Chapter 1A, Foundations of Linux

John M. Morrison

January 4, 2022

Contents

0 Introduction

You are probably used to running a computer with a graphical user interface (GUI) that gives you a desktop and icons to work with using your mouse and keyboard.

You likely run Windoze, MacOSX, or you may be running a Linux GUI on your machine. The GUI allows you to communicate with the operating system, which is the master program of your computer that manages such things as running applications and maintaining your file system.

In this book, we will study the Linux operating system in its command-line guise. You will control the computer by entering commands into a text window called a *terminal window*. Typically they look something like this.

The name "terminal" name harkens back to the days when you had an actual appliance on your desk consisting of a (heavy) CRT screen and keyboard that was connected to a computer elsewhere. Your terminal window is just a software version of this old appliance. You will enter commands into this window to get the remote machine you are communicating with to perform various tasks.

The text string `morrison@odonata` appearing in the window is called the *prompt*. Its presence indicates that the computer is waiting for you type in a command. Your prompt will likely be different.

1 In the Beginning . . .

As we progress, everything will seem unfamiliar, but actually relate very directly to some very familiar things you have seen working with a computer having a GUI. We assume you have basic proficiency using some kind of computer such as a Mac or a Windoze box, and we will relate the things you do in Linux to those you do in your usual operating system.

Log in to your UNIX account. If you are working in a Linux GUI, or a Mac, just open a terminal session. The first thing you will see after any password challenge will resemble this

```
[yourUserName}@hostName yourUserName}]$
```

or this

```
[yourUserName}@hostName ~]$
```

On a Mac, it will resemble this

```
John-Morrisons-MacBook-Pro:~ morrison$
```

The presence of the prompt means that Linux is waiting for you to enter a command. The token `yourUserName` will show your login name. Your prompt may have an appearance different from the ones shown here; this depends on how your system administrator sets up your host or on the type of Linux distribution you are using. The appearance of your prompt does not affect the underlying performance of UNIX. In fact, the properties of your session are highly configurable, and you can customize your prompt to have any appearance you wish.

To keep things simple and uniform throughout the book, we will always use `unix>` to represent the UNIX prompt. You will interact with the operating system by entering commands, instead of using a mouse to push buttons or click in windows.

When you see this terminal, a program called a *shell* is running. The shell takes the commands you type and passes them on to the operating system (kernel) for action. You will type a command, then hit the ENTER key; this causes the command to be shipped to the OS by the shell. The shell then conveys the operating system's reply to your terminal screen. Think of the shell as a telephone through which you communicate with the operating system. This analogy is only fitting since UNIX was originally developed at AT&T Bell Labs.

We will begin by learning aboout the structure of a UNIX command. These commands willgive you access to your programs and data, so what is to follow is very fundamental.

Before we start there are three imporant items to know about.

1. `stdin` This is *standard input*, which by default is the keyboard.
2. `stdout` This is *standard output*, which by default is the terminal screen.
3. `stderr` This is *standard error*, which by default is the also the terminal screen.

As we progress, you will see that UNIX thinks of these things as being files. In fact it thinks *everything* is a file. You will also learn that these things can be redirected into files and other places.

2 Anatomy of a Command

Every UNIX command has three parts: a name, options, and zero or more arguments. They all have the following appearance

```
commandName -option(s) argument(s)
```

Notice how the options are preceded by a -. Certain "long–form" options are preceded by a --. In a Mac terminal, all options are preceded by a simple -.

A command always has a name. Grammatically you should think of a command as an imperative sentence. For example, the command `passwd` means, "Change my password!" You can type this command at the prompt, hit enter, and follow the instructions to change your password any time you wish.

Arguments are supplementary information that is sometimes required and sometimes optional, depending on the command. Grammatically, arguments are nouns: they are things that the command acts upon.

Options are always, well, ... optional. Options modify the basic action of a command and they behave grammatically as adverbs. All familiar features of a graphics-based machine are present in Linux, you will just invoke them with a text command instead of a mouse click. We will go through some examples so you get familiar with all the parts of a Linux command.

Two very basic Linux commands are `whoami` and `hostname`. Here is a typical response. These commands give, respectively, your user name and the name of the host you log in to.

Now we run them. We show the results here; your computer will show your login name and your host name, planceiing them into `stdout`. Here is what they look like on a server.

```
unix> whoami
morrison
unix> hostname
carbon.ncssm.edu
unix>
```

Here is their appearance on a PC (A Mac in this instance).

```
unix> whoami
morrison
unix> hostname
John-Morrisons-MacBook-Pro.local
unix>
```

We will next turn to the organization of the file system.

3 Managing Directories

We will do a top–down exploration of the file system. In this spirit, we will first learn how to manage directories; this is the UNIX name for folders. Directories

are actually files: they contain links to the items "inside" of them. You will want to know how to create and manage folders, ummmm... directories , and how to navigate through them.

You have been in a directory all along without knowing it. Whenever you start a UNIX session, you begin in your *home directory*. Every user on a UNIX system owns a home directory. This is where you will keep all of your stuff. You will see that ownership of stuff is baked right into a UNIX system, and you will learn how to control what others see of your stuff.

To see your home directory, type `pwd` at the UNIX prompt. This command means, "Print working directory!" You will see something like this.

```
unix> pwd
/home/faculty/morrison
```

This directory is your home directory. Whenever you start a new session, you will begin here.

In this example,`morrison` is a directory inside of `faculty`, which is inside a directory `home`, which is inside the *root directory*, `/`. Your home directory will likely be slightly different. It is very common for UNIX systems to keep all user directories inside of a directory named `home`. Often, several different types of users are organized into sub-directories of `home`. Enter the `pwd` command on your machine and compare the result to what was shown here. Become familiar with your home directory's appearance so you can follow what goes on in the rest of this chapter.

If you are using Linux on your PC, your home directory will likely look like this.

```
/home/morrison
```

On a Mac it looks like this

```
/Users/morrison
```

This directory structure is exactly the same as your hierarchy of folders and files on a Mac or a Windoze box. You already know that folders can contain files and other folders. This is also true in a UNIX environment.

To make a new directory in Mac or Windoze, you right click in the open folder and choose a menu for making a new folder. In UNIX, the `mkdir` command makes a one or more new directories. The `mkidr` command requires at least one argument, the name(s) of the director(ies) you are creating. Let us make a directory by typing

```
unix> mkdir Projects
```

makes a directory called Projects; this directory is now empty. We can always get rid of an empty directory or directories by typing the `rmdir` command like so.

```
unix> rmdir garbageDirectory(ies)
```

In this case, `garbageDirector(ies)` stands for the directory or directories you wish removed.

The `rmdir` command will not remove a directory unless it is empty. There is a way to snip off directories with their contents, but we will avoid it for now because it is very dangerous. For now, you can delete the contents of a directory, then remove the directory. Be warned as you proceed: *When you remove files or directories in Linux, they are gone for good! There is no "undelete."*

If you got rid of the `Projects` directory, re-create it with `mkdir`. To get into our new directory Projects, enter this command.

```
unix> cd Projects
```

and type `ls`. You will see no files. This is because the directory `Projects` is empty, and `ls` by default only shows you the files in the directory you are currently occupying. The command `cd` means, "Change directory!" Having done this now type

```
unix> pwd
```

You will see a directory path now ending in `Projects`.

