11 Shell scripting and more useful commands

By now you should be at least familiar if not quite friendly with the command line, able to pipe commands, redirect output and so forth. And even with alias you might have noticed that typing in a lot of the same commands over and over again is getting pretty old. For this reason you need shell scripts.

A shell script is a simple method of programming that enables anyone to use the power of the Unix command line to their advantage, while they tend to be a little idiosyncratic in their behaviour they are useful.

All you need for a shell script is the following template:

```
#!/bin/sh
# Your commands go under this line
```

The most important point is the first line must contain "`#!`" followed by the program you want to interpret the commands. After that you can just list a series of commands, just like you'd type in, each on its own line.

Common things to act as interpreters include "/bin/sh" (as it's fairly small and thus isn't a load on the system), "/bin/bash" (as many people like the way it interprets commands) and the shell which you default to "/usr/bin/tcsh" which at least will ensure that you can always test the commands properly.

Each shell has its own abilities to structure these files, including flow control like if statements, for the differences see the man pages for each. But most people should be fine using "/bin/sh" for simple scripts.

Anyway, after writing the script file simply chmod it (see section 4.8) so that you have execute permissions on it then it should run just like any other command on the system, this can be done with a command like:

`% chmod 755` scriptname

This uses the octal numeric modes (as described in `man chmod`) to change the permissions for the file "scriptname" and give the user rwx permissions and all other users `r-x` permissions (they need Read permissions for their interpreters to interpret the script and execute permissions to run it as a program). This could have been done with symbolic symbols like this:

`% chmod u=rwx,go=rx` scriptname

11.1 More useful commands

While this section won't offer many details on each of the programs listed it's more useful as a quick reference section, to find out the names of some of the more common programs and what they do, as if you need a filter to act in a certain way you can often convince a common program to do it for you instead of writing your own.

As ever see the man pages for details.

11.1.1 awk

awk is a "general scanning and processing language" to quote its man page. The most common use of awk is to divide the input into columns and only print certain columns, or to print them in a different order than they first appeared in (although it can do far more than this). An example to print the first and third columns of some input, with a tab between them would be:

```
% who | awk '{print $1 "\t" $3}'
```

11.1.2 tac

You've already been introduced to the "cat" command (see section 4.2.8), tac does the same thing as cat (in that it concatenates files) but it does so in the reverse order that the files were given to it. So if you use the command:

```
% tac afile bfile cfile > output
```

It will put all of "cfile", then "bfile" and finally "afile" into the file called "output". This is just handy for when you need to do a reverse concatenation, and don't want to write complex code to reverse the arguments.

11.1.3 grep

grep has been discussed on numerous occasions, but never really documented. Essentially it is a program that is designed to accept input of plain text and filter it, only outputting the lines that match a pattern. This makes it useful to search the results of other programs for just the sections you want.

There is also an enhanced version called "egrep" which accepts a wider range of patterns, and some important flags.

The absolute basics of pattern matching are that you can put '.*' (dot star) wildcard where you want to match anything, so:

```
% who | grep '.*foo.*'
```

Will grep for any string with "foo" in it anywhere, using ".*foo" searches for lines that end in "foo", whilst "foo.*" for ones that start in foo. This is because '.' means "any character" and '*' means "any number, even 0". For more details see the man pages for grep.

The example above was written in that style to make changing it for your own uses easier. However if you want to search for the pattern ".*foo.*" (i.e. the word foo, at any place in any line) you can just use:

```
% who | grep foo
```

And grep will automatically search for any occurrences of "foo" anywhere in the input. Indeed even `who | grep 'foo'` would produce the same results.

The advanced version of grep, egrep, can do some powerful pattern matching. However one of its most useful features is the '`-i`' flag. This makes its pattern matching case insensitive, so:

```
% egrep -i bar filename | less
```

Will run egrep in case insensitive mode ('`-i`'), searching the file called "filename" for any line with "bar" in it anywhere. It will then pipe the results out to "less" for viewing.

Because `-i` was used this command will also find "BAR", "Bar", "bAr" and any other combination of case.

grep and egrep can be useful tools for more than just searching for text and looking at whats found. They can be used as the basis for useful one line script utilities. For example if you had a directory called ~/Mail/ in your home directory that contained many files with email stored in, and you wanted to know how many messages you had overall you could do the following:

```
% egrep -r "^From .*" ~/Mail/* | wc -l
```

This relies on the knowledge that all mail messages in mbox formatted mail files (which is the most common format for email programs) start with a line that matches the patter "^From .*" (i.e. the word "from" at the start of a line then a space, followed by any number of any characters). This is generally followed by an account name, and a timestamp and is whats known as the envelope header (see "`man mbox`" for more). What this does is search recursively for all lines that fit that pattern ('^' means "start of the line") in all files (the '*' means all files inside the ~/Mail/ directory) then count the number of lines of output that egrep produces using wc. Since it'll output one line per email this will give you a total number of emails in total.

egrep has a `-c` (dash c) flag that means it will only output the total number of matches, and not the matches themselves. However since the example above was counting the output from multiple mailboxes (every file in ~/Mail) then this wouldn't have worked. However to count the number of mails in a single mailbox you should be able to use:

```
% egrep -c '^From .*' $MAIL
```

Which will match the same pattern as the example above, but will only do it to the mailbox contained in the variable $MAIL (your system inbox) and will print out the number of matches, instead of matched lines (the `-c` option).

