangular box.) To type into the prompt, move the mouse cursor (it probably looks like a big "x") using the mouse into the window.

2.4 Leaving the Computer

Do not just turn off the computer! You risk losing valuable data! Unlike most versions of DOS, it's a bad thing to just hit the power switch when you're done using the computer. It is also bad to reboot the machine (with the reset button) without first taking proper precautions. Linux, in order to improve performance, caches the disk. This means it temporarily stores part of the permanent storage in RAM. The idea of what thinks the disk should be and what the disk actually contains is syncronized every 30 seconds. In order to turn off or reboot the computer, you'll have to go through a procedure telling it to stop caching disk information.

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

```
/home/larry# logout
Welcome to the mousehouse. Please have some cheese.
mousehouse login:
```

Now another user can login.

2.5 Turning the Computer Off

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

```
mousehouse login: root
Password:

Linux, version 0.99pl10.
/# shutdown now

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

The command `shutdown now` prepares the system to be reset or turned off. Wait for a message saying it is safe to and then **reset** or **turn off** the system. You must go through this procedure, however. You risk losing your work if you don't.

A quick message to the lazy: an alternative to the logout/login approach is to use the command `su` (run a shell with a **s**ubstitute **u**ser and group IDs). As a normal user, from your prompt, type `su` and **Return**. It should prompt you for the root password, and then give you root privileges. Now you can shutdown the system.

3. Kernel Messages

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

3.1 Starting Messages

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

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

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

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

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

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

Here, it found 3 serial ports. A serial port is the equivalent of a DOS **COM** port, and is a device normally used with modems and mice. What it is trying to say is that serial port 0 (`COM1`) has an address of `0x03f8`. When it interrupts the kernel, usually to say that it has data, it uses `IRQ 4`. An **IRQ** is another means of a peripheral talking to the software. Each serial port also has a controller chip. The usual one for a port to have is a `16450`; other values possible are `8250` and

`16550`. The differences are beyond the scope of this book.

• Next comes the **parallel port** driver. A parallel port is normally connected to a printer, and the names for the parallel ports (in) start with lp. lp stands for Line Printer, although it could be a laser printer.

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

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

• The Linux kernel also tells you a little about **memory usage**:

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

This said that the machine had 8 megabytesmegabyte of memory. Some of this memory was reserved for the kernel - just the operating system. The rest of it could be used by programs. The other type of "memory" is general called a hard disk. It's like a large floppy disk permenantly in your computer - the contents stay around even when the power is off.

• The kernel now moves onto looking at your **floppy drives**. In this example, the machine has two drives. In DOS, drive "**`A`**" is a 5 inch drive, and drive "**`B`**" is a 3 inch drive. Linux calls drive "A" **`fd0`**, and drive "B" **`fd1`**.

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

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

```
SLIP: version 0.7.5 (4 channels): OK
plip.c:v0.04 Mar 19 1993 Donald Becker (becker@super.org)
plip0: using parallel port at 0x3bc, IRQ 5.
plip1: using parallel port at 0x378, IRQ 7.
plip2: using parallel port at 0x278, IRQ 2.
S390.c:v0.99-10 5/28/93 for 0.99.6+ Donald Becker (becker@super.org)
WD80x3 ethercard probe at 0x280: FF FF FF FF FF FF not found (0x7f8).
3c503 probe at 0x280: not found.
8390 ethercard probe at 0x280 failed.
HP-LAN ethercard probe at 0x280: not found (nothing there).
No ethernet device found.
D10: D-Link pocket adapter: probe failed at 0x378.
```

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

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

Otherwise, you'll see:

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

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

```
Math coprocessor using irg13 error reporting.
```

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

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

• Finally, **mounts the root partition**. The root partition is the disk partition where the operating system resides. When "**mounts**" this partition, it is making the partition available for use by the

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

4. Unix Commands

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

```
/home/larry#
```

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

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

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

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

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

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

4.1 A Typical Unix Command

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

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

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

If you misspelled cat, you would have seen:

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

Thus, the shell informs you that it couldn't find a program named "`ct`" and gives you another prompt to work with. Remember, Unix is case sensitive: `CAT` is a misspelling. You could have also placed whitespace before the command, like this:

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

This produces the correct result and runs the cat program. You might also press **Return** on a line by itself. Go right ahead - it does absolutely nothing.

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

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

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

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

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

4.2 Helping Yourself

The **`man`** command displays reference pages for the command you spesify. For example:

```
/home/larry# man cat

cat(1)

NAME
      cat - Concatenates or displays files

SYNOPSIS
      cat [-benstuvAET] [-number] [-number-noblank] [-squeeze-blank]
      [-show-noprinting] [-show-ends] [-show-tabs] [-show-all]
      [-help] [-version] [file]

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

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

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

There's also a keyword function in **`man`**. For example, say you're interested in any commands that deal with Postscript, the printer control language from Adobe. Type `man -k ps` or `man -k Postscript`, you'll get a listing of all commands, system calls, and other documented parts of Unix that have the word "**`ps`**" (or "`Postscript`") in their name or short description. This can be very useful when you're looking for a tool to do something, but you don't know it's name - or if it even exists!

5. Storing Information

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

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

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

```
 /home/larry/joe
①  ②     ③    ④
```

① First, we are in the **root directory**.

② This signifies the directory called **home**. It is inside the root directory.

③ This is the directory **larry**, which is inside home.

④ **joe** is inside larry.

A path could refer to either a directory or a filename, so joe could be either. All the items before the short name must be directories.

An easy way of visualizing this is a tree diagram. To see a diagram of a typical system, look at next figure. Please note that this diagram isn't complete - a full system has over 7000 files!--and shows only some of the standard directories. Thus, there may be some directories in that diagram that aren't on your system, and your system almost certainly has directories not listed there.

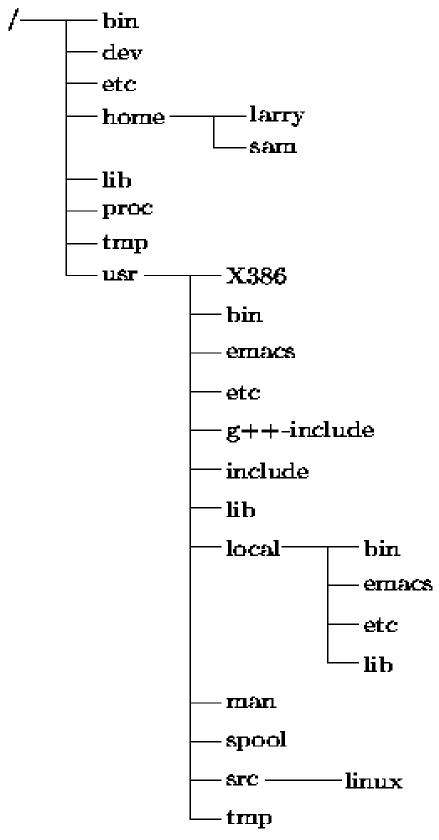
5.1 Looking at Directories with ls

Now that you know that files and directories exist, there must be some way of manipulating them. Indeed there is. The command **`ls`** is one of the more important ones. It **`l`ist`s` files**. If you try `ls` as a command, you'll see nothing.

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

Unix is intensionally terse: it gives you nothing, not even "no files" if there aren't any files. Thus, the lack of output was ls's way of saying it didn't find any files. But I just said there could be 7000 or more files lying around: where are they?

You've run into the concept of a "**current**" **directory**. You can see in your prompt that your current directory is `/home/larry`, where you don't have any files. If you want a list of files of a more active directory, try the root directory:



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

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

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

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

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

```
/home/larry# ls /usr/local
archives    bin   emacs etc   ka@q  lib   tcl
/home/larry#
```

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

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

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

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

5.2 The Current Directory and cd

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

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

As you can see, `pwd` tells you your current directory - a very simple command. Most commands act, by default, on the current directory, such as `ls`. We can change our current directory using `cd`. A generic template looks like:

```
    cd [directory]
```

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

```
/home/larry# cd /home
/home# ls -F
larry/      vasile/     shoutdown/  user1/
/home#
```

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

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

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

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

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

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

One other shortcut for lazy users: the directory `~/` is your home directory. You can see at a glance that there isn't anything in your home directory! Actually, `~/` will become more useful as we learn more about how to manipulate files.

5.3 Using mkdir to Create Your Own Directories

Creating your own directories is extremely simple under Unix, and can be a useful organizational tool.

