Intermediate Unix Training

pwalker@ncsa.uiuc.edu

National Center for Supercomputing Applications
152 Computing Applications Building
605 E. Springfield
Champaign, IL 61820-5518

Contents

Introduction

These documents contain the course materials for "Intermediate Unix", by Paul Walker. The course was first given Oct 19, 1995, and then again April 3, 1996. If you have any questions or comments about these documents, please contact Paul at pwalker@ncsa.uiuc.edu.

One comment is in order. ***Intermediate Unix*** is a very large and varied topic. There are many different classes that could have been taught with this title, and those classes could have lasted between hours and days. I've chosen the topics here because these are tools and commands I use every single day. I have also skipped many items since I have a short (2.5 hour) time limit. An omission of a topic does not mean it is useless; rather it inidicates it is too complex, obscure, or infrequently used (in my opinion) to warrant inclusion in a course of this scope.

These documents are available at:

http://www.ncsa.uiuc.edu/General/Training/InterUnix/

Chapter 1

/bin/csh concepts and tricks

Most unix users use `csh` as their primary working environment. The csh gives you the prompt and reads your commands. However, `/bin/csh` has many powerful features including history, job control, aliasing, foreach statements, and many other things. These are accessed either via the command line, your `~/.cshrc` configuration file, or csh scripts (which we will cover later in this course (p. 35)).

Some users use shells other than `csh` for their day-to-day use, the most common being `tcsh`. We will not cover these other shells.

Using shell history 1: Recalling old commands

The csh has a history manager which remembers old commands. The number of old commands it remembers is set with the `history` environment variable which you probably want to set to a large number with a command like

```
set history = 200
```

in your `~/.cshrc` file. Each command is then given a number. You can see your previous commands and their numbers with the command `history #` where `#` is the number of commands you want to see. For example,

```
loki(pwalker).30 % history 5
    26  idl
    27  setenv DISPLAY hopper:0
    28  idl
    29  more ~/.exrc
    30  history 5
```

You can recall previous commands with three basic mechanisms. The first is to use `!#` where `#` is the number of the command of interest, eg

```
loki(pwalker).31 % !27
setenv DISPLAY hopper:0
```

Note the command is echoed out to the console after being recalled. This behaviour is common.

The second method is to use `!pattern` where pattern is a pattern to be matched against a previous command. eg,

```
loki(pwalker).32 % !set
setenv DISPLAY hopper:0
```

or, here

```
loki(pwalker).33 % !s
setenv DISPLAY hopper:0
```

The third method is to use `!-#` where # is the number of commands to go back, eg

```
loki(pwalker).41 % history 3
    39  ls
    40  setenv DISPLAY hopper:0
    41  history 3
loki(pwalker).42 % !-3
ls
```

Finally, many people use `!!` to repeat their last command. I mention this here, but it really belongs in the next section.

Using shell history 2: Bits of old commands

As well as recalling old commands, you can recall segments of your previous command very easily with the csh. The syntax for this is `!x` where `x` is a special character. The following are the useful values of `x`, and how they would operate on the example string

`a.out arg1 arg2 arg3`

`!!`	Entire previous line	`a.out arg1 arg2 arg3`
`!*`	All the arguments	`arg1 arg2 arg3`
`!$`	Last argument	`arg3`
`!:#`	The #'th argument (from 0)	
`!:2`	The second argument	`arg2`

An example of when these can be useful is, for example (with the output supressed)

```
hopper(pwalker).48 % ls /afs/ncsa/common/doc/ftp
     /afs/ncsa/common/doc/web
  ...
hopper(pwalker).49 % ls -l !*
ls -l /afs/ncsa/common/doc/ftp /afs/ncsa/common/doc/web
```

```
   ...
hopper(pwalker).50 % ls !$/VR/
ls /afs/ncsa/common/doc/web/VR/
```

where I have added a line break in the first line to improve readability. Note the contraction of `!$/subdirectory` which is always useful.

Using shell history 3: Changing old commands

Often, you will want to change a previous command a little. This is also fairly straightforward in the csh, although many people prefer to cut and paste with the mouse. Well, that's because they didn't have to work on a vt100 when they were undergrads (or maybe because they did!). So, here is how you modify your old commands.

To modify your previous command in one place use the `^old^new` constuction, which replaces the old with the new. For example,

```
hopper(pwalker).52 % mire imagemap.txt
mire - Command not found
hopper(pwalker).53 % ^ir^or
more imagemap.txt
```

To modify an old command, use the (somewhat non-intuitive) construct

```
[command reference]:[g]s:old:new
```

`[command reference]` is any reference to a previous command, eg `!!` or `!35`

`[g]s` means search (`s`) or global search (`gs`) and replace. Note the *global* is a little misleading. *global* means apply the search once per word, which is a little different than most peoples perception of *global*

`old:new` means replace old with new.

So, for instance:

```
farside(pwalker).212 % foo input

farside(pwalker).213 % bar input

farside(pwalker).214 % !212:s:in:out
foo output

farside(pwalker).215 % !214:gs:o:a
fao auput
```

Notice the tricky behaviour of the global search on line 215.

Background a job

Putting a job in the background means that it will run but return the prompt for you. This is possible due to Unix's multi-tasking abilities.

There are two methods to background a job. The first is to place it in the background when you start the job. This is accomplished by placing an ampersand (&) at the end of the invocation. For instance,

```
a.out arg1 arg2 &
```

Then a.out will run but you will have your prompt back. Note the output from a.out will still come to your console. We will discuss how to avoid this in the pipes and redirection (p. 13) section of the course.

The other method is to suspend the job and then background it. To suspend (stop) a job, use `^Z` where I'm using `^` to mean control. This will stop the job. Then place the job in the background with the command `bg`.

Managing jobs in the background

You can easily have several jobs backgrounded, and you often want to bring one of them to the foreground, kill one, or otherwise manipulate them.

The `jobs` command will tell you which jobs you have running. For instance,

```
hopper(pwalker).77 % jobs
[1]  + Running              xdvi EH_V3.dvi
[2]  - Running              xemacs src/ReadData.c src/SfcEvolve.c
```

This display has several bits of information. The job number is in the []. The + indicated which job is the current default for `fg` and `bg` commands, the job status is shown, and the job name.

You can bring a job to the foreground with `fg %#` where `#` is the number indicated in the output of jobs, eg:

```
hopper(pwalker).78 % fg %2
xemacs src/ReadData.c src/SfcEvolve.c
```

You can then stop this job with `^Z` and jobs will show it as stopped.

```
hopper(pwalker).80 % fg %2
xemacs src/ReadData.c src/SfcEvolve.c
<---- I pressed ^Z here, but it didn't show up!
Stopped
hopper(pwalker).81 % jobs
[1]  - Running              xdvi EH_V3.dvi
[2]  + Stopped              xemacs src/ReadData.c src/SfcEvolve.c
```

Then to background this, use `bg %2` or simply `bg` since the job is currently selected (as inidcated by the +).