There is a command called `touch` which will create an empty file(s) with a name(s) you specify. Create files named `moo` and `baa` with `touch` as follows.

```
unix> touch moo baa
```

Then enter `ls` at the command line. This command means "list stuff." You will see just the files you created.

As we said before, The command `ls` displays only files in the directory you are currently occupying. This directory is called your *current working directory*, or `cwd` for short. Every terminal session has a working directory. When you first log in, your working directory is always your home directory.

```
/home/yourUserName/Projects
```

This directory is the `Projects` directory you just created.

If you type `cd` without arguments, you will go straight back to your home directory. This should make you will feel like Dorothy going back to Kansas. Now if we use `pwd` again we see our home directory printed out.

You can also see your home directory anywhere you are by typing

```
unix> echo $HOME
```

The fearsome–looking object `$HOME` is just a symbol that points to your home directory. There are various items like this present in your system. They are called *environment variables*. Other examples of environment variables include `$PWD`, which is just your current working directory and `$OLDPWD` which is your previous working directory.

Programming Exercises

1. Navigate to a directory. Then enter this.

   ```
   unix> pushd
   ```

   Then navigate to another directory and repeat this a few times. Now alternately type

   ```
   unix> popd
   unix> pwd
   ```

What does this do? Think of Hansel and Gretel!

2. Crawl around in your directory structure. Each time you enter a new directory type

```
unix> echo  $PWD
unix> echo  $OLDPWD
```

3. Use `cd` to change into some directory. Then type `cd -` and then `pwd`. Repeat this. What does `-` mean?

3.1 Processes and Directories

We know that when we log in, we are starting a program called a *shell*. The shell is a process, or running program. Every process has a `cwd` (current working directory). When you type `pwd` into your shell, you are asking the OS to tell you your shell's current working directory. If you log in to a UNIX server in several terminal windows, each runs in a separate shell, so each has can have its own working directory.

Observe that, much of the time, your shell is idle. When you finish typing a command and hit the enter key, that command launches a program, that program runs, and any output is directed to your terminal window.

The command `cd` is a computer program. What it does is it changes the `cwd` of the shell that calls it. Now you know what it means to be "in" a directory: it means the `cwd` of your shell is that directory.

Programming Exercises

1. Enter

   ```
   unix> cd $HOME/Projects
   ```

   and see what happens.
2. Make these directories inside of Projects `labors`, `feats` and `chores`
3. Type `cd labors` at the command line then `pwd`.
4. Type `cd ..` at the command line then `pwd`. What happened?
5. Type `cd ..` at the command line again, then `pwd`. What happened?
6. What do you think `..` is?
7. Type `cd .` at the command line then `pwd`. What happened?
8. Type `ls .` at the command line then `pwd`. What happened?
9. What do you think `.` is?

4 Paths

The location of your home directory is specified by a *path* that looks something like this `/home/morrison`. This path is an example of an *absolute path*, because it specifies a location in the file system starting at the root directory.

All absolute paths start with a `/` or a `~`. Here are the three kinds of absolute paths.

- Paths beginning with a `/` are specified starting at the root directory.
- The symbol `~` is shorthand for your home directory. It is an absolute path. Try going anywhere in the file system and type `cd ~`; it will take you straight home, just as `cd` does by itself.
- A path beginning with `~someUserName` specifies the home directory of the user `someUserName`.

Absolute paths work exactly the same, no matter where you are in the file system.

Relative paths are relative to your `cwd`. Every directory contains an entry for its parent and itself. Make an empty directory named `ghostTown` and do an `ls -a`.

```
unix> mkdir ghostTown
unix> cd ghostTown/
unix> ls -a
.  ..
```

If you type `cd ..`, you are taken to the parent directory of your `cwd`; this path is relative to your `cwd`. Any path that is not absolute is relative. When you are navigating in your home directory, you are mostly using relative paths. Note that any relative path can also be represented as an absolute path.

Programming Exercises

1. Try typing `cd ..` then `pwd` a few times. What happens?
2. Type `cd .` Where do you go?
3. Type `cd /bin` (on a mac `/usr/bin`) then `ls cd*` You will see a file named `cd` that lives in that directory.

5 A Field Trip

To get to our first destination, type `cd /`. The directory `/` is the "root" directory; it is an absolute path. If you think of the directory structure as an upside-down

(Australian) tree (root at top), the directory / is at the top. Type `pwd` and see where you are. Type `ls`; you should see that the directory `home` listed with several other directories. Here is what the directory structure looks like on a PC running Red Hat Fedora Core 9. Yours may have a slightly different appearance.

```
unix> cd /
unix> ls
bin    etc            lib         mnt   root    srv    usr
boot   home           lib64       opt   run     sys    var
cdrom  initrd.img     lost+found  proc  sbin    tmp    vmlinuz.old
dev    initrd.img.old media       root  selinux vmlinuz
unix>
```

Now type if we type `cd home` then `ls`, you will see one or more directories. On the machine being used here, you would see

```
unix> cd home
unix> ls
guest lost+found morrison
```

This machine has two users, `morrison` and `guest`. Since it is a personal computer, it does not have many users. You may be working on a server in which there could be dozens, or even hundreds of other users who are organized into various directories.

Here is an example from a fairly busy server.

```
unix> cd /home
unix> ls
2016  2018  2020  gotwals          menchini  rash
2017  2019  cs    keethan.kleiner  morrison  rex.jeffries
unix>
```

The directories with the years are directories full of user's home directories. We will list one here. It has quite a few users in it.

```
unix> ls 2019
allen19m   hablutzel19k  laney19m     mullane19n                wang19e
bounds19a  hirsch19m     lheem19h     ou19j                     wolff19o
carter19d  hou19b        lin19b       overpeck19c               yang19j
cini19a    houston19b    liu19c       perrin19p                 zhuang19a
eun19e     houston19p    manocha19a   sakarvadia19m
gupta19a   knapp19t      mitchell19m  villalpando-hernandez19j
unix>
```

See if you can follow this all the way down to another user's home directory. You may be able to list the files there, or even read them, depending on that user's permissions. From this modest demonstration, you see that you can step down through the directory structure using cd. Now we will learn how to step up.

Try typing `cd ..`; the special symbol `..` represents the directory above your `cwd`. Now you can climb up and down the directory structure! The `..` symbol works like the up-arrow in a file chooser dialog box in Mac or Windoze. You saw this when you did the last group of exercises.

Practice this; go back to your home directory. Make a new directory called `mudpies`. Put some files in it. Make new directories in `mudpies`, got down inside these and make more directories and files. Practice using `cd` to navigate the tree you create. When you are done, get rid of the whole mess; remember you have to go to the bottom, empty out the files using `rm` and then use `rmdir` to get rid of the empty directories.

