Unix for the Beginning Mage

A Tutorial by Joe Topjian

Chapter One 6

A Brief History of Unix and Unix-like Spellcrafts *7*

The Terminal *7*

Chapter Two 9

Casting Your First Spell *10*

Familiar Concepts *10*

An Introduction to the Filesystem *11*

Paths *11*

Basic Navigation Around the Filesystem *12*

Turn On the Bright Lights! *16*

Exercises for Chapter 2 *17*

Spells Learned from Chapter 2 *19*

Chapter Three 20

Creating Empty Files *21*

Separating Files and Directories *21*

Editing Files *22*

Reading Files *24*

Copying and Moving files *25*

Using Paths In Your Spells *27*

Sending Files to the Grave *29*

Exercises for Chapter 3 *32*

Chapter 3 Advanced Lesson 33

A Shell of a Time *33*

Shell History *34*

Command Completion *34*

Customizing Your Prompt *34*

Spells Learned in Chapter 3 *36*

Chapter Four **37**

Chapter 4 Advanced Lesson **41**

Raging Rapids *41*

Viewing Partitions *41*

What Partitions Do *42*

Spells Learned in Chapter 4 *44*

Chapter Five **45**

Placing Shields On Files *49*

Using the Number Method to Place Shields on Files *51*

Exercises for Chapter 5 *54*

Chapter 5 Advanced Lesson **55**

Types of Files *55*

Plain File *55*

Directory *55*

Link *56*

Pipes *57*

Block Devices and Character Devices *57*

Spells Learned in Chapter 5 *58*

Chapter Six **59**

Casting Spells With Absolute Paths *60*

Reading the Instructions for Spells *62*

Searching for Spells *63*

Using Grep *64*

Pipes: the Coffee Filters of Spells *64*

Redirection *65*

Wildcards 66

Exercises for Chapter 6 69

Chapter 6 Advanced Lesson 71

The Three Basic File Descriptors 71

Standard Output 71

Standard Error 71

Standard Input 72

Combining File Descriptors 73

Disabling All Output 73

Spells Learned in Chapter 6 75

Chapter Seven 76

Finding Other Mages 77

Ownership 79

Passwords 79

Exercises for Chapter 7 81

Chapter 7 Advanced Lesson 82

Yet Another Root 82

How To Become Root 82

The Wheel Group 83

Root's UID 83

Spells Learned in Chapter 7 84

Chapter Eight 85

Daemons (not demons!) 86

Processes 86

Casting Lots of Spells 88

Killing Processes 91

Exercises for Chapter 8 93

Spells Learned in Chapter 8 94

Epilogue 95

Appendix A 97

Appendix B 99

Mac OS X 100

BSD 100

Linux 100

Resources 100

Chapter One

Our Guest is greeted and welcomed; a lesson in history; a description of our tools.

Hello, Young Mage! Welcome to the Tower of Nix! Here you'll begin your training in the skills of Unix. I'll be your Instructor and Narrator.

We here at the tower understand if you're a little nervous or unsure about learning something so new and different. However, I can assure you that your stay and training at the Tower will be none other than the best! Before we begin, I'd like to give you a brief history of the craft you are about to learn.

A Brief History of Unix and Unix-like Spellcrafts

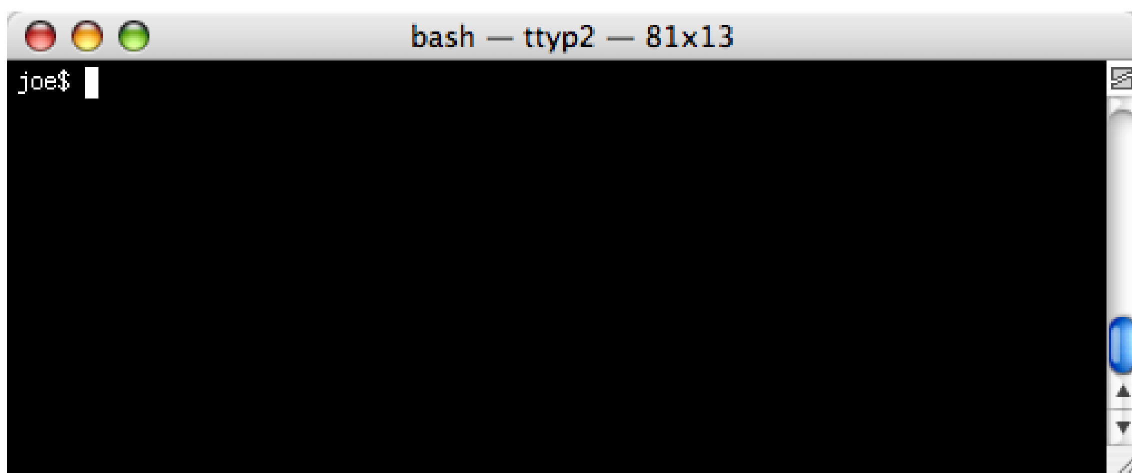
The Unix craft was created long before other spellcrafts -- before Windows, before Macintosh, even before DOS. It was first forged by archmages in the Fortress of AT&T back in the 1960's. Later, AT&T allowed other classes of mages the right to practice Unix -- but for a fealty. One of these classes was the University of Berkeley. Their branch of Unix became known as BSD, or, Berkeley Software Distribution. Eventually, Berkeley modified their craft of Unix so much that no remnants of AT&T existed. Berkeley's version of Unix is free for all and no one is required to pay fealty. Several other branches of Unix originated from BSD including FreeBSD, NetBSD, OpenBSD, Darwin, and Mac OSX.

At Vrije Universiteit in Amsterdam, another branch of Unix was created called Minix. Minix was forged from scratch and had no relation to AT&T at all. Minix was not a very popular branch of Unix but it was the inspiration to yet another branch: Linux. An archmage named Linus forged Linux and welcomed any other mage to help out. Today Linux is still being forged by thousands of mages around the world.

There are still several other branches of Unix that you must pay fealty for -- but the free ones have been gaining strong popularity. The most popular ones are the few I just mentioned. In the BSD area, there is FreeBSD, OpenBSD, NetBSD, Darwin, and Mac OSX. As for Linux, there are several different variations such as RedHat, Fedora, Debian, and SuSE. Though they may look different on the outside, they are all Linux at the core.

The Terminal

Before we begin, please make sure you are in front of a Unix Terminal. While describing every possible way to access a terminal can take up whole books of their own, we, unfortunately, don't have time for that. I will assume you have found access, one way or another, and are happily sitting in front of a blinking cursor that looks something like this:



The Terminal Program in Mac OSX

What you see before you is your terminal. The terminal is the main tool of our craft -- it's where we cast our spells. Though different terminals may look different, they all share a few characteristics.

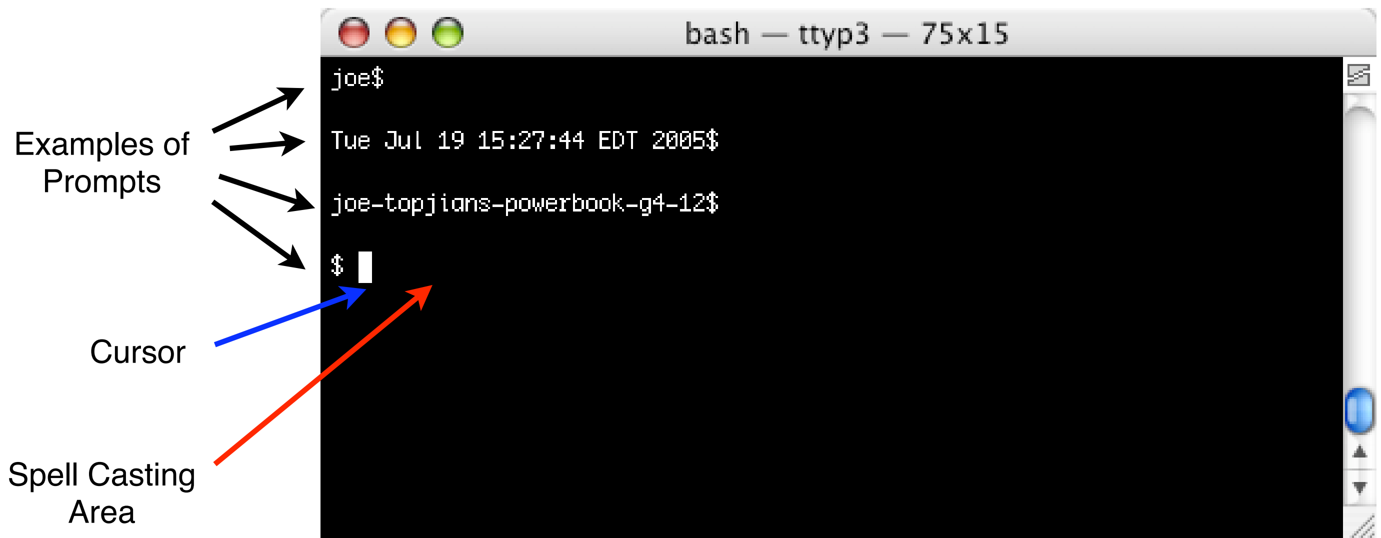


Diagram of a Terminal

Prompt: A prompt is just some text to let you know that your terminal is ready for you to cast a spell. It also has the ability to tell you other useful information. In the examples above, I have prompts set to my name, the date, and the name of my computer. Prompts usually end with a symbol such as $, >, %, or #.

Cursor: The cursor is a block or line to let you know where a letter is going to appear when you press a key.

Spell Casting Area: The spell casting area is just blank space. Any blank space after the prompt is where the letters you type will be displayed. It's in this blank space that you will be casting your spells.

Tip:

Spell Casting is known as *command execution* in the Real World.

Not bad for a first lesson, no? Now that you're a little more familiar with what you're about to get into, let's start casting spells!

Chapter Two

The Casting of the first spell; the Young Mage finds familiar ground; a talk of trees and houses; creating new rooms and exploring them.

Casting Your First Spell

To cast a spell, you simply type the name of the spell and press enter. Let's try a simple spell called `whoami`.

```
joe$ whoami (enter)
joe
joe$
```

Casting the whoami spell

`whoami` stands for Who Am I. As you can see in the figure, after you typed the word `whoami` and pressed enter, the spell returned the answer or *output*. The output of this spell is, oddly enough, your name (also referred to as a *username* or *login* name)[1]! The purpose of the `whoami` spell is to tell us the name of the user we are currently logged in as in Unix. It might sound silly right now, but it's possible to cast spells that turn us into other people!

Spells can return a wide range of information. Some return one word answers like `whoami`, some spit out pages and pages of text, some put you in interactive programs, and some just don't return anything at all!

Familiar Concepts

As you might have noticed, your mouse is not used in the terminal at all. If you've never used a command line before, this can be a confusing concept -- but fear not! You can bring over a lot of the concepts you learned in the Mouse and Window world to the command line world.

For example, with a mouse, you would *double-click* to open a directory or start a program. With Unix, we call this *spell casting* or *command execution*. You are also familiar with using your mouse for common, everyday tasks, such as moving files to new directories, renaming files, and even deleting files. In Unix, we have spells for all those actions as well! There is the `cp` spell to copy files, the `mv` spell to move and rename files, and the dangerous `rm` spell that deletes files[2].

You're also familiar with opening and closing folders with your mouse to reach different areas. This is called *browsing* or *exploring* in the Windows world. In Unix, we call this *navigating the filesystem*. Also, we call folders *directories* in Unix.

Speaking of the filesystem, that brings us to our next lesson.

---

[1] Make sure whoami did not return the name **root**. See Chapter 7 Advanced Lesson for more information.

[2] We say `rm` is dangerous since you cannot raise dead files from the grave in Unix!

An Introduction to the Filesystem
Our world of Unix starts with a single character:

```
/
```

This is called *root*. Just like the root of a tree, root is the beginning of the filesystem. A tree can grow several branches with leaves on the end. In our filesystem, directories grow from the root and contain files. You can even think of root as the front door to your house. Inside your house there are several rooms hallways leading to other rooms. The same is true for our filesystem; there are several directories and directories leading to other subdirectories.

Let's say you have a directory called **music**. In the filesystem, it would look like this:

```
/music
```

Now let's say you have a song called **mice_and_mages.mp3** in your music folder. That would look like this:

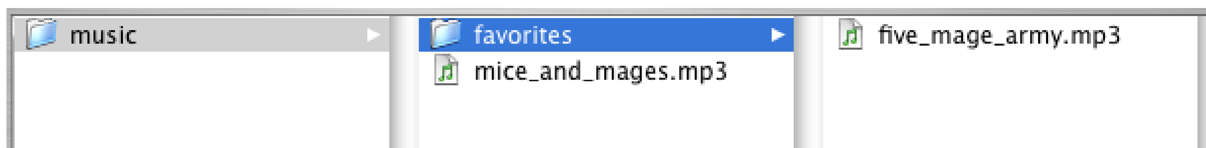
```
/music/mice_and_mages.mp3
```

Notice how root is always at the beginning. We then use other slashes (**/**) to separate files and directories. This is so we can tell where one stops and one begins. The full line is known as a *path.*

As a second example, let's say you had a directory inside **music** called **favorites**. A directory inside a directory is called a *subdirectory*. Inside your **favorites** folder you have another song called **five_mage_army.mp3**. Now it looks like this:

```
/music/favorites/five_mage_army.mp3
```

In the mouse and windows world, it would look similar to this:



Paths
Let's discuss paths a bit more. Paths are just that: *paths*. In a forest, we walk along paths so we don't get lost. In the Tower, we follow paths to different rooms. Filesystem paths are no different; they will lead you to different directories and files.