Finally, you can easily kill a job:

```
hopper(pwalker).83 % kill %1
[1]    Terminated           xdvi EH_V3.dvi
```

using the same syntax as the other commands.

Some advanced alias tricks

Most users know about the simple aliases easily available in the csh. For instance,

```
    alias rm rm -i
```

will alias the command `rm` to `rm -i`, eg, to prompt you before it removes any files.

However, you can easily write advanced aliases by making use of command line history. For instance,

```
    alias   fc      "fgrep \!* src/*.[ch] "
```

Allows me to type `fc Metric` rather than `fc Metric src/*.[ch]`. Note that we have to *protect* the `!` character. This causes the alias to contain `!*` explicity rather than all the arguments of the previous command, then when the alias is evaluated, `!*` will contain the arguments at evaluate time.

This alias shows how you can use command line history discussed earlier in your aliases. Any of the meta-character expressions explained earlier can be used in an alias to extract parts or all of the command line.

For instance, lets say a lot of the time, you edit a file, then copy it to a directory, eg do something like

```
vi myfile
cp myfile ~/mydir
```

You could create an alias called, for instance, `vic` with the command

```
alias vic  "vi \!:1; cp \!:1 \!:2"
```

which will vi the first argument, then copy the first argument into the second argument. So the above two commands could be executed as `vic myfile ~/mydir`.

Globbing

Globbing is the process by which the csh handles wildcards in file names. You probably know some basic globbing techniques, such as `ls *.html` to list all files ending with the `.html` extension, but globbing is very powerful and allows you great flexibility.

There are five basic constructs in a glob.

`*`	Match any number of anything
`?`	Match one of anything
`[...]`	Match one character in the brackets
`~`	Your home directory
`~user`	`user`'s home directory

This is best shown with an series of examples. So here are some examples!

Expression	Matches	Doesn't Match
A*.html	Albert.html	Robert.html Alhtml
A*rt.?	Albert.c	Albert.html A_Cat.html
A?[1-5].dat	Ax3.dat Ax1.dat	Albert1.dat Ax6.dat
[A-C]*[1-5][2468]?.html	Albert163.html Charlie58x.html	Charlie52.html Charlie621.html

Using foreach to access multiple files

`foreach` is a very useful construction from the command line. It allows you to loop over multiple files and do the same thing to each file. The basic syntax is:

```
foreach VARIABLE ( FILES )
? command on $VARIABLE
? command on $VARIABLE
? end
```

Where `VARIABLE` is anything you want and `FILES` is any acceptible file or glob.

For instance, to print the name of each html file in a directory, and follow that with the number of lines, you could use

```
foreach C (*.html)
? echo $C
? wc -l $C
? end
```

BackTicks

Backticks (‘) return the output of a command in a form so that you can use it in a command you construct. Take, for instance, the command **uname -n** which returns the name of the current machine.

```
hopper(pwalker).62 % uname -n
hopper
hopper(pwalker).63 % pwd
/tmp
hopper(pwalker).64 % mkdir /tmp/‘uname -n‘
hopper(pwalker).65 % cd /tmp/‘uname -n‘
hopper(pwalker).66 % pwd
/tmp/hopper
hopper(pwalker).67 %
```

The utility of this function is limitless, but I won't mention it beyond this simple example.

Grouping commands with ()

In the **csh** you can group sets of commands with (). The commands in these parens have their own shell, and can do shell things, such as set environment variables, redirect, and pipe, independently of the main shell.

The clearest example of this behaviour is with **setenv** which sets an environment variable for your current shell. Note that the setenv outside the parens affects the entire session, but inside, affects only the commands inside.

```
hopper(pwalker).17 % setenv SUMPIN Hi
hopper(pwalker).18 % echo $SUMPIN
Hi
hopper(pwalker).19 % (setenv SUMPIN Lo; echo $SUMPIN)
Lo
hopper(pwalker).20 % echo $SUMPIN
Hi
```

Chapter 2

Pipes and Redirection

What is a stream

Unix has a well defined concept of a ***data stream***. There are three well defined data streams in unix for all process, **stdin**, **stdout** and **stderr**.

The streams are merely flows of data. **stdin** is the standard input for a process and is usually generated via the keyboard. **stdout** is the standard output for a process and is usually sent to the screen. **stderr** is the standard error channel for a process, and is also usually sent to the screen. However, **stderr** only contains error messages.

In many ways, you can consider every unix process a box which reads **stdin** or command line arguments and based on that, creates something on **stdout** or in a file.

What is a pipe

A Pipe is used to connect the **stdout** of one process to the **stdin** of another process. The jargon for this action is ***Piping into***, eg ***pipe the output of command1 into command2***

What is a redirect

A redirect is a method to turn a file into a stream or a stream into a file. Redirects are used in two places, either at the beginning of a pipe to pipe a file into the **stdin** of a process or at the end of a pipe to pipe the **stdout** of the process into a file.

How do I use a pipe

The symbol for a pipe is | (the vertical bar, often shift-backslash). The essential syntax for using pipes is

```
command1 | command2
```

which will connect the **stdout** of **command1** to the **stdin** of **command2**.

Note you can pipe the **stdout** and **stderr** of a command into another command using |&. For example,

```
command1 |& command2
```

This is often useful when commands make screenfuls of **`stderr`**, such as when compiling codes.

How do I use a redirect (/bin/csh version)

Redirects are different in different shells. Here, I will use the **`/bin/csh`** version.

The 4 basic redirects in the csh are:

<	Turn file into **`stdin`**
>	Turn **`stdout`** into a file
>>	Append **`stdout`** onto a file
>&	Turn **`stdout`** and **`stderr`** into a file

For example:

`a.out < input`	The stdin of a.out is read from the file input
`a.out > output`	The stdout of a.out is placed in the file output
`a.out < input > output`	The **`stdin`** of a.out is file input and the output goes into **`output`**
`a.out >> output`	Append the output of a.out to **`output`**
`a.out < input >& output`	**`stdin`** from input and **`stdout`** and **`stderr`** to output.
`(a.out > out_std) >& out_ste`	Place **`stdout`** in **`out_std`** and **`stderr`** in **`out_ste`**

Chapter 3

Pipe Building Blocks

cat, -, and the bit bucket (/dev/null)

`cat` is sort of the pipe equivalent of a `no-op`. That is, `cat` merely reflects `stdin` in `stdout`. `cat` will also echo a file onto `stdout` if that file is given as an argument.