If you type `ls` in a directory, notice how any directories inside it are in differently colored type than regular files. This color is often blue. You can use the `-F` option in `ls` to print directory names with a slash (/) after them. Try this; it was an important option back in the days of monochrome monitors. If you use the `-l` option in `ls`, you will see that in the *permissions column*, the column begins with a `d` for any directory. Here is a possible sample

```
-rw-rw-r--    1 morrison morrison        0 Jun  9 14:54 bar
-rw-rw-r--    1 morrison morrison        0 Jun  9 14:54 foo
drwxrwxr-x    2 morrison morrison     4096 Jun  9 14:54 junk
```

You can see there that `bar` and `foo` are empty files. Notice the `d` at the beginning of the line in `junk`; this tells you `junk` is a directory.

6 Making and Listing Regular Files

The operating system is responsible for maintaining the file system. The file system maintained by a UNIX system consists of a hierarchy of files. Two types of files will be of interest to us: *directories* (folders) and *regular files*, i.e. files that are not directories. Regular files may hold data or programs. They may consist of text or be binary files that are not human-readable.

You are used to working with regular files and directories in Windoze or MacOSX. Things in UNIX work the same way, but we will use commands to manage files instead of mouse clicking or dragging.

As we have already seen us now use the UNIX command `touch` to create new files. This command creates an empty file for each argument given it. At your UNIX prompt, enter

```
unix> touch stuff
```

This creates the empty file named `stuff` in your account.

Now let us analyze the anatomy of this command. The name of the command is `touch`; its purpose is to create an empty file. Since you do not see a - sign, there are no options being used. The argument is `stuff`. This is the name of the file you created. Create a few more empty files. Enter these commands

```
unix> touch foo
unix> touch bar
```

You may create several files at once by making a space-separated list as we show here.

```
unix> touch aardvark buffalo cougar dingo elephant
```

Now you have eight new files in your account. Next we will see how to list the files. Enter this command at the UNIX prompt

```
unix> ls
```

The command `ls` lists your files. Notice we had neither options nor arguments. If you created the files using `touch` as instructed, they should appear on your screen like this

```
unix> aardvark bar buffalo cougar dingo elephant foo stuff
```

The command `ls` has several options. One option is the `l` option; it list the files in long format. To invoke it, type

```
unix> ls -l
```

You will see a listing like this

```
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 aardvark
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 bar
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 buffalo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 cougar
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 dingo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 elephant
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:49 foo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:49 stuff
```

The first column reflects the *permissions* for the files. The sequence

```
-rw-rw-r--
```

indicates that you and your group have read/write permission and that others ("the world") has read permission. We will discuss permissions in more detail when we discuss the management of directories.

You can see the name here is listed in two columns; on this machine `morrison` is in his own group. On another system, you may live in group with several other people; if so you will see the name of that group in one of these columns. The zero indicates the size of the file; each file we created is empty. Then there is a date, a time and the file name. This is the long format for file listing; it is seen by using the `-l` option in the `ls` command.

Another option is the `-a` option. This lists all files, including "hidden" files. Hidden files have a dot (.) preceding their name. To see them, enter

```
unix> ls -a
```

at the command line. One thing you are guaranteed to see, no matter where you are are are the directories `..` (parent) and `.` (current). If you are in you home directory, You will see the files you saw using `ls` and several hidden files with mysterious names like `bash_profile`. Do not delete these; they provide the configuration for your account and do things like record preferences for applications you have used or installed. You can also list all files including hidden files by entering

```
unix> ls --all
```

You can use more than one option at once. For example, entering

```
unix> ls -al
```

or

```
unix> ls -a -l
```

```
unix> ls --all -l
```

shows all of your files and hidden files in long format. Try this now on your machine.

Note to Mac Users Mac users should precede verbose commands with a single `-`. So on a Mac, you type

```
unix> ls -all -l
```

and not

```
unix> ls --all -l
```

Otherwise, your Mac will respond with a cryptic error message.

Next we will show how to display a file to the screen. UNIX commands that process files are called *filters*. Filters accept input from a file, and place output in another file. The default action of any UNIX filter is to get input from `stdin` and send its output to `stdout`. Any errors resulting from using the filter go to `stderr`, which by default is just `stdout`. If an argument is offered to a UNIX filter, that argument is generally a regular file, and it gets used in lieu of `stdin`.

Let us peek inside your `.bash_profile` file. Enter the command

```
unix> cat .bash_profile
```

The command name is `cat`, short for catalog (the file to the screen). The `cat` command is a filter that does not filtering at all; it simply dumps the entire file to the screen all at once. We are using no options, but the file name is an argument to `cat`. If a file is long and you want to see it one screenful at a time, use the filter `more`. The command `more` takes a file name as an argument and shows it on the screen a screenful at a time. You can hit the space bar to see the next screenful or use the down-arrow or enter key to scroll down one line at a time. To exit `more` at any time, type a `q` and `more` quits. You can use several arguments in `cat` or `more` and the indicated files will be displayed *in seriatum*.

Programming Exercises

1. What does the `-r` option cause `ls` to do?
2. What does the `-t` option cause `ls` to do? Try combining it with thte `-l` option and it becomes clear.
3. What does the `-n` option do to `cat`?

6.1 Globbing

Make a directory called `tossme` (you will toss this when done) and enter it by doing the following

```
unix> mkdir tossme
unix> cd tossme
```

Now create some empty files with `touch` like so.


```
MAC:Tue Jan 04:10:24:tossme> touch cat.html dog.html snake.css snake.py cow.c
MAC:Tue Jan 04:10:26:tossme> touch Turtle.java Egret.java tapir.py goat.cppw
MAC:Tue Jan 04:10:27:tossme> ls
Egret.java  cat.html    dog.html    snake.css   tapir.py
Turtle.java cow.c       goat.cpp    snake.py
```

The special character `*` is a wildcard standing for any sequence of zero or more characters. Using `*` to create wildcards is caalled "globbing" because it allows you to refer to a glob of files. Let us use it to list all files with the extansion `.py`.

```
MAC:Tue Jan 04:10:27:tossme> ls *.py
snake.py tapir.py
```

Now let's list all files taht begin with the letter `s`.

```
unix> ls s*
snake.css snake.py
```

How do we list all files whose extension begins with a `c`?

```
unix> ls *.c*
cow.c     goat.cpp  snake.css
```

You can `cat` a glob of files; for example.

```
cat *.py
```

will put all files to `stdout` with a `.py` extension.

7 Renaming and Deleting

Three commands every beginner should know are: `cp`,`rm` and `mv`. These are, respectively, copy, remove and move(rename). Here are their usages

```
cp oldFile newFile
rm garbageFile(s)
mv oldFile newFile
```

Warning! Pay heed before you proceed! To *clobber* a file means to unlink it from your file system. When you clobber a file it is lost and there is virtually no chance you will recover its contents. There is no undelete facility as you might find on other computing systems you have used.

If you remove a file it is clobbered, and there is no way to get it back without an infinitude of horrid hassle. If you copy or rename onto an existing file, that file is clobbered, and it is gone forever. Always check to see if the file name you are copying or moving to is unoccupied! When in doubt, do an `ls` to look before you leap. All three of these commands have an option `-i`, which warns you before clobbering a file. Using this is a smart precaution.