To create a new directory, use the command `mkdir`. Of course, **mkdir** stands for **ma**ke **dir**ectory.

```
    mkdir directory
```

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

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

Finally, there is the opposite of `mkdir`, `rmdir` for remove directory. `rmdir` works exactly as you think it should work:

```
    rmdir directory
```

An example of `rmdir` is, but first we must create some directorys:

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

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

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

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

6. Moving Information

All of these fancy directories are very nice, but they really don't help unless you have some place to store you data. The Unix Gods saw this problem, and they fixed it by giving the users "files". The primary commands for manipulating files under Unix are **cp**, **mv**, and **rm**. Respectively, they stand for **copy**, **move**, and **remove**.

6.1 cp

`cp` is a very useful utility under Unix, and extremely powerful. It enables one person to copy more information in a second than a fourteenth century monk could do in a year. Be careful with `cp` if you don't have a lot of disk space. No one wants to see Error saving - disk full. `cp` can also overwrite existing files - I'll talk more about that danger later. The **first parameter** to `cp` is the **file to copy** - the **last** is **where to copy** it. You can copy to either a different filename, or a different directory. Let's try some examples:

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

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

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

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

• `cp` can copy files between directories if the first parameter is a file and the second parameter is a directory.

• It can copy a file and change it's name if both parameters are file names. Here is one danger of `cp`. If I typed `cp /etc/rc /etc/passwd`, `cp` would normally create a new file with the contents identical to `rc` and name it `passwd`. However, if `/etc/passwd` already existed, `cp` would destroy the old file without giving you a chance to save it!

• Let's look at another example of `cp`:

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

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

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

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

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

6.2 Pruning Back with rm

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

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

```
rm file1 file2 ...fileN
```

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

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

6.3 A Forklift Can Be Very Handy

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

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

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

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

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

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

7. What is The X Window System?

The X Window System is a distributed, graphical method of working developed primarily at the **Massachusetts Institute of Technology**. It has since been passed to a consortium of vendors (aptly named "The X Consortium") and is being maintained by them.

The X Window System (hereafter abbreviated as "X") has new versions every few years, called releases. As of this writing, the latest revision is X11R6, or release six. The eleven in X11 is officially the version number.

There are two terms when dealing with X that you should be familiar. The **client** is a X program. For instance, **xterm** is the client that displays your shell when you log on. The **server** is a program that provides services to the client program. For instance, the server draws the window for **xterm** and communicates with the user. Since the client and the server are two seperate programs, it is possible to run the client and the server on two physically seperate machines. This is the real beauty of X. In addition to supplying a standard method of doing graphics, you can run a program on a remote machine (across the country, if you like!) and have it display on the workstation right in front of you.

A third term you should be familiar with is the window manager. The **window manager** is a special client that tells the server where to position various windows and provides a way for the user to move these windows around. The server, by itself, does nothing for the user. It is merely there to provide a buffer between the user and the client.

7.1 Starting X

Even if X doesn't start automatically when you login, it is possible to start it from the regular text-mode shell prompt. There are two possible commands that will start X, either **startx** or **xinit**. Try startx first. If the shell complains that no such command is found, try using xinit and see if X starts. If neither command works, you may not have X installed on your system - consult local documentation for your distribution.

7.2 Exiting X

One important menu entry should be "Exit Window Manager" or "Exit X" or some like entry. Try to find that entry (remember, there could be more than one menu - try different mouse buttons!) and choose it. If X was automatically started when you logged in, this should log you out. Simply login again to return. If you started X manually, this should return you to the text mode prompt.

8. Working with Unix

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

8.1 Wildcards

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

A better way of doing that task is to type:

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

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

8.2 What Really Happens?

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

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

```
/home/larry# echo Hello!
Hello!
/home/larry# cd report
/home/larry/report# ls -F
1993-1      1994-1      data1       data5
1993-2      data-new    data2
/home/larry/report# echo 199*
1993-1      1992-2      1994-1
/home/larry/report# echo*4*
1994-1
/home/larry/report# echo*2*
1993-2      data2
/home/larry/report#
```

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

One word about that, though. Other shells, like `tcsh`, will, instead of just passing the wildcard `verbatim`, will reply **No match**.

```
mousehouse>echo /rc/fr*og
echo: No match.
mousehouse>
```

The last question you might want to know is what if I wanted to have `data*` echoed back at me, instead of the list of file names? Well, under both `bash` and `tcsh`, just include the string in quotes:

```
/home/larry/report# echo "data*"
data*
/home/larry/report#
```

or

```
mousehouse>echo "data*"
data*
mousehouse>
```

8.3 The Question Mark

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

9. The Standard Input and The Standard Output

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

9.1 Unix Concepts

The Unix operating system makes it very easy for programs to use the terminal. When a program writes something to your screen, it is using something called **standard output**. Standard output, abbreviated as **stdout**, is how the program writes things to a user. The name for what you tell a program is **standard input** (**stdin**). It's possible for a program to communicate with the user without using standard input or output, but very rare - all of the commands we have covered so far use stdin and stdout. For example, the `ls` command prints the list of the directories to standard output, which is normally "connected" to your terminal. An interactive command, such as your shell, `bash`, reads your commands from standard input.