Many unix commands take `-` as an argument where normally they would take a file. This means the command will look on `stdin` for its input rather than trying to read a file. Some of the commonly used commands which do this include `xv` and `gs`.

`/dev/null` is a place you can redirect things you don't care about. You cant get this information back, but it also won't clutter your screen or a file.

Examples

```
cat nice.gif | xv -
```

is the equivalent of

```
xv nice.gif
```

```
cat  file2
```

is the equivalent of

```
cp file1 file2
```

Do you see why?

```
cat file1 > /dev/null
```

is a wast of time! Do you see why?

more or less

`more` and `less` are roughly equivalent programs which show you a screenful of output at a time. `less` aficionados claim that `less` is infinitely better than `more` since it provides searching and navigation functionality not available in `more` (for instance, the ablilty to go back through `stdin`). You can use either (if `less` is installed on your system) and should read the man page.

`less` and `more` both take files as arguments or read `stdin` for the information to page. This lets you observe a pipe.

Examples If you want to look at a file, use any one of

```
more a.file
more < a.file
cat a.file | more
```

Imagine `a.out` spews tonnes of information to your screen when given certain parameters. You could observe these with

```
a.out < input | more
```

head and tail

`head [-count] [file]` and `tail [-count] [file]` print the first (`head`) or last (`tail`) count lines of the file `[file]` or stdin. The default value of `count` is 10.

`tail` also has the useful option, `tail -f`, which will monitor a file printing new additions to it as the are added. This is very useful for watching code output when it has been redirected, or following logs and the like.

Finally, `tail -r` reverses stdin and prints it out to stdout. This is often useful. Examples This shows you the last 200 lines of `/etc/passwd` one screen at a time.

```
cat /etc/passwd | tail -200 | more
tail -200 /etc/passwd | more
```

This runs a command and then watches the output accumulate in a file.

```
a.out >\& a.file \&
tail -f a.file
```

wc

`wc` counts all the words, lines, and characters in a file or stdin. If you only want words, lines, or characters, use the `-wlc` flag respectively.

`wc` is useful for tallying how much code you've written (`wc -l *.[ch]` for instance), or things like that. It also works in pipes, reading stdin. This will be useful once we understand grep. Examples

```
wc -l HorizonMetric.c
```

tells me the number of lines in Horizon Metric.c

grep: Grab a Regular Expression

`grep` stands for ***Grab Regular ExPression.*** Yeah, I know it is a strange name, but it is far less strange than the motive behind the name **`awk`**. **`grep`** has the basic syntax **`grep [-i] [-v] Expression [file]`**. Actually, there is quite a bit more to it (see **`man grep`**) but these are all the options we are going to discuss here.

`grep` matches **`Expression`** against either the **`file`** or **`stdin`**. The default behaviour is that **`grep`** will echo any lines which match **`Expression`** to **`stdout`**. The **`-v`** flag forces grep to echo lines which **don't** match. The **`-i`** flag forces the matching to be done in a case independent fashion.

`grep` understands a limited set of regular expressions. This is usually enough for most work. The consituents of a grep regular expression most frequently needed are

"Normal" characters	Match the character
`^`	Beginning of line
`$`	End of line
`.`	Any single character
`.*`	Any number of anything

Be careful using `.*` since the shell will interpret it. You will either need to *protect* it (eg, .
) or more simply, put your expression in single quotes.

`grep` has two close relatives, **`fgrep`** and **`egrep`**. **`fgrep`** matches only text patterns and is faster than **`grep`**. **`egrep`** has a more extensive regexp recognition. Consult the man pages of these two commands. Examples of grep

These examples break down into 2 sets. The first set works on this file, which I've called **`ge`**:

```
hopper(pwalker).26 % more ge
pwalker
walker
walkerp
```

```
orangejuice
was that a warp
hopper(pwalker).27 %
```

Now let us look at grep and some of its output acting on this file.

```
hopper(pwalker).34 % grep walker ge
pwalker
walker
walkerp
hopper(pwalker).35 % grep -v !*
grep -v walker ge
orangejuice
was that a warp
hopper(pwalker).36 % grep ^walker ge
walker
walkerp
hopper(pwalker).37 % grep walker$ ge
pwalker
walker
hopper(pwalker).38 % grep 'wa.*p' ge
walkerp
was that a warp
```

The next set involve using **grep** in pipes. Here are some actual examples.

```
hopper(pwalker).49 % history 50 | grep pine
     9  pine -i
    11  pine -i
    49  history 50 | grep pine
hopper(pwalker).50 % grep -i dave /afs/ncsa/common/etc/passwd
   | wc -l
       39
hopper(pwalker).51 % grep -i paul /afs/ncsa/common/etc/passwd
   | grep -v walker | wc -l
      114
```

where I have added line breaks to improve readability; You should have the entire pipe on one line.

As you can see, using **grep** and **wc** together work very well, and gives a good way to figure out things like percentages of accesses to a server from Macintosh (`grep Mac agent_log | wc -l`) as a fraction of total access (`wc -l agent_log`). Grep is so useful, I could go on forever, but I wont. I think there is enough information here to get you started!

[n]awk: Manipulate Columns

awk is a complete programming language which is nowhere near as useful as perl. Writing large **awk** codes in a world with perl is not very useful unless you already know **awk** and don't know **perl**. **nawk** is new awk, and you should use **nawk** if it is available. On most systems, **awk** = **nawk**

However, **nawk** is very useful as a quick and dirty manipulator of columnar textual data. columnar data is data arranged in columns with a separator such as whitespace (the default) or some other delimiter, such as the : in /etc/passwd and /etc/group. **awk** allows you to directly address, and in the case of numerical data, maniuplate, the columns.

The basic syntax for this use of **awk** is

```
nawk -Fc '{print cols}' [file]
```

where `-Fc` is optional and specifies the record separator. Without it, the default separator is whitespace. `cols` is replaced with the columns of interest addressed as `$1, $2` etc... for column 1, 2 etc... `file` is an optional file. Without it, `stdin` is parsed.