The `cp` command copies *oldFile* to *newFile*. If *newFile* does not exist, it creates *newFile*; otherwise it will overwrite any existing *newFile*.

Try this at your UNIX prompt: `cp .bash_profile quack`

Notice that the command `cp` has two arguments: the donor file and the recipient file. If you executed the last command successfully, you made a copy of your `.bash_profile` file to a file called `quack`.

Next, let's get rid of all the animals in the zoo we had created before. The command `rm` will accept one or more arguments and remove the named files. We can accomplish this in one blow with

```
unix> rm aardvark buffalo cougar dingo elephant
```

Now enter

```
unix> ls -l
```

You will see that `quack`'s size is nonzero because it has a copy of the contents of your `.bash_profile` file in it. The file shown here has size 191. The size is the number of bytes contained in the file; yours may be larger or smaller. You will also see that the menagerie has been sent packing.

```
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:50 bar
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:49 foo
-rw-r--r-- 1 morrison morrison   191 Jun  9 11:25 quack
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:49 stuff
```

Let us now remove the file `stuff`. We are going to use the `-i` option. Enter this at the UNIX prompt.

```
unix> rm -i stuff
```

The system will then ask you if you are sure you want to remove the file. Tell it yes by typing the letter `y`. Be reminded that the `-i` option is also available with `cp` and `mv`. You should you use it to avoid costly mistakes.

Finally, we shall use `mv` This "moves" a file to have a new name. Let's change the name of `quack` to `honk` and back again. To change `quack` to `honk`, proceed as follows.

```
unix> mv quack honk
```

Once you do this, list the files in long format. Then change it back.

Now you know how to copy, move, and create files. You can show them to the screen and you can list all the files you have. So far, we can create files two ways, we can create an empty file with `touch` or copy an existing file to a new file with `cp`.

8 Globbing and File Management

Globbing is a powerful tool for file mangement, but like its cousin the chainsaw, it is a dangerouis tool that demands respect and caution in its use. Remember that `mv`, `rm`, and `cp` can clobber files! Let's take it for s spin.

Let us begin by makinig some files and directories with these commands.

```
MAC:Tue Jan 04:13:19:tossme> mkdir jail
MAC:Tue Jan 04:13:19:tossme> cd jail
MAC:Tue Jan 04:13:19:jail> touch rocky flattop joker penguin riddler
MAC:Tue Jan 04:13:20:jail> cd ..
MAC:Tue Jan 04:13:20:tossme> ls -F
alan  betty chuck david emily faith jail/
```

How do we get the prisoners out. Watch this trick.

```
MAC:Tue Jan 04:13:22:tossme> ls
alan    chuck   emily   flattop joker   riddler
betty   david   faith   jail    penguin rocky
```

Notice the use of the . (dot). So all files are moved out of jail into our `cwd`. Let's jail everyone whose name contains a `t`.

```
MAC:Tue Jan 04:13:22:tossme> mv *t* jail
MAC:Tue Jan 04:13:24:tossme> ls jail
betty   faith   flattop
```

Now watch this.

```
MAC:Tue Jan 04:13:24:tossme> ls jail/f*
jail/faith   jail/flattop
```

Here are all prisoners whose name begins with the letter `f`. So, globbing can be used to create sets of arguments for UNIX commands.

8.1 A Reprise

Recall that every command in UNIX is a program. Now let us take a little look under the hood. When you log in, shell is launched. The shell accepts commands you enter at the prompt and sends them to the *kernel*, or operating system, which runs the program. This can cause output to be put to the screen, as in `ls`, or happen without comment, as in `rm`.

Programs that are running in UNIX are called *processes*. Every process has an owner and an integer associated with it called a process ID (`PID`). The user who spawns a process will generally be its owner. You are the owner of all processes you spawn. Many, such as `ls`, last such a short time you never notice them beyond the output they produce; they terminate in a fraction of a second after you enter them. When you log into your host, you actually are launching a program; this is your shell. When the shell terminates, your terminal session will be gone. At the command line, enter `ps` and you will see something like this.

```
unix> ps
PID TTY
10355 pts/1
10356 pts/1
unix>
TIME CMD
00:00:00 bash
00:00:00 ps
```

The `ps` command shows all processes currently running spawned by your shell. On this machine, the shell's (bash) process ID is 10355. By entering `ps aux` at the command line, you can see all processes running on your UNIX server, along with their process IDs and an abundance of other information. Try this at several different times. If you are using a server, you will see processes spawned by other users. You will also see other processes being run by the system to support your machine's operation.

An example of a program that does not finish its work immediately is the program bc. We show a sample bc session here; this application is a simple arbitrary-precision calculator.

```
unix> bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
3+4
```

```
7
4*5
20
2^20
1048576
2^100
1267650600228229401496703205376
quit
```

When you type `bc` at the command prompt, the shell runs the bc program. This program continues to run until you stop it by typing quit. To see `bc`'s process ID, start bc and then type Control-Z to put it to sleep. This interrupts the `bc` process, puts it in the background, and returns you to your shell. Then enter `ps` at the command prompt to see the process ID for your bc session.

```
unix> bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
[1]+
unix> ps
PID
14110
14253
14254
unix>
Stopped
TTY
pts/4
pts/4
pts/4
bc
TIME
00:00:00
00:00:00
00:00:00
CMD
bash
bc
ps
```

Try typing `exit` to log out; you will see something like this.

```
unix> exit
```

```
exit
There are stopped jobs.
unix>
```

Now type `jobs` at the command prompt. You will see this.

```
unix> jobs
[1]+ Stopped
unix>
bc
```

You can end the job `bc` labeled `[1]` by doing the following

```
unix> kill %1
unix> jobs
[1]+ Terminated
unix> jobs
unix>
bc
```

If several jobs are stopped, each will be listed with a number. You can end any you wish to by entering a `kill` command for each job. When you type `jobs` at the command line the first time, it will tell you what jobs it has suspended. After that, you will see a (possibly empty, like here) list of jobs still in the background. Do not dismiss a shell with running jobs; end them to preserve system resources.

You can bring your stopped job into the foreground by entering fg at the command prompt.

Exercises

1. Start up a session of bc and put it into the background using control-Z. Do this for several sessions. Type in some calculations into some of the sessions and see if they reappear when you bring the bc session containing that calculation into the foreground.
2. The bc calculator has variables which allow you to store numbers under a name. These play the role of the symbols described in Chapter 0, but they are limited to storing numbers. Here we show some variables being created and some expressions being evaluated.

   ```
   morrison@ghent:~unix> bc
   bc 1.06.94
   Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
   Free Software Foundation, Inc.
   ```

```
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
cow = 5
pig = 2
horse = 7
horse + cow
12
horse/pig
3
pig/horse
0
cow^horse
78125
```

Replicate this session. Put it into the background and bring it into the foreground. Were your variables saved? Notice that this calculator does integer arithmetic. The = sign you see is actually assignment, which was discussed in Chapter 0.

3. Look at one of the algorithms for converting a binary number into a decimal number described in Chapter 0. Can you step through the process using `bc` and make it work?