There are two types of paths to know about: *absolute* and *relative*. An absolute path is from the root to your destination. It's like being at the front door of your house and walking to the kitchen. You are starting from the very beginning of your house. The same is true with a filesystem path -- it starts from root (**/**). So any path you see that starts with a slash is an absolute path:

```
/mudroom/hallway/kitchen
```

A relative path is from your current location to your destination. For example, if you were in your hallway and wanted to go to your bedroom closet, that is a relative path. You're not starting at your front door since you're already in your house.

```
bedroom/closet
```

Relative paths do not start with a slash -- which is a very easy way to tell the difference between absolute and relative paths!

Now, say you're in your bedroom, but you wanted to leave and go to the kitchen.

```
../hallway/kitchen
```

Notice the *dot dot* (**..**). That is a special thing in Unix that says "I am leaving my current room".

Now that you understand paths better, let's start walking around the Tower of Nix.

Basic Navigation Around the Filesystem
First, it's always best to know where you are. To find that out, cast a spell called `pwd`.

```
joe$ pwd
/Users/joe
joe$
```

Casting the pwd spell

`pwd` stands for Print Working Directory and will tell you what directory in the filesystem you're currently in -- or what room of the Tower you are standing in.

The directory you're now in is also known as your *home directory*. Think of it as your dorm room in the Tower of Nix -- it's your room to store your belongings.

In order to move around, you'll need somewhere to go. Cast the spell of `mkdir`. `mkdir` stands for Make Directory. It will create a room for us to go in.

Tip:

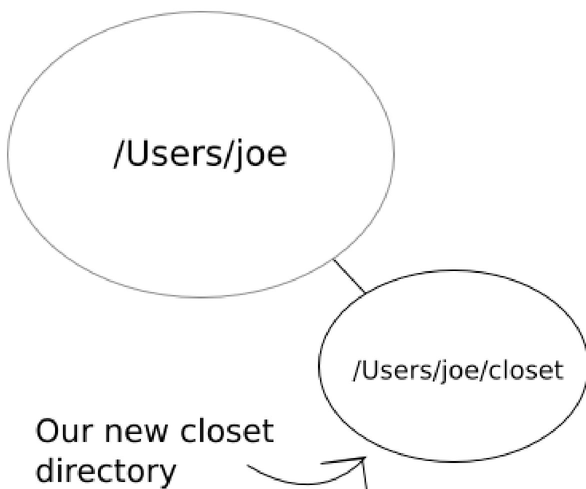
In OS X, you will see /Users. In Linux or any other Unix, you will see /home.

```
joe$ mkdir closet
joe$
```

Casting mkdir

Notice two things about this spell. The first is that you specified the word **closet** after `mkdir`. Some spells ask for extra information -- anything from one word to several. These extra words are called *arguments*. In the case of the `mkdir` spell, this argument told you what to call your room (**closet**).

The second thing to notice is that `mkdir` didn't tell you anything after you cast it. Remember before when I mentioned that some spells will never tell us anything? This is one of them! On the other hand, if something were to have gone wrong when you cast the spell, `mkdir` would've told you. So if everything went OK, it shuts up, but if there was a problem, it complains. I think we can handle that!



Here's a little visualization of the room you just created. It's an off-shoot of the directory you were already in -- just like how a closet is another room inside your own room!

Now that you have a room created, feel free to go inside! In order to do that, cast the spell called `cd` (which stands for *change directory*).

```
joe$ cd closet
joe$
```

Casting the cd spell

When casting `cd`, you need to specify the room you want to go in as an argument, so type it after the name of the spell. The name of the room is just a path. Think back to our recent discussion about paths. Since **closet** doesn't start with a slash, it's a relative path. This makes sense because you are starting from your current room and going into your closet.

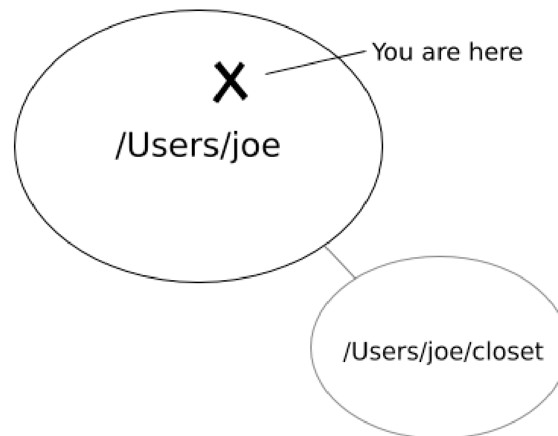
Again, the spell didn't tell you anything, so we'll assume everything went OK. To see if you're in the new directory, cast `pwd` again.

```
joe$ pwd
/Users/joe/closet
joe$
```

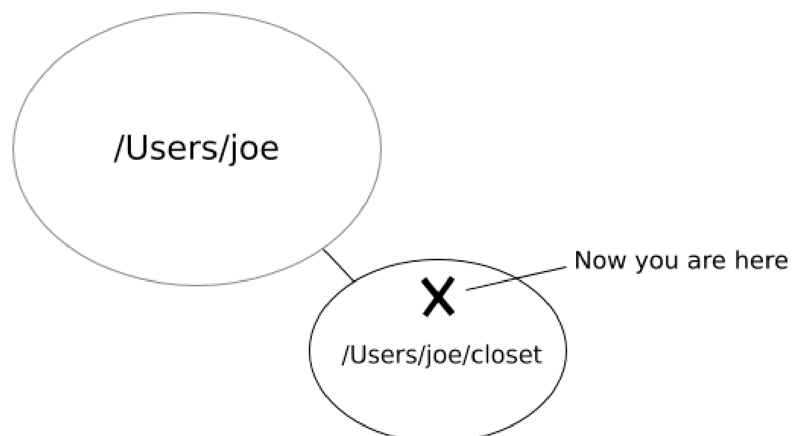
Casting pwd to see if cd worked

Yep! `pwd` shows that you are now in the closet!

Here's what it looked like before you cast `cd`:



And here's what it looks like after:



Now you will learn how to get out of the closet and back to your dorm room. There are actually four ways you can do that:

1) Typing `cd` with no arguments is a shortcut to take you directly to your home directory. So if you were ever lost in the middle of a forest, you can just type plain `cd` and you will be teleported directly home!

```
joe$ cd
```

2) Typing `cd` with a tilde at the end will also take you home. The tilde (~) is a nickname for your home -- weird, huh?

```
joe$ cd ~
```

3) From casting `pwd` earlier, you know that home is **/Users/joe**, which is right outside the closet. You also know that to go out of one room, you can use the magic *dot-dot*. So you can just cast:

```
joe$ cd ..
```

This will take you out of the room and in to the one before it.

4) Again, you know that home is at **/Users/joe**, so you could just use the absolute directory to get back to it by casting:

```
joe$ cd /Users/joe
```

So may choices! You may choose any way you wish!

Turn On the Bright Lights!

Next, you'll learn about a spell called `ls` that stands for *list*. ls will tell you everything that is in the directory -- kind of like turning on the lights in a room! Here's an example of how it works:

```
joe$ pwd
/Users/joe
joe$ ls
closet
```

Casting the ls spell

As you can see, `ls` shows the closet we made earlier. It might also show you a lot of other files and directories, but we're only worried about the closet right now. If the directory is empty, `ls` will show us nothing at all.

```
joe$ cd closet
joe$ ls
joe$
```

Casting ls in an empty directory

That ends this lesson, Young Mage. What all did we cover? You learned all about casting spells and how it's not *that* different from using a mouse. You also learned all about the different kinds of paths. Finally, you learned how to walk around the Tower of Nix. Not bad for a first day!

Try the exercises before the next lesson -- but don't stay up too late doing them!

Exercises for Chapter 2
Make sure you are starting at your home directory for each one.

1) Creating some more rooms

Inside your closet, you'll also need a room to keep all of your **spells** and **robes**. After all, what's a mage without his spells and magic robes? Use the mkdir spell to create these directories.

2) Navigation shortcuts

What happens when you cast the following spell?

```
joe$ cd closet/spells
joe$
```

Why does that work?

What about these spells?

```
joe$ cd ../..
joe$ cd closet/../closet
```

Why doesn't this work?

```
joe$ cd /closet/spells
```

3) Running into walls

What happens when you cast the following spell and why?

```
joe$ cd brickwall
```

4) Bonus Exercise

Roam free around the Tower of Nix using `cd`, `pwd`, and `ls`. Keep an eye out for interesting things and if you get lost, just cast plain `cd` to teleport home!

Spells Learned from Chapter 2

Spell	What it stands for	What it does
`whoami`	Who Am I?	Prints your username
`pwd`	Present Working Directory	Prints the absolute path of the directory you are in
`mkdir [name]`	Make Directory	Makes a directory with the specified name
`cd [path]`	Change Directory	Takes you to the specified path
`ls`	List	Prints the contents of the directory

Chapter Three

The Young Mage learns about files; discovering the world of nano; the many paths of Unix; and always make sure you have a backup.

Our next lesson is with Files. A *file* is anything that stores information. While a directory can store files and other directories, a file holds bytes and information. Files are the leaves on the tree or the furniture in the room.

There are many types of files in Unix. Some just contain text (like your research paper), some contain music (like an mp3), some are spells we can cast (like `mkdir` and `ls`), and others do special functions like provide access to your CDROM.

Creating Empty Files

There are many ways to create files. The first way is with the `touch` spell. One of the things that `touch` can do is create blank files for us. If you followed the exercises from **Chapter 2**, you now have a special room in your closet to store all your spells. Well, you can't just leave them lying around the room so you'll have to create a shelf for them!

```
joe$ cd closet/spells
joe$ touch shelf
joe$ ls
shelf
joe$
```

Creating your spell shelf

As you can see, you also cast your light-giving spell, `ls`, which tells you that the shelf was created.

Separating Files and Directories

Just having the word *shelf* on your terminal looks pretty confusing. How can we tell if it's a file or directory? Well one way is to try casting `cd` and see if you can teleport in it. If `cd` fails, then it's a file.

```
joe$ cd shelf
-bash: cd: shelf: Not a directory
joe$
```

Casting cd on a file

Yep, it failed. `cd` came back and told us it's not a directory. It also mentioned something about `bash`[3]. It would get pretty tedious if we had to cast cd every time we wanted to know if something was a file or directory. There *has* to be an easier way. Well, there is!

---

[3] `bash` will be explained in the Advanced Lesson after this chapter

```
joe$ ls -F
shelf
joe$
```

Casting ls with the -F argument

OK, that looks like it did no good at all. In fact, you even did extra work with that weird -F thing. Just give me a minute to explain before you call me crazy!

First, that weird -F thing is called an *option*. Anytime you cast a spell with a *dash-something* after it, you're using options. Options modify the spell to do different things. In this case, the -F tells `ls` to politely give us more detail about what's in the room. Unfortunately, if what's in the room is just a plain file, -F option won't report anything extra. However, it will tell us when there are directories in the room. Take a step out of your spell closet -- which takes you to your main closet -- and try casting `ls -F` again.

> Tip:
>
> Make sure you are using a capital F. Spells are *case-sensitive*!

```
joe$ cd ..
joe$ ls -F
spells/
joe$
```

Casting ls -F again

See! I'm not crazy after all! Notice the **/** at the end of **spells**. This tells us that **spells** is a directory! So when you're unsure whether or not something is a file or directory, casting `ls -F` will give you the answer. There are a few other methods that can tell the difference and we'll learn about those later.

Editing Files

Now you have your blank file, **shelf**. Let's start putting things in it. Why don't you have it hold all of the spells you know so far? In order to do that, you're going to have to *edit* the file, or, modify the information inside it.

First, go back into your spell closet where the shelf is located.

```
joe$ cd spells
joe$ pwd
/Users/joe/closet/spells
joe$
```

Going back to the spell directory

Next you'll need to use a spell called a *text editor*. There are several text editors available for Unix, but for now, you'll use a very simple one called `nano`. In order to have nano edit a file, you need to tell it what file we want edited.

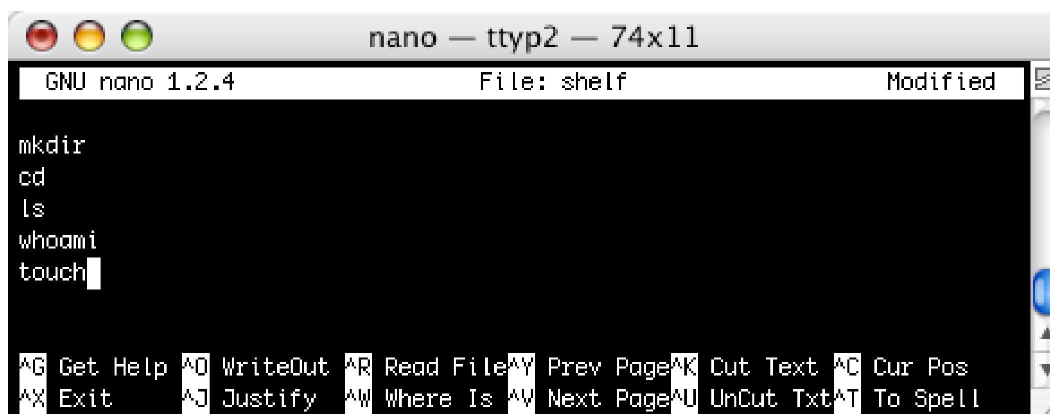
```
joe$ nano shelf
```

Casting nano

Once you hit enter, the spell takes you into a whole new world -- the wonderful world of `nano`. OK, that's too dramatic. But you did notice that your terminal just changed, right? This is called an *interactive spell* or an *interactive program*. It doesn't just return some output quit. Instead, you continue using the spell until you're done.