Example 1. Numerical Data Imagine you have a text file with data in polar coordinates as

```
r theta
r theta
r theta
```

and you want to convert this to cartesian coordinates (x = r cos(theta), y = r sin(theta)). Rather than writing a dumb little fortran or c program to do this, you can use **awk** as follows.

```
nawk '{print $1*cos($2), $1*sin($2)}' polar.dat > cartesian.dat
```

Example 2. System files Many system files in unix are : delimited files. For instance, /etc/passwd has user information separated into columns with :. (`more /etc/passwd` or `man 4 passwd` on your system for more info). You can extract subsets of this information using awk. For instance, to extract users (column 1), user numbers (column 3) and home directories (column 6) from /etc/passwd, you could use

```
nawk -F: '{print $1, $3, $6}' /etc/passwd
```

Other useful building blocks

There are several other useful items which I don't have time to go into extensively, but will mention here.

sort

Sort sorts. It seems quite cryptic, though, and the only option I can every remeber properly is `sort -n` which sorts numerically. This is useful to find out your disk usage in an area, biggest directory first, with

```
  du -k | sort -n | tail -r | more
```

`man sort` if you actually want to master this potentially useful utility.

tee

`tee` creates a sort of "T" junction in your pipe. It takes a file as an argument. The action of `tee file` is to take `stdin` and pipe it to both `stdout` and `file`. This allows you to see intermediary results in pipes. For example:

```
grep walker /etc/passwd | tee all_the_walkers | grep paul
    > all_the_paul_walkers
```

will create two files for each stage of the pipe.

sed

`sed` is a stream editor. That is, it allows you to make ed/ex/vi like operations on a stream. It is useful, and has a good man page. Given inifinite time, I would say learn `sed`. Given finite time, learn `perl` instead. (There is a separate lrc course on `perl`).

echo

Most people forget that `echo` can start a pipe. For instance:

```
hopper(pwalker).54 % echo Hi | wc -l
         1
```

This is occasionaly useful.

An example: Superpipe!

For the superpipe, we are going to use /etc/passwd. Lets take a look at a line from this file from an SGI.

```
pwalker:x:15299:1023:Paul Walker,,217-244-1144:
/usr/people/ncsa/pwalker:/bin/csh
```

where the line break is added to avoid wrapping. The entry is actually a single line.

Notice the 5th field is of the form `Full Name, Office, Phone` and my office is not specified here. Now, lets say we want to print out the office and user number of every person whose uses csh and whose name contains the string *wal*, a commen sysadmin task indeed. Oh, also we want to save all the lines from the original file in walpwd.

So how do we do this? Well, clearly we have to start off with a couple of greps, eg,

```
grep wal /etc/passwd | grep "/bin/csh" |
```

and then a tee.

```
 | tee walpwd |
```

OK, now we have to use awk twice. The first time we will turn the string into the form `userno,name,office,phone`. This is done with

```
  | awk -F: '{print $3 "," $5}' |
```

And finally we want to split this on , and print out the first, second, and fourth field.

```
  | awk -F, '{print $1, $2, $4}' |
```

and for good measure, send it into more. So our final superpipe is:

```
grep wal /etc/passwd | grep "/bin/csh" | tee walpwd |
awk -F: '{print $3 "," $5}' |
awk -F, '{print $1, $2, $4}' | more
```

I've put in the line wraps for ease of viewing, but it could not be there on your script.

OK, and lets see what this does on one of our machines:

```
farside(pwalker).238 % grep wal /etc/passwd | grep "/bin/csh" |
  tee walpwd | awk -F: '{print $3 "," $5}' |
awk -F, '{print $1, $2, $4}' | more
15299 Paul Walker 217-244-1144

farside(pwalker).239 % more walpwd
pwalker:x:15299:1023:Paul Walker,,217-244-1144:
/usr/people/ncsa/pwalker:/bin/csh
```

where once again, I've added line breaks for ease of reading.

And this is, inded, a super-pipe!

Chapter 4

vi tricks and tips

Emacs

Many people, mostly stubborn emacs users, claim that the only thing you need to know about vi is how to get out of it! This belief is due to the fact that emacs is a widely available extensible wonderfully fabulous editor. Correctly configured, it can do color sensitive highlighting of your text, indent and align text cleverly, have different behaviours based on the type of file you are editing, and many other features, none of which vi has.

If you plan to do a lot of editing in a Unix environment, let me encourage you to use emacs for all your serious needs, since it is undeniably an infintely superior editor.

However, you have to know vi. You need to know vi because it is universal, quick, and often, you are kicked into it by some program. Even the most convinced emacs users use vi for small tasks, and so, you need to know some simple tips and tricks, the purpose of this chapter.

Marks and yanking/moving blocks of text

`vi` has the concept of a **mark**. A mark is a line in your document to which you assign a special tag which is a letter. Once you have set a mark, you can use that mark as a place keeper. `vi` can also use numbers as place keepers. A number used as a place keeper specifies a line number. Finally, `vi` has two special marks `$` which is used for the end of document and `.` which is used for the current position.

A mark is set using the `vi` command `mx` where `x` is your mark tag.

Two of the most useful things you can do with marks is yank or delete text between two marks. yanked or deleted text can then be restored to the current cursor location by pulling it with `p`.

To yank text, use the command `:'a,'by` where a and b are marks. Note, `a` and `b` can be numbers or a special mark, but then do not need the quote. This text will remain in your buffer but also be in your kill ring so it can be pulled and thus copied. To delete a block of text, use `:'a,'bd`, which will delete your area and put it in your kill ring.

Some examples are best, and it is probably easiest if you try along with these (which is what we will do in class). Examples To delete everything between the current position and the end of the document do

```
:.,$d
```

or between the beginning of the document and current position

```
:1,.d
```

A quick way to delete a chunk of text is to position the cursor at the beginning of the text and make a mark with

```
md
```

Then reposition to the end of the block and use

```
d'd
```

which will delete everything from the current position to the mark `d` which we made with the `md` command. This text could then be recovered by moving to another location and issuing `p`

Finally to mark and delete a block of text, set 2 marks, then use

```
:'a,'bd
```

Searching and Replacing

Most of you know that `/string` will search for a string in your current `vi` buffer `n` will repeat the last search.

However, most `vi` users have a difficult time searching and replacing. Luckily this is quite easy. Simply use

```
:'a,'bs/old/new/[g]
```

where `a` and `b` are marks or line numbers (without the '). The optional g means to replace everywhere. Without it, only the first occurance on each line will be replaced.

Examples To replace green with blue in your entire document, use

```
:1,$s/green/blue/g
```

To replace blue with green in lines 10 through 24 use

```
:10,24:s/blue/green/g
```

Line numbers

A very quick but very useful series of tips w.r.t. line numbers. Often you want to go to a specific line number with `vi`. You can do this simply with

```
:number
```

eg `:13`.

You can display the line numbers of your current file beside the line by using `:set number` and turn it off with `:set nonumber`.

These are very useful when editing and debugging programs.

Using your .exrc (NOT your .virc)

There are a few useful environement variables and defaults which you can set to affect vi. Defaults are set in your `~/.exrc` file, and environment variables are set in the standard fashion.