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

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

9.2 Output Redirection

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

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

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

```
/home/larry# cat listing

. . .

/home/larry#
```

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

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

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

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

press **Ctrl-D**

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

9.3 Input Redirection

Like redirecting standard output, it is also possible to redirect standard input. Instead of a program reading from your keyboard, it will read from a file. Since input redirection is related to output redirection, it seems natural to make the special character for input redirection be <. It too, is used after the command you wish to run. This is generally useful if you have a data file and a command that expects input from standard input. Most commands also let you specify a file to operate on, so < isn't used as much in day-to-day operations as other techniques.

9.4 Solution: The Pipe

Many Unix commands produce a large amount of information. For instance, it is not uncommon for a command like `ls /usr/bin` to produce more output than you can see on your screen. In order for you to be able to see all of the information that a command `like ls /usr/bin`, it's necessary to use another Unix command, called `more`. `more` will pause once every screenful of information. For instance, `more < /etc/rc` will display the file `/etc/rc` just like `cat /etc/rc` would, except that more will let you read it. However, that doesn't help the problem that **`ls`** `/usr/bin` displays more information than you can see. `more < ls /usr/bin` won't work--input redirection only works with files, not commands! You could do this:

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

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

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

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

10. Multitasking

A technique used in an operating system for **sharing a single processor between several independent jobs**. The first multitasking operating systems were designed in the early 1960s.

Under "**cooperative multitasking**" the running task decides when to give up the CPU and under "**pre-emptive multitasking**" the scheduler suspends the currently running task after it has run for a fixed period known as a "time-slice" and (re)starts another task.

The running task may relinquish control voluntarily even in a pre-emptive system if it is waiting for some external event. In either system a task may be suspended prematurely if a hardware interrupt occurs, especially if a higher priority task was waiting for this event and has therefore become runnable.

Multitasking introduces overheads because the processor spends some time in choosing the next job to run and in saving and restoring tasks' state, but it reduces the worst-case time from job submission to completion compared with a simple batch system where each job must finish before the next one starts. Multitasking also allows the CPU to do useful work on other tasks while some tasks are waiting for some external event.

A multitasking operating system should provide some degree of protection of one task from another so that it is not possible for tasks to interact in unexpected ways such as accidentally modifying the contents of each other's memory areas.

The jobs in a multitasking system may belong to one or many users. This is distinct from parallel processing where one user runs several tasks on several processors but closely related to time-sharing where several users run several tasks on one, or many, processors.

Multithreading is a kind of multitasking with low overheads but less protection of tasks from each other.

10.1 The Basics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`yes` *is running, anf if we type* **Ctrl+Z** *now, we'll suspend it and get the prompot back. Imagine that I just didt that!*

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

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

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

```
/home/larry#
```

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

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

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

Now it's in the foreground again. Imagine thet I hit **Ctrl+C** *to terminate it*

```
/home/larry#
```

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

```
/home/larry# yes > /dev/null &
[1] 1024
/home/larry# yes | sort > /dev/null &
[2] 1026
/home/larry# yes | uniq > /dev/null
```

and here type **Ctrl+Z** *to suspend it*

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

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

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

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

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

and now hit **Ctrl+C** *to kill it*

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

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

and now hit **Ctrl+C** *to kill it*

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

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

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

```
/home/larry# fg %1
yes > /dev/null
```
now **Ctrl+Z** *to suspend it*

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

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

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

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

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

Notice that now it is running, and the other job has moved back up in line and has the +. Well, enough of that. Kill them all so you can get your machine back:

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

You should see various messages about termination of jobs - nothing dies quietly, it seems.

DOS doesn't have real job control, there is no key combination for suspending, make it run in the backgruond or for killing a job.

10.2 What Is Really Going On Here?

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

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

Let's take a look at one more command to understand what PIDs are. The `ps` command will list all running processes, including your shell. Try it out. It also has a few options, the most important of which (to many people) are `a`, `u`, and `x`. The a option will list processes belonging to any user, not just your own. The `x` switch will list processes that don't have a terminal associated with them. Finally, the **u** switch will give additionally information about the process that is frequently useful.