9 Editing Files

We can create files with `touch` and use `cp` to copy them. How do we edit text files and place information in them? This is the role of the UNIX text editor, `vi`. The O'Reilly book [**?**] on it comes highly recommended if you want to become a power user (you do). A second text editor, `emacs` is also available. It is powerful and extensible. Like `vi` it is a serious tool requiring serious learning, and like `vi` there is an O'Reilly book on it, too. You may use emacs instead of `vi` if you wish. Both of these are just tools for creating and editing text files, and both do a great job. You may create or modify any text file with either program. Ubuntu users can also use `gedit` or `gvim`, which have some nice advantages.

A Note for Ubuntu Users Ubuntu by default installs the package `vi-tiny`. We want `vi` with all bells and whistles. To get this, make sure you are connected to the Internet, then type the following command in an terminal window.

```
unix> sudo apt-get install vim
```

You will be asked to enter your password, then it will install the full vi package. The `sudo` command tells Ubuntu you are behaving as a system administrator, so you must enter your password to proceed. It will ask you to confirm you wish

to install, and then it will download the package from the repositories, install, and configure it for you. Ubuntu has lots of programs and packages that are freely available, and you use `sudo apt-get install` to obtain them.

9.1 Launching `vi`

To create a new file or open an existing file, type

```
unix> vi someFileName
```

at the UNIX command line. If the file *someFileName* exists, it will be opened; otherwise, it will be created. Now let us open the file `bar` we created with `touch`. You will see this:

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"bar" 0L 0C
```

DO NOT TYPE YET! Read ahead so you avoid having a passel of confusing annoying things plaguing you. The tildes on the side indicate empty lines; they are just placeholders that are not a part of the actual file. It is fairly standard for the tildes to be blue. The OL OC "bar" indicates that file `bar` has no lines and no characters. If the file `bar` were not empty, its contents would be displayed, then blue tildes would fill any empty screen lines.

9.2 `vi` Modes

Before you hit any keys there is something important to know. The `vi` editor is a moded editor. It has four modes: command mode, visual mode, insert mode, and replace mode. You will sometimes see command mode referred to as *normal mode*.

Command mode provides mobility, search/replace, and copy/paste capabilities. Insert mode allows you to insert characters using the keyboard. Visual mode provides the ability to select text using the keyboard, and then change or copy it. Replace mode overwrites existing text with new text that you type in. When you first open a file with `vi`, you will be in command mode.

We will begin by learning how to get into insert mode; there are lots of ways to do this. Here are a six basic ones that are most often used.

keystroke	Action
i	insert characters before the cursor
I	insert characters at the beginning of the line
a	append characters after the cursor
A	append characters at the end of the line
o	open a new line below the cursor
O	open a new line above the cursor

Here is an easy way to remember. What happens if you accidentally step on your cat's tail? He says IAO!!!

There is **one** way to get out of insert mode. You do this by hitting the escape (ESC) key. Let's now try this out. Go into your file `bar` and hit the `i` key to enter text. Type some text. Then hit ESC. To save your current effort type this anywhere:

```
:w
```

This will write the file; a message will appear at the bottom of the window indicating this has happened. Do not panic; that message is **not** a part of the file that is saved. To quit, type

```
:q
```

this will quit you out of the file. You can type

```
:wq
```

to write and quit. The command `:qw` does not work for obvious reasons. You have just done a simple `vi` session. You can reopen the file `bar` by typing

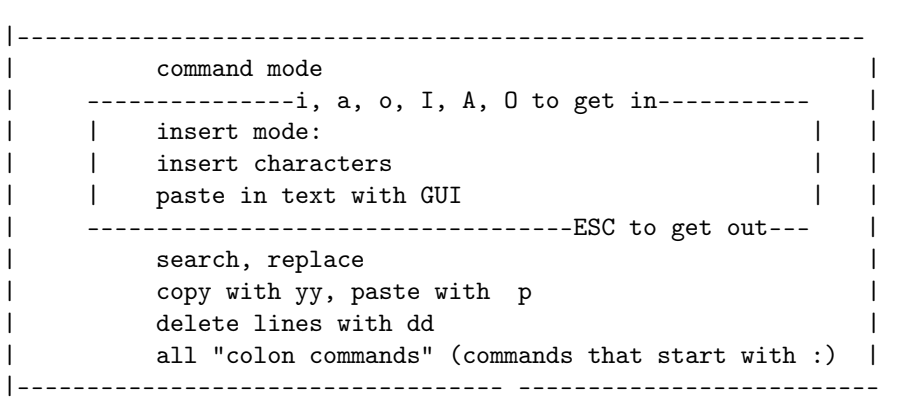
```
unix> vi bar
```

at the UNIX command line; the contents you last saved will be re-displayed. You should take a few minutes to try all of the ways of getting into insert mode. Change the file and save it. Quit and display it to the screen with `cat` and `more`.

At first, `vi` will seem clunky and awkward. However, as you ascend the learning curve, you will see that `vi` is blazingly fast and very efficient. One of its great strengths is the ability to search for and replace text. As your skill grows with it, you will see it is an amazing productivity tool. You might want to spend a little time at `https://openvim.com`, which is a great tool for learning `vi`.

A Reassuring Note If you are in command mode and hit ESC, your computer will just beep at you. This is its way of letting you know you were already in command mode. Nothing additional happens. If you are unsure what mode you are in, hit ESC and you will be back in command mode, no matter what. You can hit ESC and relax.

The figure below will help you to see the relationship between insert and command modes. When you first start editing a file, you enter in in command mode. Typing i, a, o, I, A or O all put you into insert mode. You can also see in the diagram how to get out of insert mode by typing ESC.



Let's go back in our file now and learn some more useful commands. We will look at command mode commands now.

Sometimes, line numbers will be helpful; these are especially useful when you program. To see them, you get into command mode and type the *colon command* `:set number`. Do this and watch them appear. Now type `:set nonu` or `:set nonumber` and watch them disappear. Line numbers are not a part of the file; however, they are a helpful convenience.

Here are some useful command mode mobility features. Experiment with them in a file.

Command	Action
:*lineNumber*	Go to indicated line number.
^	Go to the beginning of the current line.
$	Go to the end of the current line.
G	Go to the end of the file.
gg	Go to the beginning of the file; note that :1 also works.

These colon commands in this table will allow you to alter your editing environment. The last two are useful editing tricks that are sometimes quite convenient. Open a file and try them all.

Command	Action
:set number	display line numbers
:set nonu	get rid of line numbers
:set autoindent	This causes `vi` to autoindent.
:set noautoindent	This causes `vi` to turn off autoindent.
r (then a character)	replace character under cursor
~	change case upper → ;lower or lower → upper

9.3 Cut and Paste

The vi editor has a cache of memory called the *unstable buffer*, which we nickname Mabel. Mabel provides a temporary placd for holding things while we are editing and she is very helpful for doing quick copy-paste jobs.

This buffer is unstable because it loses its contents every time new text is placed in it. Do not use it to store things for a long time; instead write those things to files and retrieve them later. You will learn several ways to do this.

We show here a table with some cut, copy, and paste commands you will find helpful.

yy	Yank line to Mabel
dd	Delete line starting at the cursor; this cuts to Mabel
dw	Delete word; this cuts to Mabel
cw	Delete word, then enter insert mode(change word) The changed word is cut to Mabel.
p	Paste Mabel's contents at the cursor.
P	Paste Mabel's contents before the cursor.
D	Cut line at cursor; this cuts the stricken text to Mabel
C	Cut line at cursor and enter insert mode; this cuts the stricken text to Mabel

All of these commands can be preceded by a number, and they will happen that number of times. For example typing `10yy` in command mode will yank ten lines, starting at the cursor, to Mabel. Since so many of these commands place new text in Mabel, you should know that if you copy or cut to Mabel and intend to use the text, paste it right away. You should open a file and experiment with these. Spend some time fooling around with this mechanism; you will make some delightful discoveries, as well as dolorous ones.

9.4 Using External Files

You can select a range of line numbers before each of these commands, or select in visual mode and use these commands.

`:w fileName`	Write a copy of the entire the file `fileName`
`:w! fileName`	Write selection to existing file `fileName`, and clobber it.
`:w >> fileName`	Append selection to file `fileName`.
`:r fileName`	Read in file `fileName` starting at the cursor

For example