The one environment variable of interest is `ESCDELAY` which you may want to set to a value of about 1500. This variable determines the amount of time between when ESC is pressed and a new command is issued. Since arrow keys form esc sequences, if the sequences are genreated too slowly, an arrow key can, instead of moving your cursor, insert some garbage into your file (which often looks like `^[A`). Try some values of ESCDELAY to fix this.

You can set any vi settable thing in your `~/.exrc` file. For instance, if you always want numbers on, you should put in your `~/.exrc`

```
set number
```

You should consult `man vi` for more info, but a very useful thing to set is

```
set ai
```

which causes an auto-indent feature. This means your cursor lines up at the tab stop of your previous line when you create a new line. This is very useful for program editing with `vi`. Of course, if you are doing serious program editing, you should probably use `emacs`...

Chapter 5

File Management

Compressing with compress and gzip

The idea of compression is that in many files, especially ascii files, there is an awful lot of redundancy which can be represented with much less data than the standard representation. Thus, there is compression. Compression programs take your files and make them smaller, but unreadable except by uncompression programs. When you are archiving or need to save space, you should use compression.

There are 2 useful sets of commands for compression and uncompression of files in unix.

The standard unix **compress** utility creates compressed **.Z** files. To compress and uncompress a file, use

```
compress bigfile.ps
uncompress bigfile.ps.Z
```

The Free Software Foundation has written a commonly used compression program, gnu zip, or **gzip** which creates compressed **.gz** files. **gzip** also reads **.Z** files. **gzip** is pretty much faster, more efficient, better etc... You should use it given the choice. To compress and uncompress with gzip, use

```
gzip bigfile.ps
gunzip bigfile.ps.gz
```

Note, **gzip** has levels of compression. If you don't mind waiting a little longer to compress your files, and want better compression ratio's, use **gzip -9**.

Handling multiple files with tar

Often, it is useful to put multiple files into a single file before compressing, sending to collaborators, or backing up. The easiest way to do this is using **tar**. **tar** was originally designed to access tape drives, and it still does that ver well, but most users will use it to make archive files from multiple files.

The basic syntax for creating an archive with tar is

```
cd /usr/people/me/mydir
tar cvf mydir.tar .
```

This will create a file called `mydir.tar` which contains all the files in and below `.`. It will also show you the files being added. You can remeber the flag `cvf` by thinking of `create verbose file`.

To extract a tar archive, you want to use

```
cd /usr/people/me/mynewdir
tar xvf mydir.tar
```

which will recreate the directories stored within mydir.tar and put them underneath mynewdir. For `xvf` think `eXtract verbose file`.

If you simply want to see the contents of a tar file, you can use `tvf`, eg:

```
tar tvf mydir.tar
```

Here, think `Table_of_contents verbose file`.

These are the three basic uses of tar. However, we will see in the section on using tar in pipes that it can have other less obvious uses.

Locating files with find

`find` is an incredibly powerful, rich, complex command which locates things in the filesystem based on a variety of conditions. With that complexity comes a rather high level of difficulty, and we cannot possibly cover all the things find can do here. Rather, here I will show a minimal amount of find syntax which is often useful. Please consult `man find` to find out the rest of the story.

The basic usage of `find` is

```
find path condition action_list
```

`find` will then do `action_list` on all files at or below `path` matching `condition`.

Now, each of these things can have many settings, and you should consult `man find` to see the possiblities. However, I am only going to mention a few of the options.

`condition` is the most complex of the options. The two most useful `condition` flags are `-name "glob"` and `-mtime +/-#`. `-name "glob"` matches anything matching `glob`. `-mtime +/-#` matches any file modified earlier than/since `#` days.

`action_list` also has many options. The two I will mention are `-print` and `-exec`. `-print` simply prints all the matches to stdout with a pathname. `-exec` is used to execute a command on the matched file and has the syntax `-exec command`
; where is replaced with the name of the current match. Examples The best way to understand `find` is through some examples.

This prints all files below the current directory which end in .c or .h.

```
find . -name "*.[ch]" -print
```

This removes all core files beneath your home directory

```
find ~ -name "core" -exec rm {} \;
```

This shows all occurances of mosaic in all html files, as well as printing **all** .html files.

```
find . -name "*.html" -print -exec grep mosaic {} \;
```

Finally, these remove all files more than 7 days old and print all files modified within the last day.

```
find . -mtime +7 -exec rm {} \;
find . -mtime -1 -print
```

Be very careful with commands such as the first one on this line.

tar, gzip, and ftp with pipes

Of course, all of these commands are available in pipes, and many are very useful. I will give here the examples I use the most often.

`zcat` and `gunzip -c` uncompress a file and send the output to stdout, leaving the file compressed on disk. This is often useful if you want to simply view a file. For instance, imagine that you have compressed your postscript file, `Fig.ps` and now you want to see it without uncompressing it. You could use:

```
zcat Fig.ps.Z | gv -
gunzip -c Fig.ps.gz | gv -
```

depending on which compresser you used to compress it. Of course, you could pipe this output into anything you wanted!

`tar` can create a file onto stdout, and extract a tar file from stdin, by replacing the output or input tar file with `-`. That is, the following pairs are identical

```
tar xvf mydir.tar
tar xvf - < mydir.tar

tar cvf mydir.tar .
tar cvf - . > mydir.tar
```

One of the most useful applications of this is to move a directory as well as all its files and subdirectories to another location. This is done by creating a tar file and piping it into a subshell which runs in a different directory and untar's stdin. That is,

```
tar cf - . | (cd ~/newdir ; tar xf -)
```

Note I left out the `v` option to both tars so the code will run silently.

Finally, you can compress stdin and send it to stdout, or uncompress stdin and send it to stdout, using `gzip` and `gunzip` alone in a pipe. That is, these pairs are equivalent.

```
gunzip -c foo.gz > bar
cat foo.gz | gunzip > bar

gzip -c foo > bar.gz
cat foo | gzip > bar.gz
```

Now this combination is very useful due to a little know property of ftp. ftp allows you to specify pipes as source or receving files. For instance, you can get and view a gif image from an ftp site with

```
ftp> get file.gif "| xv -"
```

or view a file with your favorite page using

```
ftp> get README "| more"
```

This is useful, but you can also use this trick to create a tar file onto an ftp site without making that tar file on your local disk. This is invaluable for backup processes. An example of this is

```
ftp> put "| tar cvf - ." myfile.tar
```

And to retrieve and untar, use

```
ftp > get myfile.tar "| tar xvf -"
```

Or, to send and compress a tar file onto an ftp site, you can use this:

```
ftp> put "| tar cvf - . | gzip " myfile.tar.gz
```

That is, ftp makes a transfer `transfer file dest` by effectively taking the stdout of `cat file` and piping this into `dest`.