To really get an idea of what your system is doing, put them all together: `ps -aux`. You can then see the process that uses the more memory by looking at the `%MEM` column, and the most CPU by looking at the `%CPU` column. (The TIME column lists the total amount of CPU time used.)

Another quick note about PIDs. kill, in addition to taking options of the form `%job#`, will take options of raw PIDs. So, put a `yes > /dev/null` in the background, run `ps`, and look for `yes`. Then type `kill` PID.

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

11. Virtual Consoles

Linux supports **virtual consoles**. These are a way of making your single machine seem like **multiple terminals**, all connected to one Linux kernel. Thankfully, using virtual consoles is one of the simplest things about Linux: there are "hot keys" for switching among the consoles quickly. To try it, log in to your Linux system, hold down the left **Alt** key, and press **F2** (that is, the function key number 2).

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

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

Once you get used to them, VC's will probably become an indispensable tool for getting many things done at once. For example, I typically run `emacs` on **VC1** (and do most of my work there), while having a communications program up on **VC3** (so I can be downloading or uploading files by modem while I work, or running jobs on remote machines), and keep a shell up on **VC2** just in case I want to run something else without tying up **VC1.**

12. The Power of Unix

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

Let's cover what each of these utilities do seperately and then I'll give some examples of how to use them together.

12.1 Operating on Files

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

Some of the things these commands manipulate:

- The **time** stamptime stamp. Each file has three dates associated with it. The three dates are the creation time (when the file was created), the last modification time (when the file was last changed), and the last access time (when the file was last read).
- The **owner**. Every file in Unix is owned by one user or the other.
- The **group**. Every file also has a group of users it is associated with. The most common group for user files is called users, which is usually shared by all the user account on the system.
- The **permissions**. Every file has permissions (sometimes called "privileges") associated with it which tell Unix who can access what file, or change it, or, in the case of programs, execute it. Each of these permissions can be toggled seperately for the owner, the group, and all other users.

```
touch  file1 file2 ...fileN
```

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

```
chmod  [-Rfv] mode file1 file2 ...fileN
```

The command used to change the permissions on a file is called `chmod`, short for `ch`**ange** `mod`**e**. Before I go into how to use the command, let's discuss what permissions are in Unix. Each file has a **group of permissions** associated with it. These permissions tell Unix whether or not the file can be read from, written to, or executed as a program. (In the next few paragraphs, I'll talk about users doing these things. Naturally, any programs a user runs are allowed to do the same things a user is. This can be a security problem if you don't know what a particular program does.). However, Unix recognizes **three different people**: first, the **owner** of the file (and the person allowed to use `chmod` on that file). The **group** of most of your files might be "users", meaning the **normal users** of the system. (To find out the group of a particular file, use `ls -l file`.) Then, there's **everybody** else who isn't the owner and isn't a member of the group.

So, a file could have read and write permissions for the owner, read permissions for the group, and no permissions for all others. Or, for some reason, a file could have read/write permissions for the group and others, but no permissions for the owner!

Let's try using `chmod` to change a few permissions. First, create a new file using `cat`, `emacs`, or any other program. By default, you'll be able to read and write this file. (The permissions given other people will vary depending on how the system and your account is setup.) Make sure you can read the file using `cat`. Now, let's take away your read privilege by using `chmod u-r filename`. (The parameter `u-r` decodes to "user minus read".) Now if you try to read the file, you get a Permission denied error! Add read privileges back by using `chmod u+r filename`.

Directory permissions use the same three ideas: read, write, and execute, but act slightly differently. The read privilege allows the user (or group or others) to read the directory - list the names of the files. The write permission allows the user (or group or others) to add or remove files. The execute permission allows the user to access files in the directory or any subdirectories. (If a user doesn't have execute permissions for a directory, they can't even `cd` to it!)

To use `chmod`, replace the mode with what to operate on, either user, group, other, or all, and what to do with them. (That is, use a plus sign to indicate adding a privilege or a minus sign to indicate taking one away. Or, an equals sign will specify the exact permissions.) The possible permissions to add are read, write, and execute.

chmod's `R` flag will change a directory's permissions, and all files in that directory, and all subdirecties, all the way down the line. (The 'R' stands for recursive.) The `f` flag forces `chmod` to attempt to change permissions, even if the user isn't the owner of the file. (If `chmod` is given the `f` flag, it won't print an error message when it fails to change a file's permissions.) The `v` flag makes `chmod` verbose - it will report on what it's done.