Think of `nano` as a very, *very* basic Microsoft Word. Go ahead and play around for a bit. Type some gibberish. Move around with your cursor keys. Play! When you're done, erase everything and add your spells. Just type them in -- no need to get really fancy. And in any order, too.

> Tip:
>
> If you get "`command not found`" when casting `nano`, try `pico`. They are the exact same (besides the name).



The nano text editor

Also notice the instructions at the bottom. That ^ is called a *caret* and means to use your control key on your keyboard. So, to exit, press *ctrl+x*. It will ask if you want to save the file before it quits -- press *y* to say “yes”, then *enter* to confirm the filename to save to. Now you’ll find yourself outside the `nano` program and back to the terminal.

You don’t always have to `touch` files before you edit them. You can cast `nano` on a file that doesn’t exist yet, and when you press *ctrl+x* to exit and save, it’ll be created.

Now that you have some information in your file, how can you read it? One way is to just cast `nano` again, but there are other spells that’ll let us read files.

Reading Files

The first spell is `cat`. `cat` will just spit the content of the file to the terminal. You can cast it like so:

```
joe$ cat shelf
mkdir
cd
ls
whoami
touch
joe$
```

Casting cat

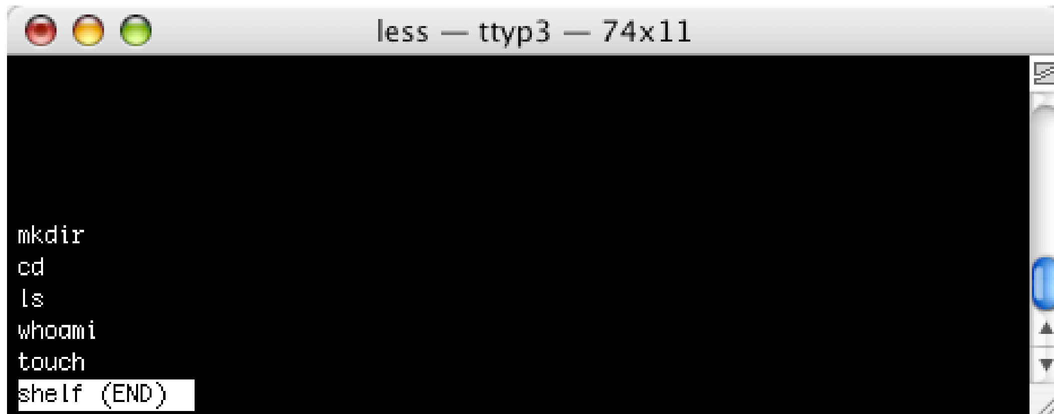
As you can see, `cat` is very simple. It takes a file as an argument and spits the content out. But what if the content of the file is too big for the terminal? You’ll never be able to see it since it’ll be cut off! Then you’ll need to use a spell that supports *paging*. A paging spell will stop when a screen is filled up with text and wait for you to press a key to continue. Two paging spells are `more` and `less`.

`less` is the preferred spell to use since you can use your cursor keys to move up and down the contents as well as using your space bar to scroll a whole page. When you are done reading the file, you must hit `q` to stop using `less` (it is another interactive spell).

To cast `less`, type it with a file name for an argument -- just like you do with `cat`.

Tip:

Historically, `more` came first, then `less`. `less` has more features since less is more!



The less spell

Now you know four ways to read files. Believe it or not, there are still many other ways to read them! But four is good enough for now -- we've got a lot more stuff to cover besides reading files!

Copying and Moving files

What if you wanted to make a backup copy of your file just incase your closet caught on fire? To do that, we can use the `cp` spell. `cp` stands for Copy and, as the name indicates, it creates a copy of a file.

```
joe$ cp shelf shelf-backup
joe$ ls
shelf           shelf-backup
joe$
```

Casting the cp spell

A couple things to notice with this spell. First, it's the first spell you've cast that uses two arguments. The first argument is the name of the file you want to copy and the second argument is the name you want to give the copy. The second thing to notice is when you cast `ls`, you see that `cp` has, in fact, created your backup copy, so you now have two files.

Having your backup copy in the same room as your original won't do you any good since if the room catches on fire, both will go up in flames. So make a fireproof box to store it in. That way, it'll be safe.

```
joe$ mkdir fireproofbox
joe$ ls -F
fireproofbox/       shelf          shelf-backup
joe$
```

Creating the fireproof box

Tip:

mv can also be thought of as 're-name' as well as 'move'.

Notice how `ls -F` shows you which one is a directory and which ones are files.

Now you'll move the backup copy to your fireproof box. Moving files is done with the `mv` spell (guess what that stands for). `mv` takes two arguments just like `cp`. The only difference, really, between `mv` and `cp` is that `cp` leaves a copy behind while `mv` doesn't.

```
joe$ mv shelf-backup fireproofbox/shelf-backup
joe$ ls -F
fireproofbox/       shelf
joe$
```

Moving the backup copy to the fireproof box

Again, two things to point out here. One is that `ls -F` tells you that the backup copy is definitely gone from its original location. The second point deals with the two arguments of mv. The first argument is the name of the file we are moving -- that's easy to see. But what is up with the second argument? There seems to be a slash in it. From our last lesson, we know that paths contain slashes. So that must mean that we can use slashes and paths in more of our spells!

What the `mv` spell did was give the backup copy a new name, just like we wanted -- but it gave it a new name in another directory. And it used a path to do it! Think of it like you were physically placing the file inside your fireproof box.

As a shortcut, you didn't have to cast

```
joe$ mv shelf-backup fireproofbox/shelf-backup
```

Instead, you could have just cast

```
joe$ mv shelf-backup fireproofbox
```

The two spells mean the same thing. Unix is smart enough to know that the second argument is a directory and that you obviously want to put the file in the directory with the same name.

Using Paths In Your Spells

If `mv` can take paths for its arguments, what other spells can? Well, since `mv` and `cp` are basically the same thing, `cp` can as well. `ls` can, too. Oh and `cat`. Come to think of it, `more` and `less` can. It works with `touch`. `mkdir`, also.

How can all these files use paths? What is the common thing between all of them? Their arguments specify files. And anywhere we specify a file, we can specify a path, too. Here are some examples:

```
joe$ cp shelf another_shelf
```

Here you are just making a copy of your shelf, just like you did before. Nothing new here.

```
joe$ mv another_shelf /Users/joe/closet
```

*Now you are casting `mv` and telling it move **another_shelf** into the directory **/Users/joe/closet**.*

```
joe$ ls -F /Users/joe/closet
another_shelf  spells/
joe$
```

In this example, you cast `ls` to see what's in another directory.
Kind of like having a magic eye where you can spy on other rooms!

```
joe$ cp /Users/joe/closet/another_shelf fireproofbox
```

Now you are casting `cp`, but with paths for both the first and second arguments.

```
joe$ cp fireproofbox/another_shelf ..
```

You can even copy things to the magic dot-dot directory.
*This will place a copy of **another_shelf** in the room before the one you're currently in.*

```
joe$ less ../another_shelf (press q to quit)
```

*Casting less to read the file, **another_shelf**, in the room outside your current room.*

Are you getting the hang, yet?

```
joe$ cp ../another_shelf .
```

Wait! Wait! Wait! What in the *world* is that dot? Everything was going fine until that dot showed up and now everything is confusing! We know what a *dot-dot* is but we've never seen a single *dot.*

It's OK! It's not as bad as it seems, I promise!

You know that *dot-dot* means the directory below us, or the room we came in through. Well, the single *dot* just means *the current directory we are in,* or, *the room we are standing in right now*. That's it. No rocket science. Just plain *here.*

The example above is using both the *dot-dot* and the single *dot.* Picture grabbing something outside of your room and bringing it in. Or any action movie where someone gets dragged to safety before the boulder crushes them. That's what this example is doing.

But using a single *dot* isn't the only way to tell a spell you want something moved *here.* There are a few ways that mean the exact same thing. You could even call these *spell synonyms.*

```
joe$ cp ../another_shelf .
joe$ cp ../another_shelf another_shelf
joe$ cp ../another_shelf ./another_shelf
```

Some spell synonyms

Note that the following are not the same:

```
joe$ cp ../another_shelf .another_shelf
joe$ cp ../another_shelf /another_shelf
```

These spells are not the same

The first one will actually name the file **.another_file** -- the file starts with a dot. The second one will copy your file to the root of the filesystem. This will spit an error out at you since only archmages are allowed to do that[4].

But if we combine them to a *dot-slash*, it will mean the same thing as a single *dot.* Two wrongs make a right, I guess.

Sending Files to the Grave

Unfortunately, after our little Path Party, you have shelves laying around all over the place. You don't need so many copies -- plus, it just makes your dorm messy. So, get rid of them! There's a special spell just for that: the `rm` spell. `rm` stands for remove. It's that dangerous spell I mentioned during the last lesson. Always be careful with `rm` because once you remove a file, it is gone forever. `rm` works just like the other spells, you cast it with a path at the end.

```
joe$ ls -F
another_shelf      fireproofbox/      shelf
joe$ rm another_shelf
joe$ ls -F
fireproofbox/      shelf
joe$
```

Casting rm

As you can see, the file is gone. Forever. There are no spells to raise the dead in Unix.

OK, finish cleaning your dorm up. As a matter of fact, get rid of the fireproof box, too. I doubt there's going to be a fire in the tower any time soon.

---

[4] If it did not print an error, cast `whoami` to see if you are the user **root**. If you are, see Chapter 7 Advanced Lesson for more information.

```
joe$ rm ../another_shelf
joe$ rm fireproofbox
rm: fireproofbox: is a directory
joe$
```

Casting rm on a directory

Yep, that's right. We can't cast `rm` on a directory. Instead, we have to cast `rmdir`. `rmdir` is the evil twin of `mkdir` -- it takes away directories instead of making them.

```
joe$ rmdir fireproofbox
rmdir: fireproofbox: Directory not empty
joe$
```

Casting rmdir

Now what's this? The directory has to be empty!? Why, that means we have to cast `cd` to teleport in, `ls` to see all of the files, then cast `rm` on all of those until we have an empty directory! And what if the directory has subdirectories? We have to do the same thing over again! Why would we ever do all that work?

Well, that's the safe way to remove things. Since you can't bring files back from the dead, Unix wants to make sure you know what you're doing.

There is a shortcut, however -- but it's a very dangerous shortcut. It's one of the most powerful spells in Unix. If used the wrong way, you can lose *all* of your information. And you won't be able to bring any of it back. The spell is known as `rm -rf`. The `-rf` is an option that stands for *recursive* (`r`) and *force* (`f`) -- basically, "I don't *care*, just get rid of it *all!*"

```
joe$ rm -rf fireproofbox/
joe$ ls -F
shelf
joe$
```

Casting rm -rf

And there goes our fireproof box. Just like that. At least we still have our original shelf. After all that trouble, we reach the end of our lesson with just a single shelf -- but a lot of knowledge and more spells to add to it!

Before I end this lesson, I just want to remind you again how dangerous `rm -rf` can be if used the wrong way. Essentially, you can burn the whole Tower down by typing

```
joe$ rm -rf /
```

Burning down the house

Fortunately, there are shields in all the Towers that won't allow you to do such a thing. However, an archmage with more power can do it -- but why would they? It would be the same as burning your house down. Just be careful, Young Mage!

Exercises for Chapter 3
Remember to start from your home directory for each one.

1) Cloning boxes

Go to your spell closet. Create a new fireproof box and place a copy of your shelf inside. Next, make a copy of your fireproof box called **iceproofbox**. In order to copy directories, you will need to use the `-R` option (which means *recursive*):

```
joe$ cp -R fireproofbox iceproofbox
```

Do the contents of the directory copy over as well? Or is it empty?

Now cast `mv` to move **iceproofbox** to **airtightbox**. Again, did everything copy over?

2) Oh, what to do?

Using `nano`, create a new file called **todo**. Inside **todo**, write a list of things you would like to do next week. Next, figure out this files absolute path and cast `cat`, `more`, and `less` with the absolute path.

3) Bonus Exercise - Attention! Extension!

From living in the Mouse and Window world, you know that files have extensions at the end of them -- those 3 character things after the dot like **.txt, .mp3,** and **.doc**.

Unix files can have those, too, but they aren't required.

For the first part of this exercise, rename your todo list that you made in Exercise 2 so it has a **.txt** extension at the end of it. Look a little more familiar now?

For the second part, you will use a new spell called `file`. When you cast `file` on a file, it will tell you what kind it is. Try casting it on your todo list:

```
joe$ file todo.txt
```

4) Finally, walk around the Tower and cast ls -F and `file` on various files you see. Note the different types of files you'll run into!

Chapter 3 Advanced Lesson

A Shell of a Time
In this lesson, you're going to learn all about *shells*. A *shell* does a lot of things -- it runs the command once you hit enter, it provides your prompt, it keeps variables for you -- lots!

Think of Unix as some foreign stranger you bump into on the street. All of the sudden you're both yelling at each other in two different languages -- you in English and Unix in binary. Neither of you understand one another and nothing gets accomplished.

This is where the shell comes in. The shell is that guy that runs out of the deli, breaking up the fight between you and Unix, speaking both languages.