NFS File Permissions: chmod and umask

File Permissions state information about accessibility of a file. Properties which are included in file permissions include whether a file is readable, executable, or writable by certain users or groups of users.

NFS breaks the entire user community into 3 classes. You, the user, which has code `u`. Your group, `g` and the others `o` All users are `a` = `ugo`

The permissions which can be given to each of these classes are read, write, and execute. `ls -l` tells you the permissions on a given file. For example:

```
-rwxr-xr-x    1 pwalker  user       13308 Oct 13 10:48 a.out
```

The first string here shows the permissions given to the classes. The first character indicates the file type. - means a normal file. The next three are the permissions for the user. Here the user has read, write, and execute permissions. The group permissions are the next three, and hear are read and execute. The world permissions are the last set, here read and execute again. This means that I can modify the file but anyone can copy or execute it.

You can change the permissions on a file using `chmod`. `chmod` has syntax `chmode permission files`. `permissions` can be stated numerically (we will skip that; See `man chmod` for more) or more intuitively, with strings.

The permission strings have the syntax `class +/- permission`, eg `u+rwx` to give a user read, write, and execute, or `go-rwx` to remove read, write, and execute from group and other. An example is:

```
hopper(pwalker).20 % chmod go-rx a.out
hopper(pwalker).21 % ls -l a.out
-rwx------    1 pwalker  user       13308 Oct 13 10:48 a.out
```

AFS File Permissions I: acl's

In AFS, the only relevant NFS file permission is whether or not a file is executable. The remaining permissions are ***directory based*** permissions based on ***AFS Users*** not user, group, and world like `NFS`.

The AFS Permissions for a directory can be examined by using the `fs la` command. (la means list attributes). This will show you the `acl` (Access Control Lists) for a directory. For example

```
hopper(pwalker).51 % fs la ./2DWave
Access list for ./2DWave is
Normal rights:
  pwalker:twodwave rl
  projects.genrel.admin rla
  system:administrators rlidwka
  system:anyuser l
  pwalker rlidwka
```

This gives the lists of users and AFS groups which have various permissions. Each character of the string indicates a permission, with the crucial ones being

r	Read
l	Lookup (eg, stat)
idwk	Various parts needed to write and create
a	Administer (eg, change acl's)

The first column indicates to whom those permissions belong. These holders may be users (eg, pwalker) or groups (eg, pwalker:twodwave, projects.genrel.admin) or special users (system:administrator, system:anyuser and system:authuser for the administrators, any user regardless of tokenization, and any user with a token).

You can set the AFS permissions in a directory using **`fs sa`**. For instance, if I want to give user johnsonb write permission to my home directory, I could go:

```
hopper(pwalker).55 % cd /afs/ncsa/user/pwalker
hopper(pwalker).56 % fs sa . johnsonb rlidwk
```

This would allow Ben to write to my home directory. (I undid this example immediately after issuing this command).

AFS provides shorthand for the most commonly used acl sets. **`read`** = **`rl`**. **`write`** = **`rlidwk`**. **`all`** = **`rlidwka`**. So above, I could have said "fs sa . johnsonb write".

ACL's allows user great deals of flexibility in setting permissions on directories. In order to maximize efficiency, though, we want to use AFS groups, to which we turn our attention next.

AFS File Permissions II: Groups

Setting acls can be a great way to allow groups of users accesses to various parts of your afs file system. In order to maximize this ease of use, though, you can create ***AFS Groups***. Any user can create groups, and then assign directory acl's to those groups rather than individuals.

The **`pts membership group`** command tells who is in a group. **`pts creategroup group`** creates a group, **`pts adduser group`** adds a user, and **`pts removeuser group`** removes a user.

An example is the easiest way to see how this works. Imagine I have a directory called **`BigPerlProject`** and I want to set acls to that directory. I could use the following:

```
farside(pwalker).19 % pts creategroup pwalker:big_p_proj
group pwalker:big_p_proj has id -750

farside(pwalker).20 % pts adduser johnsonb !$
```

```
pts adduser johnsonb pwalker:big_p_proj

farside(pwalker).21 % pts adduser royh !$
pts adduser royh pwalker:big_p_proj

farside(pwalker).22 % pts adduser sbrandt !$
pts adduser sbrandt pwalker:big_p_proj

farside(pwalker).23 % fs sa . pwalker:big_p_proj write

farside(pwalker).24 % fs la .
Access list for . is
Normal rights:
  pwalker:big_p_proj rlidwk
  system:administrators rlidwka
  system:anyuser l
  pwalker rlidwka

farside(pwalker).25 %
```

so now johnsonb, royh, and sbrandt can write to this directory. As the project personnel change or expand, I only have to modify this group, not each directory to which this group has permissions. As you can see, this is really a powerful feature for collaboration.

Chapter 6

Basic Scripting with the csh

Scripting with the csh is something you probably don't want to do too much of. As a scripting language, the csh lacks many powerful features available in real languages like perl and more powerful shells such as sh and bash.

However, short csh script can ease repetitive tasks and allow you to use your already existant csh tricks in simple scripts. Also, configuration files such as your `~/.cshrc` use csh constructs, so understanding scripting can help you modify them. In this section, we will cover scripting basics, command line parsing, and simple control flow in the csh. By the end of this chapter, you should be able to read and write basic csh scripts.

In the rest of this chapter, I will use ***shell script*** and ***csh script*** interchangeably. Most people think of ***shell script*** as `/bin/sh`, so this is sloppy language on my behalf, but you've been warned!

Basic Ideas

The basic idea of scripting with the shell is that you have a set of commands, and perhaps some control statements, which make the shell do a series of commands. Basically there is no difference between a ***script*** and a ***program***.

There are a few basic syntactical elements you need to know.

First, every shell script begins with the line

```
#!/bin/csh (optional arguments)
```

where *(optional arguments)* is an argument to the shell. The only argument we will mention here is `-f` which stops the shell script from sourcing your `~/.cshrc`.

This line tells a unix machine that everything in this file should be processed with the command `/bin/csh`. You can use this exact same syntax to run other programs over a script, which is how `sh` and `perl` scripts work.

Next, anything after a # is a comment. For instance

```
# This line does nothing
/bin/ls   # This line runs ls, but this is a comment
```

Finally, anything else is run by the `csh`

The Simplest Script of All!

OK, so lets start with the simplest shell script there possibly is. This script doesn't offer much more than an alias, but it shows a full example of creating a script. It is also very contrived!

Lets say that lots of times in a directory, we do an `ls *.o` followed by `ls -l a.out`. Here is a script which does that

```
#!/bin/csh
# A very simple csh script!

/bin/ls *.o
/bin/ls -l a.out
```

To make this script executable, save it as `my_script` and issue the command `chmod u+x my_script` (for more on `chmod` see the section on nfs file permissions (p. 31)). Then run it with `my_script` alone on a line.