12.2 System Statistics

Commands in this section will display statistics about the operating system, or a part of the operating system.

```
du  [-abs] [path1 path2 ...pathN]
```

`du` stands for d**isk** u**sage**. It will count the amount of disk space a given directory and all its subdirectories take up on the disk. `du` by itself will return a list of how much space every subdirectory of the current directory consumes, and, at the very bottom, how much space the current directory (plus all the previously counted subdirectories) use. If you give it an option parameter or two, it will count the amount of space used by those files or directories instead of the current one.

The `a` flag will display a count for files, as well as directories. An option of `b` will display, instead of kilobytes (1024 characters), the total in bytes. One byte is the equivalent of one letter in a text document. And the s flag will just display the directories mentioned on the command-line and not their subdirectories.

```
df
```

`df` is short for something, although this author isn't quite sure what. `df` summarizes the amount of disk space in use. For each filesystem (remember, different filesystems are either on different drives or partitions) it shows the total amount of disk space, the amount used, the amount available, and the total capacity of the filesystem that's used.

One odd thing you might encounter is that it's possible for the capacity to go over 100%, or the used plus the available not to equal the total. This is because Unix reserves some space on each filesystem only for root. That way, if a user accidentally fills the disk, the system will still have a little room to keep on operating.

For most people, `df` doesn't have any useful options.

```
uptime
```

The uptime program does exactly what one would suspect. It prints the amount of time the system has been "up" - the amount of time from the last Unix boot.

`uptime` also gives the current time and the load average. The load average is the average number of jobs waiting to run in a certain time period. uptime displays the load average for the last minute, five minutes, and ten minutes. A load average near zero indicates the system has been relatively idle. A load average near one indicates that the system has been almost fully utilized but nowhere near overtaxed. High load averages are the result of several programs being run simultaneously.

Amazingly, `uptime` is one of the few Unix programs that have no options!

```
who
```

`who` displays the current users of the system and when they logged in. If given the parameters `am i` (as in: `who am i`), it displays the current user.

```
w  [-f] [username]
```

The `w` program displays the current users of the system and what they're doing. (It basically combines the functionality of `uptime` and `who`. The header of `w` is exactly the same as uptime, and each line shows a user, when the logged on (and how long they've been idle). JCPU is the total amount of CPU time used by that user, while PCPU the the total amount of CPU time used by their present task.

If `w` is given the option `f`, it shows the remote system they logged in from, if any. The optional parameter restricts `w` to showing only the named user.

12.3 What's in the File?

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

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

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

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

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

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

- **Ctrl** + **L**: Moves to the next screen of text.
- **Space**: This will scroll the screen by 11 lines, or about half a normal, 25-line, screen.
- d: Searches for a regular expression. While a regular expression can be quite complicated, you can just type in a text string to search for. For example, `/toad` would search for the next occurence of "`toad`" in your current file. A slash followed by a return will search for the next occurence of what you last searched for.

- `/` This will also search for the next occurence of your regular expression.
- `n/` If you specified more than one file on the command line, this will move to the next file.
- `n:` This will move the the previous file.

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

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

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

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

`file [file1 file2 ...fileN]`

The `file` command attempts to identify what format a particular file is written in. Since not all files have extentions or other easy to identify marks, the file command performs some rudimentary checks to try and figure out exactly what it contains.

Be careful, though, because it is quite possible for `file` to make a wrong identification.

12.4 Information Commands

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

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

One of the most useful commands in Unix is `grep`, the generalized regular expression parser. This is a fancy name for a utility which can only search a text file. The easiest way to use `grep` is like this:

```
/home/larry# cat animals
Some of the animals are nice creatures
My favorite one is the cat
/home/larry# grep ca animals
My favorite one is the cat
/home/larry#
```

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

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

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

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

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

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

The three parameters, `clw`, stand for character, line, and word respectively, and tell `wc` which of the three to count. Thus, `wc -cw` will count the number of characters and words, but not the number of lines. wc defaults to counting everything - words, lines, and characters.