```
:20,25 w foo.txt
```

will write lines 20-25 to the file `foo.txt`. If you want to write the entire file, omit the line numbers and that will happen. If you want to write from line 20 to the end of the file, the usage is as follows.

```
:20,$ w foo.txt
```

Note the use of `$` to mean "end of file." When you learn about visual mode (just ahead), you can use these command to act on things you select in visual mode as well.

Housekeeping Tip If you use this facility, adopt a naming convention for these files you create on a short-term basis. When you are done editing, get rid of them or they become a choking kudzu and a source of confusion in your file system. Use names such as `buf`, `buf1`, etc as a signal to yourself that these files quickly outlive their usefulness and can be chucked.

9.5 Search and Replace

Finally we shall look at search capabilities. These all belong to command mode. Enter

```
/someString
```

in command mode and `vi` will seek out the first instance of that string in the file or tell you it is not found. Type an `n` to find the next instance. Type `N` to reverse direction. You can enter

```
?someString
```

to search for someString backwards from the cursor. Type `n` to find the previous instance, and `N` to reverse direction. Your machine may be configured to highlight every instance of the string you searched for. If you find this feature annoying, you can deactivate it with

```
:set nohlsearch
```

Now let us look at search and replace. This is done by a colon command having this form.

```
:s/old/new/(g|c|i)
```

The `s` means substitute; this substitutes `old` for `new`. The three *flags* at the end specify how the substitution should work By default, substitutions are confined to the cursor line, but you can control the scope of a substitution in these two ways.

Bound	Scope
`a,b s/old/new/(g|c|i)`	Perform the substitution on lines `a` through `b`, inclusive.
`a, $`	Perform the substitution on line `a` until the bottom of the file.

Here is how the flags work. At the end you can append any of `g`, `c`, or `i`. Here is a decoder ring.

`c`	Check after each substitution to see if you want to replace.
`g`	Replaces all instances on each line. By default, only the first one is replaced.
`i`	Replace `old` case-insensitive.

You will also learn how to control the scope of substitutions in visual mode below. That method is extremely nice and quite simple to learn.

10 .vimrc

You have the option of creating a `.vimrc` file in your home directory. This file can customize your editing environment. If you use a Mac, you should have these lines in your `.vimrc`.

```
syntax on
set et
set tabstop=4
```

This file runs every time you open a session with `vi`. The three lines present in the example enable syntax coloring (you want this), expand tabs into spaces (for Python sanity), and set your tabstop to 4 spaces. *Note:* Type this exactly as shown; do not put spaces around the = or you will get nastygrams whenever you open `vi`.

If you want line numbers by default, just add this line.

```
set number
```

If you find yourself using several colon commands at the start of a session, add them to your `.vimrc`.

11 Visual Mode

The third mode of the `vi` editor, visual mode is actually three modes in one: line mode, character mode, and block mode. To enter line mode from command mode, hit `V`; to enter character mode hit `v`, and to enter block mode, hit Control-v. You can exit any of these by hitting the ESC key; this places you back in command mode. Visual mode has one purpose: it allows you to select text using keyboard commands; you may then perform various operations on these selections. First, let us see the selection mechanism at work.

Go into a file and position your cursor in the middle of a line. Hit `v` to enter visual character mode. Now use the arrow keys; notice how the selected text changes in response to arrow key movement. Try entering gg and G and see what happens. Hit ESC to finish. Now enter visual mode and use the `/` search facility to search up something on the page. What happens? Search backward and try that too.

Now enter visual line mode by hitting `V`; now try the keystrokes we just indicated and see how the selection behaves. This mode only selects whole lines.

Finally if you enter Control-V and you enter visual block mode, you can select a rectangular block of text from the screen by using the keyboard.

Now let's see what you can do with these selections. First let us look at character and line mode, as block mode behaves a little differently. You can cut the selected text to Mabel by hitting `d`. You can yank it into Mabel by hitting `y`. Upon typing either command, you will be put back into command mode. Once any text is placed into Mabel, you can paste it with `p` as you would any other text yanked there. If you hit `c`, the selection will be deleted and you will be in insert mode so you can change the text. The stricken text is cut to Mabel.

In block mode, things are a little different. If you hit `d`, the selected block will be deleted, and the lines containing it shortened. The stricken text is cut to

Mabel. If you hit `y`, the block will be yanked just as in any other visual mode, and its line structure will be preserved. If you hit `c`, and enter text, the same change will be made on all line selected provided you do not hit the ENTER key. If you do, the change will only be carried out on the first line. You can insert text rather than change by hitting I, entering your text, and then hitting ESC. If the text you enter has no newline in it, the same text will be added to each line; if it has a newline, only the first line is changed.

If you hit `r` then any character in any visual mode, all selected characters are changed to that character.

Here is a very common use for character or line visual mode. Suppose you are editing a document and the lines end in very jagged fashion. This sort of thing will commonly happen when maintaining a web or if you are editing a LaTeXdocument such as this one, where the page that is subjected to repeated edits. Use visual mode to select the affected paragraphs and hit `gq` (think Gentleman's Quarterly) and your paragraphs will be tidied up.

You can also do search-and-replace using visual mode to select the text to be acted upon. Simply select the text in visual mode. Then hit

```
: s/outText/inText/g
```

to perform the substitution in the selected text. For example if you select text in visual mode and change every w to a v, you will see this.

```
:'<,'>s/w/v/g
```

The `<,'>` is a quirky way of indicating you are doing a visual-mode search-replace operation.

11.1 Replace Mode

In `vi` if you hit `r` then a character, the character under the cursor is replaced with the character you it. If you hit `R`, you are in *replace mode*, and any test you type "overruns" existing text. Experiment with this in a file you don't care about.

Replace mode is fabulous for making ASCII art such as this.

```
< Galactophagy >
 --------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

You should play around with this. Do a Google search to learn about ASCII art.