Here are a couple of things to note

1. Even this simple script has a comment!
2. Note I used `/bin/ls` rather than just plain `ls`. This means that any aliases for `ls` will be ignored, which will make the behaviour of the script portable. However, if you have `ls` aliased and don't realize it (which is the case with many people) this may produce unexpected results.

Grabbing Command Line Arguments

Command line arguments are passed into your shell script as special variables. There are two ways to address these variables, and thus grab command line args.

The first method is to use the variables `$n`, eg `$1` is the first argument. `$0` is the name of the script. You can use these variables as you would any other string, eg, as arguments to commands and the like.

The second method is to use the special array `$argv`. You can grab an argument as a member of this array, eg, `$argv[n]` is the nth argument. Note that you can also address `$#argv`, the total number of arguments. (This is an example of an unmentioned feature, which is for any array in the `csh`, `$#array` is the length of the array). Note that `$#array` returns the position of the last element, so it returns the number of elements - 1.

Here is an example of parsing command line args, shown by a simple script `ShowArgs`

```
#!/bin/csh
# An example of showing command line arguments with echo, and
# doing things with them with /bin/ls

# Method 1 uses dollar number
```

```
echo -----------------------------------------
echo Script: $0
echo
echo $1 was first and then $2 and $3
/bin/ls -l $1
echo -----------------------------------------

# Method 2 uses argv array
echo $argv[1] was the 1 out of $#argv args to $argv[0]
/bin/ls -l $argv[2]
echo -----------------------------------------
```

And here is the output:

```
hopper(pwalker).612 % ShowArgs ken.f ben.jpg addr.pl
----------------------------------------
Script: ShowArgs

ken.f was first and then ben.jpg and addr.pl
-rw-rw-rw-    1 pwalker  user      32243 Nov 21 09:40 ken.f
----------------------------------------
ken.f was the 1 out of 3 args to
-rw-r--r--    1 pwalker  user      34394 Nov  9 14:40 ben.jpg
----------------------------------------
```

Get it?

Using Variables

You can use variables in the csh for many things, but the three uses I will discuss here are

1. Getting environment information
2. Storing user-defined information
3. Getting command output with backticks

Getting information from the environment is trivial. If you want to use an environment variable `FOO` in your script, simply address it as `$FOO`, just like you would on the command line

Using variables to save user-defined information, such as executable locations, is a good habit since it eases portability and modification of your script. The archetypical example is to set a variable to point at an executable, then if the executable moves, your script only need change in one spot. A short example of this is

```
#!/bin/csh
# Run /usr/people/pwalker/mycmd from a script
set MYCMD = /usr/people/pwalker/mycmd
$MYCMD
```

I'd reccomend that you make all executables you call, even `ls` and `rm` variables to fully resolved paths, then you can avoid user aliases making unexpected script behaviour etc...

The final use is to store the output of backticks. For instance, an often seen construct is

```
set DATE = ‘date‘
```

Conditionals with `if`

Conditionals allow you to execute parts of code only if certain conditions are true. The conditional construct we will consider in the csh is:

```
if (condition) then
  statements
else
  statements
endif
```

where the else block is optional.

The only conditions we will discuss here are equality and non-equality among string variables, represented by `==` and `!=`. Examples of these are

"pwalker" == "pwalker"	true
"pwalker" == "johnsonb"	false
$DISPLAY == "hopper:0"	Checks value of $DISPLAY

As an example of this, I present this somewhat self-serving script:

```
#!/bin/csh
# compare my login with the login I *wish* I was

set ME         = ‘whoami‘
set IWISHIWERE = "pwalker"

if ($ME == $IWISHIWERE) then
  echo You are cool
else
  echo You are not as cool as $IWISHIWERE
endif
```

Now, when I run it, we see

```
hopper(pwalker).76 % if.sh
You are cool
```

but when user **web** runs it, we find:

```
farside(web).32 % if.sh
You are not as cool as pwalker
```

as expected.

Loops with foreach

Using foreach in a csh script is just as simple as using it on the command line, as described in the csh tricks and tips section (p. 10). You simply will not be prompted with a ? but will put your commands between the foreach() and the end.

Here is a simple example

```
#!/bin/csh

foreach C (‘ls‘)
echo $C
end
```

just as expected.

Writing to stdin with <<

The final aspect of shell scripting I want to mention is writing to the **stdin** of a process using the ¡¡ construct. The basic construction here is

```
command <<TAG
  stuff sent to stdin of command
TAG
```

There are many uses for this, including controlling programs which read stdin. The example I'm going to give here, though, will show how to originate an item of mail to a user based on a command line argument.

```
#!/bin/csh
# DemoEndDoc
# Send the size of $1 to user $2

# Use backticks to grab the file info
setenv RESULT = ‘/bin/ls -l testmxs‘

# Start up a "mail" process.
mail $2 <<ENDMAIL
Subject: Size of file \$1
```

```
Hi there $2.

Here is ls -l for $1

$RESULT

        - Paul
ENDMAIL

# End of script
```

You should be aware that their are some subtleties involved in whether or not you enclose your TAG in quotes or not. If you do, then variables will not be expanded in your input. However, if you are running into this sort of problem, you're probably doing something far too tricky for csh programming, and should rewrite it in `/bin/sh` or perl!

Also note the sneaky way I fake up the `Subject:` header in the outgoing mail.

Chapter 7

Miscellanous Other Things

Starting up with your . files

When you start a new shell, the shell will execute all the commands in your `~/.cshrc`, and when you log in for the first time, it sources your `~/.cshrc` and then your `~/.login`. Thus, users use the `~/.login` to set things they only need set once, such as terminal types, and their `~/.cshrc` to set things they need in all their processes, such as aliases.

If you change one of these files, you do not need to log out and then log back in. Rather you can simply `source` the file, eg `source ~/.cshrc`

Any of the commands we have discussed to day can be put in your . files. The most useful of these, though, is alias. For instance, many people have the following cannonical aliases:

```
alias ls ls -FC
alias rm rm -i
alias a alias
alias h history
alias m more
```

Of course, you can make up whatever aliases you like!

Also your `~/.cshrc` and `~/.login` set several important environment variables.

Your `path` gives a list of where the shell searches for executables.

Your `TERM` is your current terminal type.

Your `DISPLAY` tells where XWindows information is displayed.

Finally, many people set their prompt with the `.cshrc`. There are many ways to do this, and I won't explain it at length, but a personal favorite (which gives, for instance `hopper(pwalker).36 %`) is

```
set prompt = "`hostname | sed -e 's/\..*//'`($LOGNAME).\! % "
```

Seeing whats running and stopping it with ps and kill

`ps` shows processes which are currently running on a machine. `ps` has two basic varients, the SYSV and BSD. Suns are BSD, SGI's are SYSV.