11.1.4 head/tail

head and tail have already been encountered in passing (see section 10.1 for examples with them in). Essentially tail can get you the last n lines (defaulting to about 20) of a file or a programs output, head does the opposite, getting you the first N lines. These programs are handy to filter output, but tail also has another good feature:

```
% tail -f filename
```

The '`-f`' option means that tail will output the last few lines of the file "filename" then will stay running and any new input that goes into it will be outputted to the terminal. This makes it handy to watch log files generated by a programming running in another terminal as they are created.

11.1.5 wc

Word Count was detailed in section 5 but has another often overlooked behaviour. Since it can count lines of a file it can be attached to the end of a pipe-line of other commands to count how much output there is. This will allow you to count the number of occurrences of things, as illustrated by the following example:

```
% grep foo filename | wc -l
```

This will grep the file called "filename" for lines containing "foo" and instead of outputting them will simply count the number and print that. This is mostly handy for auditing log files.

11.1.6 bc

bc is an "arbitrary precision arithmetic language", basically it is a calculator for the command line, and can be used in scripts. While its operation is actually rather complex (see "`man bc`" and "`man dc`" for a related program) it can be used quite simply as the following example illustrates:

```
% echo "3 - 2" | bc
```

Echoing simple strings into it that contain normal arithmetic symbols (see "`man bc`") results in it returning the results, here it would print "1" to the command line.

If you are using the bash shell (see section 13.12) you could alternatively do:

```
% echo $(( 3 - 2))
```

To use its inbuilt maths functions. This is generally fractionally quicker from within bash as it doesn't need to invoke a seperate `bc` process.

11.1.7 sed

sed is a "Stream EDitor" it is essentially an editor that is designed to change parts of "streams of bytes" (see section 10.1) as it passes. Essentially you can write regular expressions (see 11.1.3) to spot patterns then change them to something else.

A simple example of this would if you wanted to see a list of all the hosts that are being used to remotely log into a machine:

```
% who | grep '(' | sed -e 's/.*(//' -e 's/).*$//' | sort | uniq | less
```

So what this does is runs who (to see who's logged in), and if you look at the output of who you'll notice that the last column contains the hostname of where that user is logged on from, which is surrounded by two brackets in the format "(hostname)".

So the first thing to do is pass it to grep, and grep for '`(`' which is every line with the an open bracket character '`(`' in. After this point only the lines which deal with remote users will be shown.

Then the results of that grep are passed to sed, and sed executes two commands to replace text, each is proceeded by a -e argument, and the whole replacement is shown in single quotes `' '`.

The first one reads "`s/.*(//`" which means a substitution (the 's') of the first part (contained inside forward slashes) for the second part. The first part is ".*`(`" which means anything followed by a `(` symbol. The second part is just //, since there is nothing between the two forward slash then ".*`(`" is replaced by nothing (i.e. it is deleted).

The second replacement reads "`s/).*//`' which looks for a `)` (closing bracket) followed by anything '`.*`' and replaces it again with the nothingness between the two forward slashes.

Now that we've just got a large list of hostnames these are passed to the "uniq" program (see section 11.1.17, note they also are passed through "sort" (see section 11.1.9), as uniq requires sorted input) which ensures there are no duplicates in the list, and passes its results to "less" for viewing.

11.1.8 tr

tr is used to "translate characters", that is to do read a stream of input and swap one character for another (or sequences for other sequences). Anything that tr can do can also be done by sed (see section 11.1.7) but tr can often do it more easily. For an example there is often a command called "users" that prints out a simple list of all the users logged into a machine, separated by spaces. However cent1 doesn't have a users command, so we have to make our own:

```
% who | awk '{print $1}' | sort | tr '\n' ' '
```

What this does is get the who list, pass it through awk to get the first column only (the username) and pass that through sort to alphabetise it. All that should be fairly familiar from earlier examples. However it then passes it through tr.

What tr does is exchange the characters inside the first quotes for those in the second. In this example there is a "`\n`" inside the first quotes, which generally means "a new line", the second quotes simply contain " " (a space).

So tr simply gets all the input from sort and swaps all line endings for spaces, thus giving us our space separated list.

tr's simple syntax allows for quick editing jobs on the command-line where only very simple things need to be changed. For example if you had a file full of data, all of which was separated by tab characters, and you needed to separate them with comma's (tab and comma separated files are actually fairly common) you could use:

```
% tr '\t' ',' < tabbed-file > comma-file
```

This will input the file "tabbed-file" into the standard input of tr (see sections 10.2 if this syntax is confusing). tr will then change every tab character (shown by the standard escape '
t') into a comma and then send the output to comma-file.