The name *shell* originally came about because it's viewed as the outermost layer of Unix. Like an egg -- the egg and yolk are the Operating System while the shell is, well, the shell.

Without the shell, you'd have no way of communicating with Unix -- just two guys yelling in a street.

Historically, the very first shell was known as `sh`. As time went on, more shells were created: `csh`, `tcsh`, and `bash` for instance. `bash` is the most popular one today, and chances are, it's what you're currently using. You can see what shell you're using by echoing an environment variable. Environment variables and `echo` will be explained in a later chapter. (This is an *advanced lesson*, after all!)

```
joe$ echo $SHELL
/bin/bash
joe$
```

Echoing the $SHELL variable

`bash` uses the $SHELL variable while others use `$shell`. Remember, Unix is case sensitive -- upper and lowercase letters make a difference! If nothing showed up on your screen, try the next example.

```
joe$ echo $shell
/bin/tcsh
joe$
```

Echoing the $shell variable

Shells can provide us with some cool features. We'll be looking at three of them: history, command completion, and customizing your prompt.

Shell History
You're being watched. Every command you type is being recorded -- by the shell! Why? Just incase you forget how to run a command or maybe you want to see how someone else typed a command.

You can see your history of commands by typing just that -- `history`.

You can also view commands you've previously used by using your up and down cursor keys. Pretty neat, huh?

Command Completion
What if you couldn't remember how a command was spelled, but you knew it began with `ca`? With command completion, you can type the letters `ca` then press tab twice. A list of all the commands that start with `ca` will appear!

If there is only one command that fits the description you're typing, pressing tab once will complete the command for you. So how do you know to use one tab or two? Rule of thumb is to hit tab once, if nothing comes up, try twice.

You can even use the completion feature for directories, too! Try using the `cd` command to go into the **closet** directory, but instead of typing `closet`, type:

```
joe$ cd clo (press your tab key after o)
```

Notice how the directory will automatically complete for you! The same rule applies here with the tab key: if there's only one result, one tab press will do, but if there's more than one, you need to hit tab twice.

Customizing Your Prompt
OK, it's time to Pimp that Prompt! Well, customizing your prompt isn't *that* cool, but if you want to think of it that way, then go for it!

In order to change your prompt, you have to set an environment variable. It isn't as hard as it sounds.

If you're using the `bash` shell, here's how you change your prompt:

```
joe$ PS1="My New Prompt> "
My New Prompt>
```

Changing your bash prompt

If you’re using another shell, like `tcsh` or `csh`, here’s how to change your prompt:

```
joe$ set prompt=”My New Prompt> “
My New Prompt>
```

Changing you csh or tcsh prompt

Here are two websites that have more examples on prompt customization:

- http://www2.linuxjournal.com/article/3215
- http://www.freebsddiary.org/prompt-tcsh.php

That’s all for this advanced lesson!

Spells Learned in Chapter 3

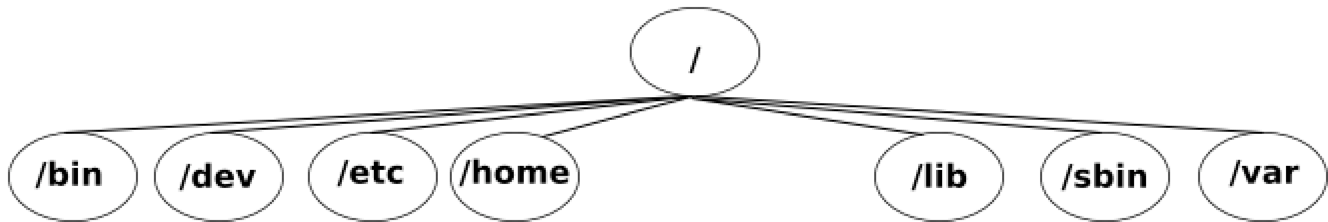
Spell	What it stands for	What it does
`touch [name]`	touch	Creates a blank file
`nano`	Nano	One of several text-editors
`pico`	Pico	Another text-editor
`cat [file1] ([file2])..`	Concatenate	Sends the output of one or more files to the screen
`less [file]`	Less	Allows you to read a file with many pages easily
`more [file]`	More	Allows you to read a file with many pages easily
`cp [src] [copy]`	Copy	Creates a copy of a file or directory
`mv [src] [dst]`	Move	Moves a file or directory to a new location
`rm [file]`	Remove	Removes a file
`rmdir`	Remove Directory	Removes an empty directory
`file [file]`	File	Prints the type of file

Chapter Four

With a spinning head, the Young Mage takes a walk around the Tower; discovers several rooms and floors, but no maze.

We've been making great progress so far, Young Mage! Why don't we take a little break and walk around the Tower for some fresh air. I'll show you all the different rooms that are here. Cast `cd` to teleport to root (`/`) and we'll begin!

If you cast `ls` here, you'll see several different rooms. You won't be learning about all of them, but I'll teach you about the most important ones.



The first floor of the Tower

The first one we'll look at is `/bin`. Inside `/bin` are lots of files -- but not just any kind of files, these are all spells! `/bin` holds all of our most important spells in the Tower. We give it the name bin because bin is short for binaries. A binary is an executable, compiled program that only Unix can understand. If you try casting `cat` on a binary file, you'll get nothing but gibberish back.

Speaking of `cat`, there it is in `/bin/cat`. And `ls` too! In fact, almost all of the spells you have learned so far are here! There are also lots and lots of other spells you haven't even heard of. Unfortunately, I can't teach you all of them, but I'll show a way to learn them yourself, later on.

Next, we'll go into `/dev`. If you cast `ls` in `/dev`, you'll, again, see lots of files. `/dev` stands for devices and these files are device files. They are special files that can interact with hardware -- like your monitor and your CDROM drive and your Keyboard. There are lots of different ways we can use dev files, but we needn't worry about that now.

The next room is `/etc`. `/etc` is our board room where *config* files are stored. Laws and rules that govern how the Tower runs are made and kept here. For example, if the Tower wanted to make a law that kept a certain mage out of the tower because he's been causing problems, we place that law in `/etc`.

`/home` is next. These are the dorms of the Tower, as you already know. Each mage who lives here has a dorm room in `/home`. And there we can see yours!

> Tip:
>
> If you are using Mac OSX, you will see `/Users` instead of `/home`.

Now we come to `/lib`, our Library. Sometimes when we cast spells, they need to look up some information on how to cast them properly. Yes! Spells are smart enough to do that! Even while they're being cast, they can still make a quick trip to the Library. All of this happens without the mage knowing!

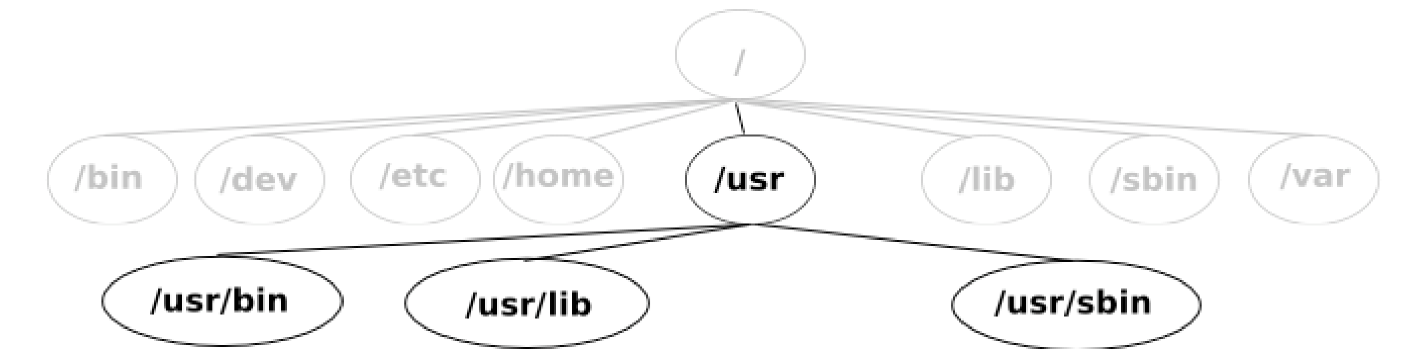
> Tip:
>
> There is no `/lib` in Mac OSX, but there is `/usr/lib`.

The next room is `/sbin`. `/sbin` is another spell directory, but these are important spells that only archmages should use. Some have shields that

prevent you from casting these spells, but others don't. Even so, if you try casting one of the spells in `/sbin`, chances are it won't work correctly.

The last room we'll look at on the first floor is `/var`. `/var` stands for variable and it's where we keep things that change all the time. For example, your mail is usually stored on var. Another important thing kept in `/var` is the `log` room. The `log` room holds all the histories of the Tower so we can go back and reference them later. Mages entering the Tower and spells being cast are two of the many things kept track of in `log`.

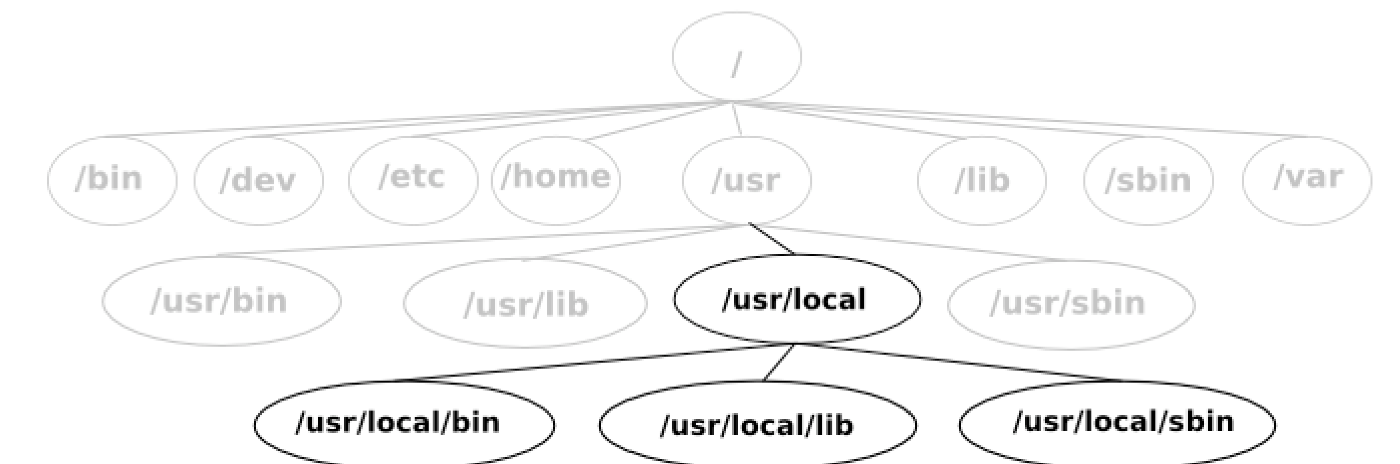
Now we come to one of the staircases in the Tower. This staircase is called `/usr`.



The second floor of the Tower

At the top `/usr`, we see a lot of directories and most of them have the same name as on the first floor! We see a `bin`, a `lib`, an `sbin`, and sometimes we'll even see an `etc`. These rooms all have the same purpose as the ones on the first floor, only now *everyone* in the Tower should be able to use the rooms here. Since the rooms on the first floor can be really important, shields are sometimes put in place to protect them.

There's a second staircase here called `local`.



The third floor of the Tower

At the top of `local`, we, once again, see rooms with the same names as the first and second floors. The `local` rooms are usually customized for the Tower itself -- the archmages can create whatever they want here. It's an unwritten mage rule that says you should try to keep your Towers first two floors similar, but you can use the third floor however you want. So don't be surprised if you ever visit another Tower and see that the third floor looks completely different!

That's it for this lesson, mage. No exercises tonight, but if you feel like, take another walk around the tower and see all the new directories you learned about!

Chapter 4 Advanced Lesson

You just learned about the different floors to the Tower. While our Tower only has three floors, others can have as many as they need. These floors, as you know, are nothing but simple subdirectories, but some of them can be known as *partitions*. This is what we'll cover in this advanced lesson.

Raging Rapids

Let's say you wake up one morning, grab a cup of coffee, and hear a low rumbling. You open your front door only to see a huge tidal wave heading right toward your house. You slam the door shut and start running upstairs. The water breaks in and instantly floods the whole first floor. It starts coming up to the second floor, so you run up to the attic. Before it reaches the attic, the water, thankfully, stops rising.

Now imagine if you had a water-proof hatch going to your second floor. When you climbed through it, you sealed the hatch and finished your cup of coffee with confidence. When the water rose up to the second floor, the hatch prevented anything from coming in -- keeping all the water on the first floor.

The hatch can be thought of as a *partition*. Partitions are invisible hatches in the filesystem that keep things from flooding over.

What could flood over? Disk space! Let's say a friend of yours (never *you*, of course) accidentally sign up for a mailing list that turns out to be spam. Now yo-- your friend -- has spam coming in every 5 minutes! Well the spam just keeps coming in and taking up more and more disk space on the computer. Later on, after working on your research paper for the past four hours, decide to finally save it. When you hit "save", an error pops up saying "Out of Disk Space"! That's because all that spam hogged it. If you were to use partitions, you would not have run into that problem. That invisible hatch would kick in and say "Hey! You stop right there! You've used up enough space, buddy, settle down!"

Viewing Partitions

You can use a command called `df` to see the partitions on the computer. Here's an example of how `df` works:

```
joe$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda3               256665    157316     86096  65% /
/dev/sda1               101086     10982     84885  12% /boot
/dev/sda8              1035660     33508    949544   4% /home
/dev/sda5              3621020    614696   2822384  18% /usr
/dev/sda6              2063504    362340   1596344  19% /var
joe$
```

Viewing partitions with df

There are several columns to `df`.