Once again, this is a command with many options. See `man ps` for more info.

On a BSD-type machine, you can show all processes with `ps -aux`. Let's look at an example:

```
danube(pwalker).11 % ps -aux | grep pwalker
pwalker  16331 39.5  1.9  256  576 p7 R    14:35   0:00 ps -aux
pwalker  16314 17.1  1.6  104  504 p7 S    14:34   0:06 -csh (csh)
pwalker  16332  2.1  1.2   96  360 p7 S    14:35   0:00 grep pwalker
```

The most important fields are the user, PID (column 2) time (second-to-last) and process name.

On a SYSV-type machine, you can show all processes with `ps -ef`. eg,

```
farside(pwalker).20 % ps -ef | grep pwalker
 pwalker 24308 24307  0 09:06:08 pts/4    0:03 -csh
 pwalker 29735 29734  2 12:42:14 pts/10   0:01 -csh
 pwalker  4253 29735  9 14:39:08 pts/10   0:00 ps -ef
 pwalker  4254 29735  1 14:39:08 pts/10   0:00 grep pwalker
```

Once again, column 1 and 2 are the user and PID. Column 3 is the Parent (eg, executing process's) PID. The last two are time used and command.

The PID of the a process is very useful, since you can use the PID in conjunction with `kill` to kill a process. Varients of kill are `kill PID` to kill process with PID, `kill -9 PID` to kill it now (which is often unclean, but always effective) and `kill -9 -1` which kills all your processes including usually your current shell. So, I could kill one of my csh processes on farside with `kill -9 29735`.

See whos on with finger and who

`finger user@host` gives you information about `user` on `host` For instance,

```
farside(pwalker).179 % finger pwalker@loki
[loki.ncsa.uiuc.edu]
Login name: pwalker                     In real life: Paul Walker
Phone: 217-244-3008
Directory: /u/ncsa/pwalker              Shell: /bin/csh
On since Oct 17 13:04:31 on ttyq37 from hopper.ncsa.uiuc.edu
1 hour 16 minutes Idle Time
No Plan.
```

(Note: you can create a ~/.plan file which contains the a message to print rather than *No Plan*).

`finger @host` gives you information about all the users on host.

`finger` gives you information about all the users on localhost.

`who` is a program like finger, about which I will only say see `man who`. Note that the argument `am i` to who (eg, `who am i`) is useful, since it returns your username and tty information. Use `whoami` or `$LOGNAME` to get just your username.

See whats up with ping, rup, and uptime

Several utilities tell you the status of a machine in unix.

`ping` sends packets to a machine and expects them to come back. This is useful to see if your local machine can reach another. The syntax is simply `ping machine`.

`uptime` reports the uptime and load on your local machine. The output looks like:

```
farside(pwalker).193 % uptime
 2:51pm up 8 days, 6:22, 19 users, load average: 1.58, 1.70, 1.55
```

The most important number reported here is the load average, averaged over 1, 5, and 15 minutes (usually). The load average gives an idea of the number of processes running at once. On a single processor machine, a load of 1 is maximum efficient utilization. Loads more than the number of processors mean the machine is too heavily loaded.

`rup host` gives uptime information about a remote host in effectively the same format.

Seeing where it lives with which

Many commands you use are in your path. Sometimes it is useful to know where in your path they are. Hence the command `which`. For example,

```
farside(pwalker).195 % which xemacs
/usr/ncsa/bin/xemacs
```

This is often useful when a system has several copies of an executable, or you are trying to see when an executable was last modified. Note `which` will source your .cshrc before starting so it can handle aliases, eg

```
farside(pwalker).197 % which rm
rm aliased to "rm -i"
/bin/rm
```

Seeing whats left and used with du, df, and fs lq

Three commands help you manage your disk usage properly.

`du` tells you your total disk usage. Oft used options are `du -k` which makes it report in kB and `du -s` which simply sums in the current directory rather than recursing down directories. These can be combined into `du -sk`.

A usesful pipe to show you your usage with the most used directory at the top is

```
du -k | sort -n | tail -r | more
```

`df -k` tells you about kB free on NFS or mounted file systems. The usual report contains kbytes available and used, as well as a mount point (eg, directory name) for the file system.

`fs lq dir` tells you the afs quota available in `dir`.

Seeing whats changed with diff

You can compare two files for differences using `diff`. `diff` will tell you exactly where files differ and what the differences are. Take these two example files:

`ham.1`	`ham.2`
Hey. Shakespeare wrote in Hamlet: to be or not to be That is the question whether 'tis nobler to suffer the slings and arrows of outrageous fortune	Hey. Shakespeare wrote in Hamlet: 2b or not 2b That is the question whether 'tis nobler to suffer the slings and arrows of outrageous fortune

`diff` will point out that ham.2 has the alternate spelling of "to be"

```
hopper(pwalker).42 % diff ham.1 ham.2
4,5c4,5
< to be
< or not to be
---
> 2b
> or not 2b
hopper(pwalker).43 %
```

Note it tells you it is comparing lines 4,5 of file 1 to lines 4,5 of file 2, and shows the stuff from file 1 with a < and file 2 with an >.

Things I don't have time to mention

There are many other tools which are very useful. You should consider learning them. I don't have time for any details here, but will give a simple one line description of each.

make, gmake
These allow you to compile multi-file projects based on change dates, making sure your current build is up to date with your source. gmake is from the FSF, and is better than make for the most part

sccs, rcs
These allow you to have control of your source code with versioning, differencing versions, backing out versions and the like. rcs is from the FSF and is better than sccs for the most part.

pine, pico
pine and pico are the best terminal mailer and quick-and-dirty editor available, in my opinion. If a unix non-user needs to edit, let em use pico, if it is installed.

perl
The all-purpose super scripting/programing language.

ph
Access the U of I phone book on most UI systems.

Chapter 8

Concluding Comments

Of course, the material in this course did not cover even close to all the aspects of the concepts or commands mentioned. For that, I would suggest three resources.

First, just look over the shoulder of other Unix users you know. This is how you pick up many neat tricks. Be sure you ask em if it is OK first, of course!

Second, read the man pages for these commands. After this course, you should be able to understand them pretty easily. Remeber, all the information you could possible want about a command is just a **`man command`** away!

Finally, I **highly** recommend the book ***Unix in a Nutshell*** published by O'Reilly and Associates. They have it at all the bookstores in town, and it only costs about $10. This reference has all the information in this course plus much more in an easy to use concise reference. I would be lost without a copy floating around my office!