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

`spell [file1 file2 ...fileN]`

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

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

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

`cmp file1 [file2]`

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

`diff file1 file2`

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

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

```
/home/larry# cat frog
Animals are interesting creatures.
One of my favorite animal is the cat.
I also like mice.
/home/larry# cp frog toad
/home/larry# diff frog toad
Animals are interesting creatures.

One of my favorite animal is the cat.
I also like mice.
/home/larry# diff frog dog
1c1,2
< Animals are interesting creatures.
---
> Animals are interesting creatures.
```

```
>
3c4
< I also like mice.
---
> I also like mice.
/home/larry#
```

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

13. bash Customization

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

This is mainly done through configuration files. These are also known as "init files", "rc files" (for "run control"), or even "dot files", because the filenames often begin with ".". If you'll recall, filenames that start with "." aren't normally displayed by `ls`.

The most important configuration files are the ones used by the shell. Linux's default shell is bash, and that's the shell this chapter covers. Before we go into how to customize bash, we should know what files bash looks at.

13.1 Shell Startup

There are several different ways bash can run. It can run as a login shell, which is how it runs when you first login. The login shell should be the first shell you see.

Another way bash can run is as an interactive shell. This is any shell which presents a prompt to a human and waits for input. A login shell is also an interactive shell. A way you can get a non-login interactive shell is, say, a shell inside `xterm`. Any shell that was created by some other way besides logging in is a non-login shell.

Finally, there are non-interactive shells. These shells are used for executing a file of commands, much like MS-DOS 's batch files - the files that end in .BAT. These shell scripts function like mini-programs. While they are usually much slower than a regular compiled program, it is often true that they're easier to write.

Type of shell	Action
Interactive login	The file `.bash_profile` is read and executed
Interactive	The file `.bashrc` is read and executed
Non-Interactive	The shell script is read and executed

13.2 Startup Files

Since most users want to have largely the same environment no matter what type of interactive shell they wind up with, whether or not it's a login shell, we'll start our configuration by putting a very simple command into our `.bash_profile`: `"source~/.bashrc"`. The `source` command tells the shell to interprete the argument as a shell script. What it means for us is that everytime `.bash_profile` is run, `.bashrc` is also run.

Now, we'll just add commands to our `.bashrc`. If you ever want a command to only be run when you login, add it to your `.bash_profile`.

13.3 Aliasing

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

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

That command defined a shell alias called `ll` that "expands" to the normal shell command `"ls -l"` when invoked by the user. So, assuming that bash has read that command in from your `.bashrc`, you can just type ll to get the effect of `"ls -l"` in only half the keystrokes. What happens is that when you type `ll` and hit **Enter**, bash intercepts it, because it's watching for aliases, replaces it with `"ls -l"`, and runs that instead. There is no actual program called ll on the system, but the shell automatically translated the alias into a valid program. Some sample aliases now presented in the following examples. You could put them in your own `.bashrc`. One especially interesting alias is the first one. With that alias, whenever someone types `ls`, they automatically have a `-F` flag tacked on. (The alias doesn't try to expand itself again.) This is a common way of adding options that you use every time you call a program.

```
alias ls="ls -F"         #gives characters at the end of listing
alias ll="ls -l"               #special ls
alias la="ls -a"
alias ro="rm *~; rm .*~"       removes backup files created by Emacs
alias rd="rmdir"
alias md="mkdir"
alias pu=pushd           #pushd, popd, and dirs weren't covered in this
alias po=popd                  # course - you might want to look them up in
alias ds=dirs                  #the bash manual
#these are just keyboard shortcuts
alias to="telnet tempus.east.utcluj.ro"
```

Notice the comments with the # character in the commands. Whenever a # appears, the shell ignores the rest of the line. You might have noticed a few odd things about them. First of all, I leave off the quotes in a few of the aliases - like pu. Strictly speaking, quotes aren't necessary when you only have one word on the right of the equal sign. It never hurts to have quotes either, so don't let me get you into any bad habits. You should certainly use them if you're going to be aliasing a command with options and/or arguments:

13.4 Environment Variables

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

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

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

Here's how you can change your environment in bash :

```
/home/larry# export PGPPATH=/home/larry/secrets/pgp
```

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

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

It is also useful to be able to query the environment:

```
/home/larry# echo $PGPPATH
/home/larry/.pgp
/home/larry#
```

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

```
/home/larry# echo PGPPATH
PGPPATH
/home/larry#
```

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

There's another command that's very useful for querying the environment: `env` . `env` will merely list all the environment variables. It's possible, especially if you're using X, that the list will scroll off the screen. If that happens, just pipe `env` through `more`: `env |more`.