1. **Filesystem:** These are the partitions. Sometimes they're known as slices, too, because a partition slices up a hard drive into pieces. You can see they use /dev files because, as you learned during the Tour, /dev files access hardware -- in this case, your hard drive.
2. **Available Space:** This shows you how much total space is available on the slice in Kilobytes.
3. **Used Space:** How much space has been used in Kilobytes.
4. **Available:** How much space you have left. (Simply Available minus Used.)
5. **Used Percentage:** Just a percentage representation of how much space is used.
6. **Mounted On:** The directory of the partition.

Since nobody reads in Kilobytes these days, you tell `df` to use a more readable format by specifying the -h option (which stands for Human Readable).

```
joe$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda3             251M  154M   85M  65% /
/dev/sda1              99M   11M   83M  12% /boot
/dev/sda8            1012M   33M  928M   4% /home
/dev/sda5             3.5G  601M  2.7G  18% /usr
/dev/sda6             2.0G  354M  1.6G  19% /var
joe$
```

Using df with the -h option

Look a little easier to understand?

Another command that you can use to view partitions is called `fdisk`. but only archmages should use it since it can break your hard drive if used the wrong way.

What Partitions Do

Like I mentioned before, partitions help disk space from overflowing to other directories. Take a look at the `/home` partition in the `df` example. Right now, only 4% of the space is being used, but what if someone saved a file that took all available 928 megabytes up? Then it would look like this:

```
joe$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda3             251M  154M   85M  65% /
/dev/sda1              99M   11M   83M  12% /boot
/dev/sda8            1012M   1012M  0M 100% /home
/dev/sda5             3.5G  601M  2.7G  18% /usr
/dev/sda6             2.0G  354M  1.6G  19% /var
joe$
```

A partition with no space left

As you can see, 100% of the space is used. Now whenever someone else tries saving a file, they will get an error. But notice the other partitions still have free space. This ensures that other things on the computer can still function properly even though /home is full. Going back to the flood example, the `/home` room is full of water, but the waterproof hatch isn't letting any water out to the other rooms.

There are other uses for partitions, as well, such as security, mounting, and quotas. One security example is the ability to tell a partition that no commands that are located in the directory are allowed to be executed. With mounting, you are able to effectively move a whole partition (such as `/home`) to a new hard drive (for more space or if your hard drive dies, for example). And finally, with quotas, you're able to restrict the amount of disk space each user is able to take up.

You don't need to use partitions when building your own Unix system -- but if you're building a computer for important work, it's recommended that you do so.

Spells Learned in Chapter 4

Spell	What it stands for	What it does
`df`	Disk Filesystems	Shows the partitions and filesystems of the computer

Chapter Five

The Young Mage learns that not everything in the Tower is free to use because of both privacy and protection.

Earlier, I mentioned how some of the spells in `/sbin` have shields to keep you from casting them. Actually, all spells, files, and directories have some kind of shield on them! Also, not only can shields be used to prevent you from casting spells, they can also be used to prevent you from reading files and writing information to files.

Tip:

Shields are also called permissions.

Now you'll learn how to read those shields and even place some on your own files!

Before you begin learning about shields, I need to cover a few basic things. In the Tower of Nix, you have your name (which you can find by casting `whoami`). This is also called your *login* name or *username*. When you create a file or directory in Unix, that name is now known as the *owner* of that file or directory.

Next, all mages in the Tower belong to *groups.* You can belong to several groups or even just one group -- but you always have to be in at least one. To see what groups you are in, you can cast the `groups` spell.

```
joe$ groups
joe appserveradm appserverusr admin
joe$
```

Casting the groups spell

As you can see, I am a part of 4 groups: `joe`, `appserveradm`, `appserverusr`, and `admin`.

Like the name indicates, groups are used to group users together. When you are a part of a group, you have access and premission to everything all the other members of the group have. It's like being a part of the Tower's swim team -- all members are allowed to use the pool after hours.

Finally, you need to know about 3 little letters: **R**, **W**, and **X**.

Letter	What It Means	What It Does
R	Read	Allows you to Read the file
W	Write	Allows you to create, edit, and delete files
X	Execute	Turns the file into a spell and allows you to cast it

Now let's get started with shields! First use `cd` to teleport to `/bin`. Once you're there, cast `ls -l`. `-l` is another spell modifier that will show you lots of details about the files -- including the shields.

```
joe$ ls -l
-rwxr-xr-x   1 root  wheel  581636 Mar 20 18:39 bash
-r-xr-xr-x   1 root  wheel   14380 Mar 20 18:38 cat
-r-xr-xr-x   1 root  wheel   19108 Mar 21 17:48 cp
-r-xr-xr-x   1 root  wheel   18964 Mar 21 17:49 mv
joe$
```

Casting ls -l

The -l option gives us nine columns of information. Let's go over them (they aren't that bad, don't worry).

1	2	3	4	5	6,7,8	9
-rwxr-xr-x	1	root	wheel	581636	Mar 20 18:39	bash

1. The Shield
2. Number of Links
3. The Owner of the file
4. What Group the file belongs to
5. The Size of the file in bytes
6. The Month the file was created
7. The Day the file was created
8. The Time the file was created
9. The name of the file

So there you have it -- all nine columns. Not so bad, right?

Let's go over the first column a little bit more. I'm sure it looks really goofy, but in a few minutes, you'll understand everything about it!

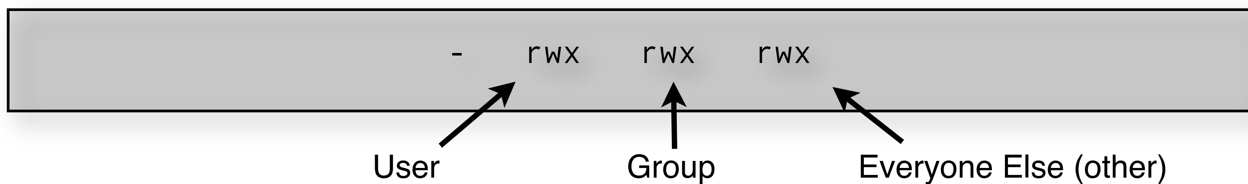
First, let's break it up into some sections like this:

```
-   rwx   rwx   rwx
```

Breaking the shield up into sections

Now we've got four sections to look at. Ignore that first dash for now; I'll tell you about it later. Take a look at the other three sections. They each have three letters -- the same three I told you about a little bit ago. Coincidence? I think not!

Now take another look:



The sections are for the Owner (user) of the file, a Group, and Everyone Else (which we'll call Other). We call this group of three UGO: User, Group, Other.

Each section explains what shields are turned on for the owner, group, or other. If a letter is there, that means that action is enabled. If it's just a dash, that means the shield is preventing that action.

Let's take a look at one of the files from `/bin` again:

```
-rwxr-xr-x   1 root  wheel  581636 Mar 20 18:39 bash
```

The bash file from /bin

Like I said before, ignore that first dash and just take a look at the other part. In your head, break it up into the three sections like I did above. So now you have:

```
-    rwx    r-x    r-x
```

Finally, let's put all the pieces together: The owner of the file is allowed to read the file, write to the file, and cast (execute) the file as a spell. The group can only read and cast the file. The same goes for Other. Who in the world is Other? Other is everyone else in the world who is not the owner and who does not belong to the group.

See? It's not so hard after all! Just remember to break it up into three pieces. Then if there is a dash, the shield is preventing you from that action, but if the letter is there, you can do that action.

Finally, it's time to learn what that mysterious dash at the beginning is all about. The dash tells you what the file is. Is it just a plain file? Is it a directory? Take a look for yourself:

```
-rwxr-xr-x   1 root  wheel  581636 Mar 20 18:39 bash
drwxr-xr-x   41 root  wheel  1394 Jul 13 18:22 bin
```

Different types of files and directories

`bash` starts with just a plain dash. This means that it's a normal, everyday file. However, `bin` shows a `d` -- that means it's a directory. So now you can cast `ls -l` along with `ls -F` to see if something is a directory, too!

Placing Shields On Files

Now you'll learn how to actually put shields on files. To do that, we use the `chmod` spell. `chmod` stands for Change Mode, but you can actually think of it as Change Shield. The only requirement of using `chmod` is that you have to be the owner of that file. We can't just let everyone go around and change the shields on anyone else's files!

Go into your spell closet and cast `ls -l` on your shelf.

```
-rw-r--r--   1 joe  joe  25 Jul 15 20:35 shelf
```

Casting ls -l on the shelf

As you can see, the shields that are currently on the file say that the owner can read and write, and the group and everyone else can read. Let's make it so the group `joe` can write to the shelf as well:

```
joe$ chmod g+w shelf
joe$ ls -l
-rw-rw-r--   1 joe  joe  25 Jul 15 20:35 shelf
joe$
```

Casting chmod on the shelf

`chmod` looks like a really funny spell. You'll notice that it takes two arguments. The second argument is the name of the file we are changing the shields on. The first argument looks strange. `g+w`? What's that mean?

Remember UGO? User, Group, Other -- the three parts of the shield. That's what the `g` means: group. The plus sign means that you are adding to the shield, and `w`, like you learned before, means write.

Now say it to yourself: group add write. And that's exactly what you did! You added write to the shield! Now anyone who belongs to the `joe` group can write to the shelf.

Taking away from the shield is just as easy:

```
joe$ chmod g-w shelf
joe$ ls -l
-rw-r--r--   1 joe  joe  25 Jul 15 20:35 shelf
joe$
```

Using chmod to take away from the shield

Instead of using the plus sign, you use the minus sign.

Here are a couple other examples:

```
joe$ chmod u+rwx shelf
```

Looks a little strange at first, but if you read it and say it silently to yourself, it all makes sense. To **User** we are **Adding Read**, **Write**, and **Execute**. `u+rwx`.

```
joe$ chmod ugo+rx shelf
```

This second example is just as easy: `ugo+rx`. To **User**, **Group**, and **Other**, we are **Adding Read** and **Execute**. This example could also be written as `a+rx` where `a` stands for All.

```
joe$ chmod u+x,g+rw,o-r shelf
```

Finally, the third example looks different than the others, but it's still just as easy: `u+x,g+rw,o+x`. To **User** we are **Adding Execute**, to **Group** we are **Adding Read** and **Write**, and to **Other** we are **Taking Away Read**.

Using the Number Method to Place Shields on Files
Sometimes typing all of those letters can get tedious -- especially once you get the hang of using `chmod`. There's a faster way to use `chmod` that uses numbers.

To understand the numbers, first we have to convert r, w, and x to a number:

Letter	R	W	X	None
Number	4	2	1	0

If we wanted to put **Read** and **Write** on a shield, we add 4 and 2 to get 6. **Execute** and **Read** would be 5. And all three would equal 7.

Next, you group three numbers together for a UGO combination. For example, if you wanted **User** to have **Read** and **Write**, **Group** to have **Read** and **Execute**, and **Other** to have **Read**, it would be 654. 6 = **rw**, 5 = **rx**, and 4 = **r**.

Finally, to cast chmod with the numbers, you would do this:

```
joe$ chmod 654 shelf
joe$ ls -l
-rw-r-xr--   1 joe  joe  25 Jul 15 20:35 shelf
joe$
```

Casting chmod with numbers

And you can see that the shield changed to what we wanted!

Another difference between using letters and numbers with chmod is that with letters, you can use + and - to add and subtract actions. With numbers, you can't do that, you have to start over each time.

So right now, your shelf has **Read** and **Execute** for the group shield. If you wanted to add **Write** to it, you would have to add up **Read**, **Write**, and **Execute** to get 7.

Also, there is no way to just change the **Users** shield and not the **Group** and **Other** when using the numbers. You always have to cast `chmod` with the three numbers. If you don't use the three groups, something like this will happen:

```
joe$ chmod 4 shelf
joe$ ls -l
-------r--   1 joe  joe  25 Jul 15 20:35 shelf
joe$
```

Casting chmod with only one number

See how it erased everything but **Read** for **Other**? That's because using a single 4 is just like using 004 -- if you don't specify one of the three numbers, Unix will use a 0 instead. So there's pros and cons to using the letters and numbers. Use either method you feel most comfortable with!

What a lesson! Talk about some really obscure stuff! But don't worry, Young Mage, the more you work with shields and permissions, the more comfortable you'll be with them.

And if you're wondering about the **Execute** part of shields and when you'll need to use it, don't worry about it right now. Adding the **Execute** action to a shield is only done by mages who create their own spells. One day you'll be creating your own, but for now, just understand that **Execute** can turn a file into a spell!

Don't forget to try the exercises!

Exercises for Chapter 5

1) Casting two's

Go into your spell closet and cast

```
joe$ chmod 222 shelf
```

What do you think the shield will look like when you cast `ls -l`?

What happens when you try to cast `cat` on your shelf now?

2) Casting four's

Go into your spell closet and cast

```
joe$ chmod 444 shelf
```

What do you think the shield will look like when you cast `ls -l`?

What happens if you cast `nano` on it and try to save your work?

3) Creating your own Shields

Go into your spell closet. Use `chmod` to make it so the shelf:

Has **Read** and **Write** on **User** and **Group** but nothing on **Other**
Has **Read** and **Execute** on **User**, **Write** on **Group**, and **Execute** on **Other**
Has **Write** and **Execute** on **User**, **Nothing** on **Group**, and **Read** and **Execute** on **Other**

Chapter 5 Advanced Lesson

By using `ls -l`, you're able to see a lot of details about a file. One piece of information is what *kind* of file the file is. We'll be going over the different types in this advanced lesson.