12 Copy Paste from a GUI

You can copy and paste with the mouse in a window or between windows. The way you do it varies by OS so we will quickly discuss each. If you are pasting into a file you are editing with `vi`, it is a smart idea to use the colon command `:set paste`. This will prevent the "mad spraying" of text. For certain types of files, this turns off automatic indentation or formatting. You can use `:set nopaste` to turn off the paste mode.

Windoze If you are copying *from* a Windoze application into a terminal window, select the text you want to copy and use control-C in the usual way. This places the text in your Windoze system clipboard. Now go into your terminal window and get into insert mode where you want to paste. It is also wise, in command mode, to enter `:set paste`. Right-click to paste the contents of your system clipboard into the terminal window. Many of you will say, "Why did the beginning of the text I copied get cut off or why didn't it appear at all?" This will occur if you are not in insert mode when you paste. It is important to be in insert mode before pasting to avoid unpleasant surprises. If this happens, hit ESC then `u` in command mode. The `u` command undoes the last `vi` command. Then you can take a fresh run at it.

If you are copying from a terminal window, select the text you wish to copy; PuTTY will place the text in your system clipboard. Then go into the window in which you wish to paste it. If the window is another terminal, get into insert mode and right-click on the mouse. If it's a Windoze app, use control-V as you usually do.

Mac Use command-c to copy and command-v to paste to or from a terminal window, just as you would with any other mac app. Mac gets this right.

Linux If you use a Linux box, use control-shift-C for copying in terminal windows and control-shift-V for pasting to terminal windows.

A Reprise: A Warning About `autoindent` and `paste` Before pasting with the mouse make sure you have autoindent turned off. Otherwise, your text will "go mad and spray everywhere," especially if you are copying a large block of text with indents in it. You can turn autoindent on with `:set autoindent` and off with `:set noautoindent`. This feature can be convenient when editing certain types of files. You can use the command `:set paste` to turn off all

smart indentation; when finished use `:set nopaste` to set things back to their original state.

A Warning abut Line Numbers If you copy-paste to a GUI, line numbers will get copied. To prevent this from happening, use the colon command `:set nonu` before copying.

Experiment with these new techniques in some files. Deliberately make mistakes and see what happens. Then when you are editing files, you will know what to expect and how to recover.

There are a lot of excellent tutorials on `vi` on the web; avail yourself of these to learn more. Remember the most important thing: you never stop learning `vi`! Here are some useful `vi` resources on the web.

- The site [**?**] for is complete, organized and well-written. You can download the whole shebang in a PDF. Read this in little bits and try a few new tricks at a time.
- The site [**?**], `vi` for Smarties will introduce you to `vi` with a bit of churlish sneery attitude. It's pretty cool. And it's sneery like the author of this august volume.
- The link `ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf` will download The Vim book for you. It is a very comprehensive guide, and it has excellent coverage on the visual mode.

12.1 Permissions

Now we will see how you can use permissions to control the visibility of your files on the system. You are the owner of your home directory and all directories and files it contains. This is your "subtree" of the system's directories belonging to you. You may grant, revoke or configure permissions for all the files and directories you own as you wish. UNIX was designed with the fundamental idea that your data are your property, and you can control what others see of them.

There are three layers of permission: you, your group, and others. You is letter `u`, your group is letter `g` and others is letter `o`. There are three types of permission for each of these: read, write and execute. Read means that level can read the file, write means that level can execute the file, and execute means that level can execute the file. In the example above, the file `bar` has the *permission string*

```
-rw-rw-r--
```

which means the following.

- You can read or write to the file. You cannot execute.
- Your group can read or write but not execute.
- The world can read this file but neither write nor execute.

For the user to execute this file, use the `chmod` command as follows

```
unix> chmod u+x bar
```

The u(ser's, that's you) permission changed to allow the user to execute the file.

If you do not want the world to see this file you could enter

```
unix> chmod o-r bar
```

and revoke permission for the world to see the file `bar`. Since you are the owner of the file, you have this right. In general you can do

```
$  chmod (u or g or o)(+ or -)(r or w or x) fileName
```

to manage permissions. You can omit the u, g or o and the permission will be added or deleted for all three categories. In the next subsection, we discuss the octal representation of the permissions string. This will allow you to change all three levels of permissions at once quickly and easily.

13 The Octal Representation

There is also a numerical representation for permissions. This representation is a three-digit octal (base 8) number. Each permission has a number value as follows.

- The permission `r` has value 4.
- The permission `w` has value 2.
- The permission `x` has value 1.
- Lack of any permission has has value 0.

We show how to translate a string in this example.

```
-rw-r--r--
  6  4  4
```

The only way to get a sum of 6 from 1,2 and 4 is 4 + 2; therefore, 6 is read-write permission. The string translates into three digits 0-7; this file has 644 permissions. It is a simple exercise to look at all the digits 0-7 and see what permissions they convey.

We show some more examples of `chmod` at work. Look at how the permissions change in response to the chmod commands. Suppose we are a directory containing one file named `empty`, which has permission string `-rw-r-r-`, or `644`. We begin by revoking the read permission from others.

```
unix> chmod o-r empty
```

We now list the files in the directory

```
unix> ls -l
total 0
-rwxr----- 1 morrison morrison 0 2008-08-26 10:52 empty
unix> ls
empty
```

We can now restore the original permissions all at once by using the octal number representation for our permissions.

```
 chmod 644 empty
unix> ls -l
total 0
-rw-r--r-- 1 morrison morrison 0 2008-08-26 10:52 empty
```

Notice what happens when we try to use a 9 for a permission string.

```
unix> chmod 955 empty
chmod: invalid mode: ‘955’
Try ‘chmod --help’ for more information.
```

Try typing the `chmod -help` command at your prompt and it will show you some useful information about the chmod command. Almost all UNIX commands have this help feature.

Directories must have executable permissions, or they cannot be entered, and their contents are invisible. Here we use the `-a` option on `ls`. Notice that the current working directory and the directory above it have execute permissions at all levels. Try revoking execute permissions from one of your directories and attempt to enter it with `cd`; you will get a `Permission Denied` nastygram from the operating system.

```
unix> ls -al
```

```
total 20
drwxr-xr-x 2 morrison faculty 4096 2008-10-17 11:51 .
drwx--x--x 9 morrison faculty 4096 2008-10-16 08:39 ..
-rw-r--r-- 1 morrison faculty 0 2008-10-17 11:51 empty
unix>
```

Here we shall do this so you can bear witness

```
unix> mkdir fake
unix> chmod u-x fake
unix> cd fake
bash: cd: fake: Permission denied
unix>
```

Assigning 600 permissions to a file is a way to prevent anyone but yourself from seeing or modifying that file. It is a quick and useful way of hiding things from public view. Later, when you create a web page, you can use this command to hide files or directories in your website that you do not want to be visible.

13.1 Globbing and Permissions

You can use globbing to change permissions on a glob of files. For eample

```
chmod 644 *.html
```

gives all HTML files in your `cwd` 644 permissions.