A few of these variables can be fairly useful, so I'll cover them. Look at the next table. Those four variables are defined automatically when you login: you don't set them in your `.bashrc` or `.bash_login`.

Variable name	Contains	Example
`HOME`	Your home directory	`/home/larry`
`TERM`	Your terminal type	`xterm, vt100 or console`
`SHELL`	The path to your shell	`/bin/bash`

`USER`	Your login name	`larry`
`PATH`	A list to search for programs	`/bin:/usr/bin:/usr/local/bin:/usr/bin/X11`

Let's take a closer look at the `TERM` variable. To understand that one, let's look back into the history of Unix: The operating system needs to know certain facts about your console, in order to perform basic functions like writing a character to the screen, moving the cursor to the next line, etc. In the early days of computing, manufacturers were constantly adding new features to their terminals: first reverse-video, then maybe European character sets, eventually even primitive drawing functions (remember, these were the days before windowing systems and mice). However, all of these new functions represented a problem to programmers: how could they know what a terminal supported and didn't support? And how could they support new features without making old terminals worthless?

In Unix, the answer to these questions was `/etc/termcap`. `/etc/termcap` is a list of all of the terminals that your system knows about, and how they control the cursor. If a system administrator got a new terminal, all they'd have to do is add an entry for that terminal into `/etc/termcap` instead of rebuilding all of Unix. Sometimes, it's even simplier. Along the way, Digital Equipment Corporation 's vt100 terminal became a pseudo-standard, and many new terminals were built so that they could emulate it, or behave as if they were a vt100.

Under, `TERM's` value is sometimes `console`, which is a vt100-like terminal with some extra features.

Another variable, `PATH`, is also crucial to the proper functioning of the shell. Here's mine:

```
/home/larry# env | grep ^PATH
PATH=/home/larry/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11
/home/larry#
```

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

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

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

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

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

```
/home/larry# echo $PATH
.:/home/larry/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11
/home/larry#
```

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

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

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

```
export PATH=${PATH}:.:${HOME}/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11
/home/larry#
```

Here, I am taking advantage of the fact that the `HOME` variable is set before `bash` reads my `.bashrc`, by using its value in setting my `PATH`. The curly braces ("`{...}`") are a further level of quoting; they delimit the extent of what the "`$`" is to evaluate, so that the shell doesn't get confused by the text immediately following it ("`/bin`" in this case). Here is another example of the effect they have:

```
/home/larry# echo ${HOME}foo
/home/larryfoo
/home/larry#
```

Without the curly braces, I would get nothing, since there is no environment variables named `HOMEfoo`.

```
/home/larry# echo $HOMEfoo

/home/larry#
```

Let me clear one other thing up in that path: the meaning of "`$PATH`". What that does is includes the value of any `PATH` variable previously set in my new PATH. Where would the old variable be set? The file `/etc/profile` serves as a kind of global `.bash_profile` that is common to all users. Having one centralized file like that makes it easier for the system administrator to add a new directory to everyone's `PATH` or something, without them all having to do it individually. If you include the old path in your new path, you won't lose any directories that the system already setup for you.

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

```
export PS1='$PWD# '
```

As you can see, there are actually two variables being used here. The one being set is `PS1`, and it is being set to the value of `PWD`, which can be thought of as either "Print Working Directory" or "Path to

Working Directory". But the evaluation of `PWD` takes place inside single quotes. The single quotes serve to evaluate the expression inside them, which itself evaluates the variable `PWD`. If you just did export `PS1=$PWD`, your prompt would constantly display the path to the current directory at the time that `PS1` was set, instead of constantly updating it as you change directories. Well, that's sort of confusing, and not really all that important. Just keep in mind that you need the quotes if you want the current directory displayed in your prompt.

You might prefer export `PS1='$PWD>'`, or even the name of your system: `` export PS1=`hostname`'>' ``. Let me dissect that last example a little further.

That last example used a *new* type of quoting, the back quotes. These don't protect something - in fact, you'll notice that "`hostname`" doesn't appear anywhere in the prompt when you run that. What actually happens is that the command inside the backquotes gets evaluated, and the output is put in place of the backquotes and the command name.

Try `` echo `ls` `` or `` wc `ls` ``. As you get more experienced using the shell, this technique gets more and more powerful.

There's a lot more to configuring your `.bashrc`, and not enough room to explain it here. You can read the bash man page for more, or ask questions of experienced bash users.