Types of Files
Remember from earlier, that the first dash in the permission set specifies the type of file.

```
-rwxr-xr-x   1 root  wheel  581636 Mar 20 18:39 bash
drwxr-xr-x   41 root  wheel  1394 Jul 13 18:22 bin
```

Output of ls -l

Here, we see that the first entry, `bash`, has just a dash. Therefore, we know it's just a plain file. The second entry, however, has a `d` which stands for directory. There are a few other types besides just plain file and directory.

Letter	What it is
-	Plain File
d	Directory
l	Link
p	Pipe
b	Block Device
c	Character Device

Plain File
A Plain File is just that -- a plain file. It can hold any type of information: from music to binary to just text. A file can also have the `x` action set on it which turns it into a spell.

Directory
According to Unix, a Directory is nothing more than a file -- but a special file. I know it sounds like a catch-22, but it's true. If a Directory is just a file, then what's inside it? Special information about the files contained in the directory. Simply speaking, a Directory is kind of like a phone book. When you want access to a file, the shell will consult the Directory on how to reach the file. This is just another way the Shell abstracts the user from the strangeness of Unix -- by making it seem like we're in an actual directory, but in fact, the Shell is just reading information from a file.

Link
A Link is a file that points to another file. When you move, you can leave a change-of-address so any mail that goes to your old address gets forwarded to your new one. This is kind of like a link.

Here is an example of how a link looks to `ls -l`:

```
joe$ ls -l pico
lrwxrwxrwx  1 root root 9 Jul 22 08:51 pico -> /bin/nano
joe$
```

Example of a Link

There's a couple things to notice about the Link:

- It has the `l` at the beginning of the permissions
- It has rwx for User, Group, and Other
- It shows an arrow that points to the real file
- Its size is the number of characters of the real file (`/bin/nano` is 9 characters which equals this files size).

If you were to run the command `pico`, the Shell will instead run the command `/bin/nano` without you knowing.

You can create links with the `ln` command. There are two types of links you can create, hard and soft (or symbolic). A hard link is an exact clone of the file. It will not show the arrow like in the example because the hard link thinks it's a real file, not a pointer to another. A soft link is a separate file whose only job is to point to the correct file (think forwarding address).

Here's how you create a hard link:

```
joe$ ln file1 file2
joe$ ls -l file1 file2
-rw-r--r--   2 joe  joe  0 Jul 31 18:58 file1
-rw-r--r--   2 joe  joe  0 Jul 31 18:58 file2
joe$
```

Creating a hard link

See how both files think they're separate, but they're really exact twins.

Here's how you create a soft link:

```
joe$ ln -s file1 file2
joe$ ls -l file1 file2
-rw-r--r--   2 joe  joe  0 Jul 31 19:01 file1
lrwxr-xr-x   1 joe  joe  5 Jul 31 19:01 file2 -> file1
joe$
```

Creating a soft link

To create a soft link, you just use `ln` with the `-s` option. Notice how file2 has an arrow pointing to the file it links to. Plus it has the `l` in the permissions set.

Why would you use one over the other? A hard link takes up no extra disk space but can't span file-systems (see Chapter 4 Advanced Lesson). Conversely, a soft link takes up a few bytes of memory, but it can span filesystems.

Pipes

In mini-golf, some of the holes have a pipe the ball goes down to get to another part. Unix Pipes work the same way -- they connect two programs together to share information.

Block Devices and Character Devices

Both of these are device files -- meaning they interact with hardware. The difference is that a Block Device will read information in blocks while a Character Device reads information in a stream.

What does it mean by "reading in blocks"? Think about putting your hand under running water. If you just let the water pour over your hand, that is like a Character Device and streams. However, if you cup your hands until they are full and then let the water fall, that is a Block Device -- it reads a bunch of stuff then lets it all go at once.

An example of a Block Device would be your hard drive and an example of a Character Device would be a modem.

That covers just about everything with file types in Unix!

Spells Learned in Chapter 5

Spell	What it stands for	What it does
`chmod [shield] [file]`	Change Mode	Changes the Shield on a file
`ln [file1] [file2]`	Link	Links two files together

Chapter Six

The Young Mage learns more details about casting spells: Paths are revisited; Controlling the flow of output; and a cute calendar.

Casting Spells With Absolute Paths
During our tour of the Tower, you saw a lot of spells in the `/bin`, `/usr/bin`, and maybe even the `/usr/local/bin` directories -- but how do you cast them? Up until now, you've just been typing in a spell name and hitting enter and it magically works. Now you'll learn how to cast any of the spells in any directory.

First, try casting this:

```
joe$ ls -l /
```

Casting ls

You've cast similar things before. Nothing new here. But how about this:

```
joe$ /bin/ls -l /
```

Casting ls with an Absolute Path

It did the same thing! Except we used an absolute path with `ls`. How does that work? With something called the *Path Variable*.

A *variable* is something that holds information. It's called variable because the contents can vary. A bucket can be considered a variable because it can hold water along with almost anything else -- it's not stuck to one thing.

The *Path Variable* is a special variable that holds some paths for us. We can see what paths are stored in the Path Variable like this:

```
joe$ echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin
```

Viewing the Path variable

`echo` is a simple command that prints things out for us. Variables always start with a `$` -- that's the way we know `$PATH` is a variable. Don't get this `$` confused with the one in the prompt, though.

What `echo` returned looks kind of confusing at first. But if you break it up by the colon (**:**), it looks a little familiar:

```
/bin    /sbin    /usr/bin    /usr/sbin
```

Notice something? They're paths! The Path Variable stores paths with colons between them. Yeah, I know, big deal -- but here's where it all comes together. Whenever you cast a spell, the Path Variable will go through it's list of Paths and see if the spell is inside that directory. So when you cast `ls`, it kind of looks like this:

```
joe$ ls
hmm.. is that /bin/ls? Yep! I'll use /bin/ls and you won't notice!
```

That was a simple example. Here's something more involved. There is a spell called `cal` that will print a nice calendar to your screen (go on, try it, you know you want to!). Here's what happens now:

```
joe$ cal
hmm.. is that /bin/cal? No, no cal there.
what about /sbin/cal? Nope. not there, either.
/usr/bin/cal? Yes! There it is, I'll cast that one!
```

So with the Path Variable, these two spells are just the same:

```
joe$ cal
joe$ /usr/bin/cal
```

What happens when the Path Variable can't find a match at all? It will go through it's list of paths, see that the spell is not located in any of those directories and tell you it can't find a match:

```
joe$ oogabooga
-bash: oogabooga: command not found
```

What if you want to cast a spell that is not inside one of the directories Path knows about? Then you have to specify the absolute path to the spell or else you will get a `command not found` error like above.

```
joe$ /super/secret/directory/Fireball
```

Running the Fireball command which is not in the path

So there you have it, now you can look through all the bin directories and cast any spell you want. Unfortunately, I can't teach you about every single spell -- you're going to have to learn about most of them yourself. It's not that hard, though. The archmages have created instruction books for almost every spell in the Tower. Now you'll learn how to read them!

Reading the Instructions for Spells

The archmages knew they would never have enough time to teach all the other mages every single spell in the Tower, so they wrote instructions on how to do it. We call these instructions *man pages* or *manual pages*.

In order to read the man page, you cast a spell called `man` like this:

```
joe$ man ls
```

Casting the man spell

Once you press enter, you'll see a lot of text appear on your screen. If you look close enough, you'll see that `man` has actually used either the `more` or `less` program to show you the text.

```
less — ttyp2 — 78x15
LS(1)                    BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLPRTWZabcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.
:
```

A Manual Page

The man page will tell you everything you want to know about a spell: how to use it, what options are available, and even similar and related spells! Unfortunately, man pages can be pretty dry and boring -- but the information is there, so just bear with it!

Searching for Spells

If you're not sure about the spelling of a spell, there's an easy way to search for it using the `apropos` spell. apropos takes a word as an argument and will print out all of the available spells related to that word or with that word in it's spelling.

```
joe$ apropos whoami
bpwhoami(1)              - print the output of a bootparams
                           whoami call
ldapwhoami(1)            - LDAP who am i? tool
whoami(1)                - display effective user id
zipgrep(1)               - search files in a ZIP archive for
                           lines matching a pattern
```

The apropos spell

Using Grep
Another important spell in Unix is called `grep`. `grep` should have been called grab since that's what it does -- grabs text. Remember the shelf you made that stores all the spells you know in it? We'll use it to practice casting `grep` on it.

You know that with spells like `cat`, `more`, and `less`, you can view the contents of the whole file. But what if you only wanted to see spells that had the letter "d" in them? `grep` can do that for us!

Tip:

`grep` actually stands for

"Global Regular Expression Print" -- what a mouthful!

```
joe$ cd /Users/joe/closet/spells
joe$ grep d shelf
mkdir
cd
joe$
```

Casting grep

As you can see, `grep` only printed out the spells with the letter "d" in them -- just like we wanted!

`grep` is a really easy spell to cast and it works even better with *pipes*.

Pipes: the Coffee Filters of Spells
Have you ever seen that vertical bar on your keyboard? The one above your enter key? This guy: **|** ? Ever wonder what it was? It's called a *pipe.* What good is it for? Lots! What does it have in common with a coffee filter? Lots!

Everyone knows what a coffee filter is used for: it's that thing you put in your coffee machine so only the coffee gets to the pot and not the ground beans. Pipes do the same thing -- they only let things you want through.

Say, for example, you cast

```
joe$ ls /bin
```

Casting a simple ls spell

You get a lot of spells returned. What if you didn't want to see all of them? What if, just like the shelf file, you only wanted to see the ones that had the letter "d" in it? We could use both, `grep` and a pipe. Don't believe me? Just watch:

```
joe$ ls /bin | grep d
```

Casting ls with a pipe and grep

Now you only see spells that have the letter "d" in them! How's that work? Well, the `|` is our coffee filter and the `grep d` part is telling the filter what is OK to let pass. The result is what you see in your terminal. Pretty cool, huh?

Redirection

Sometimes when you cast a spell, the stuff it returns is really useful and you might even want to save it for later. It would be really great if you could save it to a file and just use `cat`, `more`, or `less` on it whenever you want. But how? With *redirection.*

Redirection is the traffic cop of Unix. It tells the spell output where to go -- like to a file. If there's no cop to redirect the output, it just goes to your screen.

Remember that cal spell I mentioned earlier that prints a calendar? If you didn't try casting it, here's what it looks like:

```
joe$ cal
     July 2005
 S  M Tu  W Th  F  S
                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
joe$
```

Casting the cal spell

What if you wanted to save that calendar to a file called **july.txt**? You could use redirection like this:

```
joe$ cal > july.txt
```

Saving the output of cal to july.txt

Notice the `>`. That's the redirector. It looks like an arrow and that's exactly how you can think of it, too. It's the traffic cop saying, "OK, all output go *this* way: -->", and it gets placed inside the **july.txt** file.

Now if you cast `ls`, you'll see that a file called **july.txt** is really there, and even more, if you cast `cat` on it, the month of July will show up on your screen!

You can even send text the other way, too. Let's say you wanted to use `grep` to only show lines of the calendar that have a number "8" somewhere in them. You could cast a spell like this:

```
joe$ grep 8 < july.txt
 3  4  5  6  7  8  9
17 18 19 20 21 22 23
24 25 26 27 28 29 30
joe$
```

Importing the july.txt file into grep

Now we're feeding the contents of **july.txt** to the `grep` spell. Unfortunately, this is the exact same as casting

```
joe$ grep 8 july.txt
```

Casting grep on july.txt

So feeding text into a command really doesn't come in handy as much as sending text to a file. Oh well, just more ways to cast spells!

Wildcards

In poker, a wild-card is a card that can be anything you want. Spells have wild-cards, too -- two in fact. The spell wildcard are a star (*****) and question mark (**?**).

The question mark means *any one character*. So if you wanted to see all spells in the `/bin` directory that start with an `l` and have any one character, you would cast:

```
joe$ ls /bin/l?
/bin/ln   /bin/ls
joe$
```

Casting ls with a ? wildcard

As you can see, we have two spells: `ls` and `ln`.

Now what if you wanted to see all the spells that start with `l` no matter how many characters are in it's name? You could just keep using more and more question marks, but the easier way is to just use the star. The star wildcard means *any character and any number of characters*.

```
joe$ ls /bin/l*
/bin/launchctl  /bin/link       /bin/ln         /bin/ls
joe$
```

*Casting ls with a * wildcard*

And now we have many more results -- some with more than two characters in their name.

You can even use two stars to find the spells with `l` anywhere in their name.

```
joe$ ls /bin/*l*
/bin/kill       /bin/launchctl  /bin/link       /bin/ln
/bin/ls         /bin/sleep      /bin/unlink
joe$
```

Casting ls with two wildcards

This is the exact same as casting

```
joe$ ls /bin | grep l
/bin/kill       /bin/launchctl  /bin/link       /bin/ln
/bin/ls         /bin/sleep      /bin/unlink
joe$
```

Casting ls and using grep to find the l character

Unix has a lot of other wildcards, but they're called *Regular Expressions*. Regular Expressions can get really complicated and there are whole books devoted just to them!

That ends this lesson, Young Mage. You learned a lot about working on the command line: casting more spells, grepping, redirecting, reading instructions, and wild-cards! It was a lot of information in a short amount of time, so make sure you do the exercises for some practice!