14 The Man

The command `man` is your friend. Type man then your favorite UNIX command to have its inner secrets exposed! For example, at the UNIX prompt, enter

```
unix> man cat
```

This brings up the man(ual) page for the command `cat`. A complete list of options is furnished. Notice that some of these have the `-`, or long form.

```
CAT(1)                User Commands                    CAT(1)

NAME
       cat - concatenate files and print on the standard output
```



```
SYNOPSIS
       cat [OPTION] [FILE]...

DESCRIPTION
       Concatenate FILE(s), or standard input, to standard output.

       -A, --show-all
              equivalent to -vET

       -b, --number-nonblank
              number nonblank output lines

       -e     equivalent to -vE

       -E, --show-ends
              display $ at end of each line

       -n, --number
              number all output lines

       -s, --squeeze-blank
              never more than one single blank line

       -t     equivalent to -vT

       -T, --show-tabs
              display TAB characters as ^I

       -u     (ignored)

       -v, --show-nonprinting
              use ^ and M- notation, except for LFD and TAB

       --help display this help and exit

       --version
              output version information and exit

       With no FILE, or when FILE is -, read standard input.

EXAMPLES
       cat f  g
              Output f's contents, then standard input,
              then g's contents.

       cat    Copy standard input to standard output.
```

```
AUTHOR
       Written by Torbjorn Granlund and Richard M. Stallman.

REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT
       Copyright Â© 2006
       Free Software Foundation, Inc.
       This  is  free software.  You may redistribute
       copies of it under the terms of the GNU General
       Public License <http://www.gnu.org/licenses/gpl.html>.
       There is NO WARRANTY, to the extent permitted by law.
SEE ALSO
       The full documentation for cat is maintained
       as a Texinfo manual.  If the info and cat
       programs are  properly installed at your site,
       the command info cat should give you access
       to the complete manual.

cat 5.97     August 2006                        CAT(1)
```

You can see here that even humble cat has some options to enhance its usefulness. Here is `cat` at work on a file named `trap.py`.

```
unix> cat trap.py
def trap(a, b, n, f):
    a = float(a)
    b = float(b)
    h = (b - a)/n
    list = map(lambda x: a + h*x, range(0,n+1))
    tot = .5*(f(a) + f(b))
    tot += sum(map(f, list[1:n]))
    tot *= h
return tot
def f(x):
    return x*x
print trap(0,1,10,f)
print trap(1,2,100,f)
```

Using the -n option causes the output to have line numbers.

```
 cat -n trap.py
    1 def trap(a, b, n, f):
```

```
    2     a = float(a)
    3     b = float(b)
    4     h = (b - a)/n
    5     list = map(lambda x: a + h*x, range(0,n+1))
    6     tot = .5*(f(a) + f(b))
    7     tot += sum(map(f, list[1:n]))
    8     tot *= h
    9     return tot
   10 def f(x):
   11   return x*x
   12 print trap(0,1,10,f)
   13 print trap(1,2,100,f)
unix>
```

View the manual pages on commands such as `rm`, `ls chmod` and `cp` to learn more about each command. Experiment with the options you see there on some junky files you create and do not care about losing.

Exercises

1. Use the `man` command to learn about the UNIX commands `more` and `less`. You will see here, that in fact, less is more!
2. Use the `man` command to learn about the UNIX commands `head` and `tail`. Can you create a recipe to get the first and last lines of a file?
3. What does the `ls -R` command do?

15 Lights, Camera, Action! Where's the Script?

Sometimes you will find yourself doing certain chores repeatedly. An intelligent question to ask is, "Can't I just save this list of commands I keep typing over and over again in a file?"

Happily, the answer to this is "yes." It's called writing a *shell script*. In its simplest form, a shell script is just a list of UNIX commands in a file. We will see how to make one of these and run it. Begin by creating this file, `greet.sh`.

```
#!/bin/bash
echo Hello, $LOGNAME!
echo Here is the calendar for this month:
cal
```

Type commands you see in this file into your shell. You will see this.

```
unix> echo Hello, $LOGNAME!
Hello, morrison!
unix> echo Here is the calendar for this month:
Here is the calendar for this month:
unix> cal
    January 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Now give this file execute permissions like so.

```
unix> chmod +x greet.sh
```

Now run it (note the slash-dot).

```
unix> ./greet.sh
Hello, morrison!
Here is the calendar for this month:
    January 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

We can make this process even better. If you don't have one, create a directory named `bin` in your home directory. Then open the dotfile `.bash_profile` and add this line to it.

```
export PATH=$PATH:"/Users/morrison/bin"
```

Replace the `/Users/morrison` with the path to your home directory. Put your shiny new script into this directory. Then you don't need the slash-dot any more unless you are in the `bin` directory. This has the benefit of allowing you to run the script from anywhere in the file system.

16 Terminology Roundup

- **absolute path** This is a path specifying a file that starts at the root directory / or with a , and which indicates a file's absolute location within the file system.

- **append operator** This operator » causes a stream to appended to a target file.
- **clobber** This occurs when a directory or file is unlinked from the file system. Clobbered files, for all practical purposes, are irretrievable.
- **cwd** This is your shell's current working directory
- **directory** This is a special file that links to its contents. Directories behave just like folders on a PC or a Mac.
- **environment variable** This is a value that configures your account. We have, so far, met the environment variable `$HOME`, which points to your home directory.
- **filter** A UNIX filter is a program that accepts and input stream (by default `stdin`), transforms it, and sends it to an output stream (by default `stdout`.
- **flag** This is synonomous with an option.
- **home directory** This is the directory within a UNIX system that belongs to you
- **insert mode** In `vi` this mode allows you to type characters into a file via the keyboard.
- **kernel** This is the heart of the operating system.
- **normal mode** In `vi` this is command mode.
- **path** This is a sequence of directories that may have a regular file or directory at the end. If it does not start with a `/` or a `..`, it is a path relative to your `cwd`.
- **permission string** If you use `ls -l` this is the string consisting of r, w and x that gives the three levels of permission.
- **permissions** This determines what is allowed to happen with a file. They come in the flavors read (r), write (w), and execute (x).
- **permissions column** This is the column of permission strings you see when you execute `ls -l`.
- **pipe** This takes the output of one filter and makes if the input of the next.
- **processes** This refers to any running program.
- **processes ID** Every process is assigned a permanent ID for its lifetime. The kernel assigns and keeps track of these.
- **prompt** This is the string in a terminal telling you the kernel is ready to accept a command.
- **regular files** These are files that are not directories.
- **replace mode** In this mode, `vi` "mows down" characters as you type new ones.
- **root directory** / This is the top file in a UNIX file system.

- **shell** This is the program that accepts your commands and relays them to the kernel. It also shows you the kernel's replies.
- **shell script** This is an executable program of UNIX commands.
- **standard error** This is the stream where all error messages go; by default it is the screen.
- **standard input** This is the keyboard, but it can be redirected to be a file.
- **standard output** This is by default the screen. It can be redirected to a file or into a pipe.
- **terminal window** This is the container for the shell you are running.
- **unstable buffer** This is `vi`'s temporary storage place for items that are cut or deleted. Each time a new cut or deletion occurs, its contents are replaced.