Exercises for Chapter 6

1) ECHO Echo echo

There are lots of things you can do with the `echo` spell. Try these out:

```
joe$ echo “HI”
joe$ echo “HI” > hi.txt
```

Now notice you have a file called hi.txt. What’s inside it?

2) More variables

There are a lot of other variables that exist besides the path variable. You can find them by casting

```
joe$ env
```

`env` stands for *Environment*. All the variables in your environment affect how things get done -- just like how the weather affects how you get things done in the real world.

How can you cast echo to show the contents of these variables? Here’s a hint: Remember

```
joe$ echo $PATH
```

3) More redirection

In **Exercise 1**, you created a file called **hi.txt** by casting echo and using a redirect. What happens to the contents of **hi.txt** if you cast

```
joe$ echo "Hello, there!" > hi.txt
```

Now what happens if you cast

```
joe$ echo "How are you today?" >> hi.txt
```

What do the `>>` arrows do as opposed to the `>` ?

4) Learn a new spell

Read the man page for the `wc` spell. What does it do?

5) Double Pipes

You know that by casting

```
joe$ ls /bin | grep o
```

the output you see will be all the files in /bin that contain the letter o. Now what happens if you cast

```
joe$ ls /bin | grep o | wc
```

And why?

Chapter 6 Advanced Lesson

Unix gives you the ability to channel information through several different streams. These streams are also known as *file descriptors* and they're what you'll learn about in this Advanced Lesson!

The Three Basic File Descriptors

Unix has the ability to support 10 file descriptors numbered 0 to 9, but you'll only use the first three for the majority of the time. The basic three are standard input, standard output, and standard error. Some more information is shown in the table:

File Descriptor	Short Name	Descriptor Number
Standard Input	STDIN	0
Standard Output	STDOUT	1
Standard Error	STDERR	2

Standard Output

Whenever a command prints something to your screen, STDOUT is being used. You may redirect where the output goes by using pipes and redirectors.

```
joe$ echo “Hello” > newfile.txt
```

Redirecting STDOUT

You've seen this example in the previous lesson -- it's just a simple redirect. But what you haven't seen yet is that there is an implied, hidden File Descriptor 1 embedded in the command:

```
joe$ echo “Hello” 1> newfile.txt
```

Redirecting STDOUT with the implied File Descriptor

If you take a look at the contents of `newfile.txt`, you'll notice there's nothing different about it than without using the 1. When you use a simple `>` in Unix, it is just the same as `1>`.

Standard Error

If a command responds with an error message, it's using the STDERR channel. You may also redirect STDERR, but an extra step is needed.

Try saving the error message of this command to a file:

```
joe$ ls doesnotexist > newfile.txt
ls: doesnotexist: No such file or directory
```

Attempting to redirect STDERR

If you look at the contents of `newfile.txt`, you'll notice it's empty -- it did not redirect the error message. Why not? Because errors use File Descriptor 2. In order to properly save the error message, you'll need to do this:

```
joe$ ls doesnotexist 2> newfile.txt
```

Redirecting STDERR

Now if you look at the contents of `newfile.txt`, you'll see the error message. Remember, when using `>`, 1 is by default and anything else has to be explicity typed.

Standard Input

By default, anytime a file receives more information, it's using File Descriptor 0. Just what do I mean by "receives more information"? Receiving information can be in the form of adding text to a file, reading command line arguments, or having information piped to a command.

The tricky part about STDIN is that it's a part of almost every command. Using the previous STDERR example, the error message is the STDERR of the `ls` command, but becomes the STDIN of `newfile.txt`. Anything that accepts input is STDIN.

When using `>` to redirect output, you can think of the left side as STDOUT or STDERR and the right side as STDIN. Of course, it's vice versa with the `<` redirector -- the left is now STDIN and the right is STDOUT or STDERR!

The same is true with pipes -- the left side is the STDOUT and the right side is STDIN. Using that concept, we can prove that (if programmed correctly) any command that can take a file as an argument, can also take it's input by STDIN. Here's how we can prove it:

You know that you can use the `wc` command to count the number of lines, words, and characters in a file like this:

```
joe$ wc file.txt
```

Using wc with a file as an argument

You can also pipe information to `wc` and it will still count the number of lines, words, and characters just as if you would have specified a file.

```
joe$ cat file.txt | wc
```

Using cat to pipe output to wc[5]

As you can see, the same output is shown. This proves that (almost) any Unix command that can accept a file as an argument can also have information piped to it for input. I say *almost* any command, because when programming Unix commands, you have to follow a standard set of rules to allow this. Most programs do, but there are the odd few that do not conform to this standard way of Unix programming.

Combining File Descriptors

It's also possible to combine file descriptors. You would do this if you wanted to save, both, the input and output to a file. You can combine them like this:

```
joe$ cat newfile.txt > newerfile.txt 2>&1
```

Combining STDERR and STDOUT

What this does is redirect all STDERR to STDOUT. Since STDOUT is being redirected to `newerfile.txt`, so does all of the error messages (if any exist). Yes, I know it looks goofy to have the combination at the end of the command!

Disabling All Output

Sometimes you'll want to run a command and just not see *any* output -- whether it be STDOUT or STDERR. In that case, you can redirect everything to a special device file known as `/dev/null`:

[5] Using cat in this form is known as a Useless Use of Cat and should be avoided. It's used here for visual purpose only.

```
joe$ cat newfile.txt > /dev/null 2>&1
```

Redirecting all output to /dev/null

`/dev/null` is your own personal blackhole in Unix. Anything that goes to it just vanishes.

That just about covers everything with File Descriptors and Redirecting!

Spells Learned in Chapter 6

Spell	What it stands for	What it does
`echo`	Echo	Prints something to the Terminal
`grep [text] [file]`	Grep	Grabs text you asked for
`cal`	Calendar	Prints a nice calendar out for you
`wc`	Word Count	Counts words for you

Chapter Seven

The Young Mage learns that the Tower is actually full of other Mages and workers; a peek into the Towers registration.

As you saw during the tour, the Tower has a dormitory located at `/home`. You aren't alone here; many other people can be working in the Tower at the same time! You'll learn more about working with multiple mages in this lesson.

Finding Other Mages

First, to see if anyone else is in the Tower with you, cast `who`. This will show you a list of everyone currently in the Tower, plus some other information.

```
joe$ who
joe       console   Jul 20 18:29
joe       ttyp1     Jul 21 16:01
joe$
```

Casting the who spell

The who output for me shows that I'm logged into the Tower twice: once for my main computer and once for my terminal. Being in the Tower multiple times is not something unusual -- all mages can multiply themselves and perform *multi-tasking*.

Next, let's take a look at the central listing of everyone who's registered in the Tower. This registry is the file `/etc/passwd`. You can view your registration by casting grep:

Tip:

Unfortunately, Mac OSX doesn't use `/etc/passwd` or `/etc/group` for normal accounts, but you can still read it with `less` and follow along!

```
joe$ grep joe /etc/passwd
joe:x:1000:100:Joe Topjian:/home/joe:/bin/bash
joe$
```

Viewing your /etc/passwd information

Just like the Path variable, there's a lot of information joined by colons. Let's break it up and go through each field.

1	2	3	4	5	6	7
`joe`	`x`	`1000`	`100`	`Joe Topjian`	`/home/joe`	`/bin/bash`

1. Username (the same as what whoami) shows
2. Your password used to go here, but that was in the old days of the Tower. Now it's located in /etc/shadow and only the archmages can read it.
3. User Identification (UserID or uid). Unix thinks of everything as a number, so it takes your username and makes a unique number that only you have. Just like that number on your Drivers License -- no one else has that number.
4. Group Identification (GroupID or gid). Unix does the same thing for groups, too. Even though you can belong to more than one group, the first group you are in is shown in this forth column.
5. The GECOS field. It can store anything you want. It's usually used for the persons full name or a description of what the account is for.
6. Dorm location
7. Shell[6]

Tip:

GECOS is actually an old term from the General Electric company way back in the 1970's.

All mages are able to read the `/etc/passwd` file to see who else is registered in this Tower. Take a look at it, if you want! Some of the accounts have weird names like `nobody` and `www` and `mail`. These are called *system accounts*. System accounts are the janitors and maintenance people of the Tower -- they take care of everything going on in the background. You'll learn more about them in Chapter 8.

There is another registry, just like `/etc/passwd`, used for groups. It's located at `/etc/group`. The entries there look like this:

```
wheel:x:103:root,joe
```

An entry in /etc/group

[6] For a discussion of shells, see the advanced lesson in Chapter 3

1	2	3	4
wheel	x	103	root,joe

1. Name of the group
2. Group password (rarely used)
3. Group Identification (GroupID or gid). This is the same number as what you'll find in /etc/passwd. In fact, Unix will reference the number found in /etc/passwd to this file in order to get the name that the groups spell reports!
4. Extra members of the group (optional). Each extra member is seperated by a comma.

Ownership
Remember when you first learned about ls -l and you saw how it displayed the owner and group that belonged to the file? There is actually a way to change the ownership and group for files and directories! When you change the owner and group of a file, there has to be a valid entry for the person or group you want to change it to in the /etc/passwd and /etc/group files. Unfortunately, this spell can only be cast by archmages. Though if you're curious, the spell is called chown (change ownership).

Passwords
As mentioned above, there is a file called /etc/shadow that contains all the passwords for each mage in the Tower. Only archmages are able to read this file. And even if you were an archmage, you still couldn't understand what the password was. That's because they're encrypted. A /etc/shadow entry looks something like this:

```
joe:$1$XGpds6rh$WqOLe0a0TyGf9QdgN0x3C1:12846:0:99999:7:::
```

An /etc/shadow entry

We won't go into much detail about this file -- just take a look at the first two fields. The first one is the username, just like in /etc/passwd. The second field is where the encrypted password is. It's just a long string of gibberish.

If you ever need to change your password in Unix, you can do so by casting the passwd spell. When you cast it, it will ask you for your current password, then ask you to type in your new one twice. It looks like this:

```
joe$ passwd
Changing password for joe
(current) UNIX password: (output is not displayed)
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
joe$
```

Changing your password with passwd

Of course your password is never shown on the screen for security reasons!

That's all there is for this lesson, Mage. Not so hard today, was it? In that case, you won't have any problems doing the exercises!

Exercises for Chapter 7

1) Groups

Using the `groups` spell, you can find out what groups you belong to. Grep all of those groups from `/etc/group` to see if you can tell if anyone else belongs to those groups, too!

2) passwd and shadow

What are the permissions on the `/etc/passwd` file that let you read it? Can you write to it? What about the `/etc/shadow` file -- why can't you read it?

Chapter 7 Advanced Lesson

Throughout the book, I've mentioned the concept of an *archmage* quite a few times. Who and what are the archmages? Well, you'll find out in this lesson!

Yet Another Root

The first slash in the filesystem is known as *root.* Root is also the name of a user in the Unix system. Root has the ability to do anything -- he's the most powerful user in the system. You can't lock Root out of any files or directories, nor can you prevent him from running any commands. Root can do *everything* and *anything* in the Unix system -- get it?

By the way, where's Roots home directory? It's not `/home/root`, it's just `/root` -- there's another root for you!

How To Become Root

Although you can just log in to your Unix system as root, there's another way to use the root account - by switching to root while you're logged in. You can do this by using the `su` command. `su` stands for Switch User or Superuser or Substitute User -- take your pick. When you run `su`, you'll be prompted for the root password. If you have not been given this password, then you obviously don't have permission to use the root account!

```
joe$ su
Password: (output is not displayed)
root#
```

Using su to become root

Notice how your prompt has now changed. Normally root is denoted by a # at the end of the prompt. You can also run su with a dash at the end. If you do so, you'll inherit all of Roots specific environment variables (including the Path) -- but if you don't, you'll just keep your own variables. The choice is yours!

You can also change into other users using the `su` command -- of course, you need to be root first. To do that, just specify a dash and the name of a user as an argument.

```
root# su - merrit
merrit$
```

Using su to change to a different user

Some newer Unix systems (like Ubuntu and Mac OS X) that have moved away from the `su` command to a different root-enabling command called `sudo`. sudo allows you to run a single command as root instead of switching completely to root. This way, owners and administrators on the system can restrict certain root-only commands to certain people.

```
joe$ sudo root_only_command
Password: (output is not displayed)
```

Using sudo to run root commands

The Wheel Group
Some Unix systems have a special group called wheel. Only the group members of wheel may su to root and use the root capabilities.

Root's UID
Root has a UID of 0 -- this is the same across every Unix system. You can effectively duplicate the root account just by setting the UID of any other account to 0 in `/etc/passwd`. Keep an eye out for this, though, as it's something regularly done by malicious people.

And that's about it for the Root account!

Spells Learned in Chapter 7

Spell	What it stands for	What it does
`who`	Who	Shows all mages currently in the Tower
`passwd`	Password	Changes your password
`chown`	Change Owner	Changes the owner of a file
`su`	Switch User	Switches you to another account
`sudo`	Superuser Do	Allows you to run root commands

Chapter Eight

A closer look at the inner workings of the Tower; how to put spells on hold; the unfortunate demise of deranged spells.

In the previous lesson, you heard about something called a *system account*. You'll learn all about those and much more in this lesson!

Daemons (not demons!)
If mages did all the work in the Tower to keep it running normally every day, there would be no time to get anything else done! Being the thoughtful people mages are, they decided to create little helpers. These helpers are called *daemons*. Each daemon has a job to do and they only do that job. That's it! Daemons just sit in the tower and wait until they have to do their job.

What kind of work do the daemons do? Lot's! Anything from delivering your mail to serving web pages to cleaning up memory a messy mage left laying around -- they do it all. The Tower would never run as smooth as it does without them.

But just like all the mages in the Tower, the daemons need to be registered in /etc/passwd. That's what all those system accounts are -- they're the daemon registrations.

Can you see daemons working? Sure! Let's find out how!

Processes
Whenever you cast a spell, you create something called a *process*. Makes sense, though, right? A process is an action -- just like casting a spell is an action! You can see all the processes you are running by casting the `ps` spell (which stands for process).

```
joe$ ps
  PID  TT  STAT      TIME COMMAND
  610  p2  S      0:00.05 -bash
joe$
```

Casting the ps spell

Let's look at the different columns.

The PID column is for the Process ID. Remember in the last lesson I said Unix likes to convert everything to numbers? This is another example. All processes in Unix get a unique number.

The TT column displays a location for your terminal. Nothing important to worry about.

The STAT column stands for state. It shows the state of the process. The TIME column shows how much processing time Unix has used on the process. Again, neither of these are important right now.

Finally, the COMMAND field shows you the name of the command. In this case, `ps` shows only one process: bash[7].

[7] Bash is explained in Chapter 3's Advanced lesson

One process is pretty boring, I know. Let's add the `a` option to `ps.` `a` stands for All. It will show us all the spells currently being cast in the Tower -- even other mages' spells!

```
joe$ ps -a
  PID  TT  STAT      TIME COMMAND
  609  p2  Ss     0:00.02 login -pf joe
  610  p2  S      0:00.10 -bash
  660  p2  R+     0:00.01 ps -a
joe$
```

Casting the ps -a spell

That looks a little more interesting! Now we see two more processes. The first, `login`, is the very first spell that was cast when we originally got to our terminal. It was the magic spell that teleported us into the Tower.

The second new process we see is the `ps -a` spell that we just cast!

Finally, let's see how we can take a peek at the daemons. To do that, we'll use `ps` with both the `a` and `x` option. `x` tells the spell that we want to see all the activity in the Tower -- even the stuff the daemons are working on.

So the spell looks like this:

```
joe$ ps -ax
```

Casting ps with the a and x options

The output is way to long to print in the book, so you will just have to see for yourself!

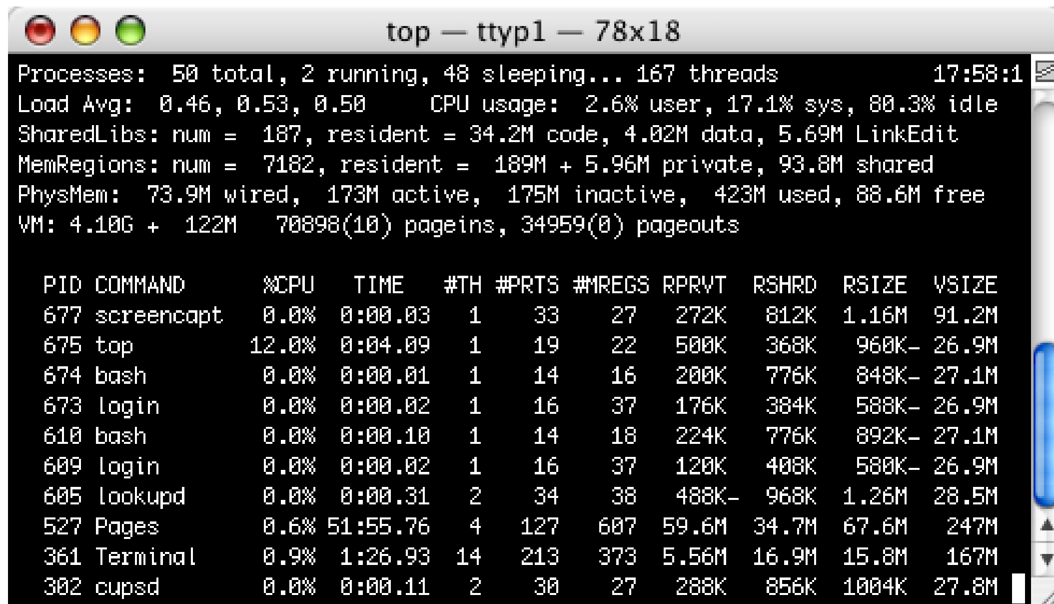
I bet you had no idea all that stuff was going on in the Tower, did you? See all the work daemons do for us!

Now there's one last option we can use with `ps` so we can see the names of the daemons -- the `u` option:

```
joe$ ps -aux
```

Casting ps with the a, u, and x options

There's another spell you can cast to take a look at all the work daemons are doing. This spell will even give you some stats on the Tower (like memory and CPU usage), too! Not only that, but it will constantly update the screen so you can watch the daemons move around! This spell is called `top` and when you cast it, it looks like this:



Casting the top spell

To exit the `top` spell, press `q`.

Casting Lots of Spells

Wouldn't it be great if you could cast more than one spell in a terminal? Well, you can! There are two ways, actually.

The first is known as casting a spell in the background. It's really simple to do. You just add a `&` at the end of any spell, and it goes in the background. Try it out:

```
joe$ top &
[1] 678
joe$
```

Casting top in the background

The number in the brackets ([1]) is known as your job number. The second number is the Process ID or pid. Try casing another spell in the background. This time, use `nano`.

```
joe$ nano &
[2] 684
joe$
```

Casting nano in the background

Notice how your job number went up? The job number shows you how many jobs (or spells) you currently have running. You can use the jobs spell to see a list of all of your currently running jobs:

```
joe$ jobs
[1]-  Stopped                    top
[2]+  Stopped                    nano
joe$
```

Using jobs to see your currently running spells

The plus sign (+) means that job was the last one you placed in the background. The minus sign (-) means it was the second to last one. Third and on get no symbol.

OK, now that you have some spells running in the background, how do you get them back? Well, technically you would be bringing the spells into the *foreground*. So we use the `fg` spell -- which stands for foreground. You cast fg with the job number you want to be brought forth.

```
joe$ fg 1
```

Bringing job number 1 into the foreground

And now `top` will pop up in your terminal! You can press q to exit `top`. Now cast jobs again to see what's left:

```
joe$ jobs
[2]+  Stopped                    nano
joe$
```

Casting jobs

As you can see, nano is left. So go ahead and bring it to the foreground.

```
joe$ fg 2
```

Bringing nano into the foreground

To exit nano, just type ctrl+x.

The second way to cast multiple spells is to put a spell into the background while it's running. You can do that by pressing ctrl+z inside any program. Go ahead and cast `top` again. Now when it's running, press ctrl+z and you will exit out but you'll notice this on your screen:

```
[1]+  Stopped                    top
```

The result of pressing ctrl+z inside a spell

Like we just covered, this means that top is now in the background with a job number of 1 and you can bring it to the foreground with

```
joe$ fg 1
```

Bringing top to the foreground again

So there are your two ways to free up your terminal to cast more than one spell. When would you normally use this? Sometimes a spell will take a really long time to run, so you can cast it with the & at the end right away to free up your terminal. Other times you will need to switch to a new program so you can use ctrl+z. But normally, you can just open up a new terminal -- there is no limit to them anymore (there used to be back in the old days)!

Killing Processes

Sometimes a spell will to awry and you will need to kill it. I know it sounds mean, but sometimes it just has to be done!

To kill a disgruntled spell, you cast `kill` and use the Process ID as an argument. For example, if the spell with Process id 699 went bad, you could kill it like this:

```
joe$ kill 699
```

Killing a spell

There are cases when a spell has gone so crazy that even using a normal `kill` won't put it out of it's misery. In that case, we have to use a greater killing force. This is known as `kill -9`. You cast it like this:

```
joe$ kill -9 699
```

Using kill -9 on a spell

And if *that* doesn't do anything, then you're in some trouble and you should contact an archmage!

There are several other numbers you can use as options with kill -- not just 9. To learn about them, read the `kill` man page.

Exercises for Chapter 8

1) Spells that end right away

What happens if you were to cast this spell and why?

```
joe$ ls /bin &
```

Spells Learned in Chapter 8

Spell	What it stands for	What it does
`ps`	Process Status	Shows information about running processes
`jobs`	Jobs	Shows current jobs in the background
`fg`	Foreground	Brings a job to the foreground
`kill`	Kill	Kills a process

Epilogue

The Young Mage has completed the basic training; A farewell.

Congratulations, Young Mage, you've reached the end of the basic training! You've definitely learned a lot about the Unix Spellcraft since your training began and you should be very proud for finishing! As a gift for completing, the Archmages of the Tower have decided to raise your rank to a standard Mage! Congratulations!

But your journey is far from over! There are several topics you can learn from here -- such as creating your own spells, managing your own Tower, and communicating with other Towers.

If you choose to learn about creating your own spells, you'll be known as a Spellcrafter. Spellcrafters use a variety of languages to create their spells such as Python, Perl, C, and C++. The Unix Spellcraft provides an amazing environment to create your own spells and all Mages are encouraged to learn, at least, the basics of spellcrafting.

Managing your own Tower is an art known as System Administration. There are several aspects to System Administration such as building and maintaining your own Towers, keeping your Tower secure, and making sure the data stored in the Tower is properly backed up.

Finally, communicating with other Towers is an art known as Networking. You'll learn all about sockets and ports and a numerous amount of daemons. During your training of the Networking art, you'll run into a few different factions of Mages such as the White Mages and Black Mages -- and you'll eventually have to join one yourself!

As you can see, you have several different choices to pick from! Also, don't think that you're confined to just picking one choice -- if you can't make up your mind, pick all three!

Good luck with your learning, and above all else, don't forget to have fun!

Appendix A

A complete list of all spells learned.

Spell	What it stands for	What it does
`cal`	Calendar	Prints a nice calendar out for you
`cat [file1] ([file2])..`	Concatenate	Sends the output of one or more files to the screen
`cd [path]`	Change Directory	Takes you to the specified path
`chmod [shield] [file]`	Change Mode	Changes the Shield on a file
`chown`	Change Owner	Changes the owner of a file
`cp [src] [copy]`	Copy	Creates a copy of a file or directory
`df`	Disk Filesystems	Shows the partitions and filesystems of the computer
`echo`	Echo	Prints something to the Terminal
`fg`	Foreground	Brings a job to the foreground
`file [file]`	File	Prints the type of file
`grep [text] [file]`	Grep	Grabs text you asked for
`jobs`	Jobs	Shows current jobs in the background
`kill`	Kill	Kills a process
`less [file]`	Less	Allows you to read a file with many pages easily
`ln [file1] [file2]`	Link	Links two files together
`ls`	List	Prints the contents of the directory
`mkdir [name]`	Make Directory	Makes a directory with the specified name
`more [file]`	More	Allows you to read a file with many pages easily
`mv [src] [dst]`	Move	Moves a file or directory to a new location
`nano`	Nano	One of several text-editors
`passwd`	Password	Changes your password
`pico`	Pico	Another text-editor
`ps`	Process Status	Shows information about running processes
`pwd`	Present Working Directory	Prints the absolute path of the directory you are in
`rm [file]`	Remove	Removes a file
`rmdir`	Remove Directory	Removes an empty directory
`su`	Switch User	Switches you to another account
`sudo`	Superuser Do	Allows you to run root commands
`touch [name]`	touch	Creates a blank file
`wc`	Word Count	Counts words for you
`who`	Who	Shows all mages currently in the Tower
`whoami`	Who Am I?	Prints your username

Appendix B

A quick guide to building your own Tower

If you would like to build your own Tower of Nix, you've got plenty of options to choose from. Unfortunately, for beginners, this isn't exactly a good thing. I'll try to explain as easily as I can the different paths you can take.

Mac OS X
When Apple released OS X, it instantly became the most advanced and user-friendly Unix-based operating system. If you're a Mac user, chances are you're already using OS X (as opposed to OS 9), so you already have a Unix-based computer to use. You can find the Unix terminal under the Utilities folder of Applications. Just open that up and you're all set!

BSD
If you would like to try out the BSD family of operating systems, you have a few choices. The two easiest choices would be PC-BSD or a Live FreeBSD CD[8]. If you would like to rough it out and start from scratch, you're more than welcome to try FreeBSD, OpenBSD, or NetBSD from a clean install. A word of warning, though -- they're not very friendly to absolute beginners. I would recommend finding a good walk-through or book.

Linux
There are several Linux distributions to choose from and many of them are beginner-friendly. My choice would be to go with Ubuntu -- a newer distribution of Linux that's incredibly easy to use. You have two choices with Ubuntu: a full install and a Live CD. I would recommend starting with a Live CD.

If you would rather not go with Ubuntu, you are more than welcome to try another distribution such as Fedora, Mandriva, Debian, SuSE, Slackware, or Gentoo. Please be aware that some of these are more difficult to install than others and you should consult the installation instructions.

Resources
OS X: http://apple.com/macosx
PC-BSD: http://pcbsd.org
FreeBSD Live CD: http://livecd.sourceforge.net
FreeBSD: http://freebsd.org
OpenBSD: http://openbsd.org
NetBSD: http://netbsd.org
Ubuntu: http://ubuntulinux.org
Fedora: http://fedora.redhat.com
Mandriva: http://www.mandriva.com/
Debian: http://debian.org
SuSE: http://www.novell.com/linux/suse/
Slackware: http://slackware.com
Gentoo: http://gentoo.org

[8] A Live CD is a CD you can boot your computer up with and will provide you with a full BSD or Linux installation. When you are done, simply reboot your computer!