Another useful feature of tr is the '`-d`' flag. This is used to simply remove characters instead of exchanging them for another character, as shown in this example:

```
% ls -l | grep "^-" | wc -l | tr -d " "
```

This prints out the number of files in the current directory. It does this by doing an '`ls -l`' to get the long list of files in the current directory, grepping that for a '`-`' (dash) character as files start with that (directories start with 'd'). It then word counts the number of lines (thus the number of files) and finally uses tr to simply remove all the space characters, thus making the number of files not have the space prefix that wc adds.

As a final example of using `tr` lets return to the example of `sed` above. The same thing could be done using `tr` as follows:

```
% who | grep '(' | awk '{print $6}' | tr -d '()' | sort | uniq | less
```

Here you can see it greps the output of `who`, gets the last column with `awk`, removes the brackets with `tr`'s `-d` flag, sorts it, removes duplicates then makes it viewable in `less`. Learning when its best to use `tr` and when to use `sed` is generally a matter of experience, but for simple things like this `tr` is often better, sed however is far more powerful and better suited to anything even a little more complex.

11.1.9 sort

`sort` has already been encountered in numerous examples, but to reiterate it simply takes input in, sorts it either alphabetically or numerically and then outputs it. Its handy for formatting output before showing it to the user, or for sorting lists that then get head or tailed.

sort also supports a '`-r`' option to reverse its behaviour, and its man pages contain details of how to create more complex sorting rules.

`sort` supports a rather useful feature for frequent scripters. With the use of the '`-u`' flag (unique) then it will function the same as using "`sort | uniq`" and print only the first occurrence of clumps of similar results. Using "`sort -u`" is the more efficient method as it saves you having to run the separate uniq process.

11.1.10 date

`date` does exactly that, it outputs the current date in a standard manner (for example, the current date is: "Fri Aug 6 16:46:01 BST 2004"). However date can be used to get other formatted time, for example:

```
% date +%H:%M
```

Will print out the hour in 24 hour format (that's the %H) then a : then the minutes (that's the %M). The rule of thumb is that you need a + before the string describing the output, and any symbols inside it preceeded by %

(percent) symbols will be replaced by what they mean e.g. %H for hour, %M for minute, %B is the full month name.

"`man strftime`" contains the full list of % codes for date (usually "`man date`" would, but with Solaris this is not the case).

11.1.11 diff

`diff` is used to find differences between two byte streams (either two files, or a file and the standard input to diff). The idea is that it if you have two very similar files (say you copied a file, then made changes to the copy) you could use diff to spot the differences without needing to do it by hand, or write a horrible script to compare them line by line. diff's use is fairly simple, just use:

```
% diff foo bar
```

And `diff` will print out the differences between the file "foo" and the file "bar". Stuff from the first file will be prefixed by '`<`', and stuff from the second file prefixed by '`>`'.

See "`man diff`" for full details.

11.1.12 tee

`tee` is a useful command to know about in scripting, as it can be used to dump the results half way through a long command line (amongst other things). Primarily it can be used to write the input it gets to both a file and its output. For example:

```
% who | tee who-list | grep $USER
```

This runs who as usual, and hands the output of who over to tee. tee writes all of its input to the file "who-list" and then outputs all of it to grep, grep then greps for your username. So the user will see only their own user-details outputted to their terminal but the full who list stored in a file in the current directory.

Again, see the man pages before using this program.

11.1.13 md5sum

Assuming you want to be sure that one file is exactly the same as the other you could use diff and see if it returns anything, or you could use a program called md5sum. This will generate checksums of the file using the md5 algorithm. This guarantees (for all practical purposes) that if the two checksums are the same then the two files will be.

Its use is fairly simple, for example to generate the checksum for a file called "filename" type:

```
% md5sum filename
```

You will then see a very long string of numbers and characters followed by some space, then the filename. This is the checksum.

11.1.14 xargs

`xargs` is a program you've already seen the use of (see section 10.3). Essentially what it does is take a program as an argument, then make its standard input the arguments for that program. While this may sound a little confusing consider:

```
% ps -U $USER | grep vim | awk '{print $1}' | xargs kill
```

What this does is search the processes list for all your processes (see section 7.4), grep that for a specific program (in this case vim), and pass that list to awk to print out the first column, which will be the PID (see section 7). So at this point xargs comes into play.

`xargs` has a single number for its input, which is the PID of the process you want to kill (this example has a flaw in that if there is more than one processes running by that name it'll probably not work) and its one argument is the string "kill".

This means that it will run the command "kill" and give the kill command its input (the PID) so kill will then issue a TERM signal to that PID, and kill the process.

As ever see the man pages, xargs can be helpful in getting around problems of flow in scripting.

11.1.15 split

split is a program used to literally split a file, or its input, into sections. It outputs each section to a file called the same as the originally file followed by "aa" and so on. It can divide files by bytes, kilobytes, megabytes or line count (for textual files). See "`man split`" for details.

`cat` (see section 4.2.8) is probably the easiest way to reassemble them afterwards.

11.1.16 time

If you're ever interested in how long a command takes to run you can simply use the "time" command, as in this example:

```
% time who
```

As told in its man pages time will output the "real" time the command took to run, then two times relating to the CPU time used, see the man pages for details.

11.1.17 uniq

`uniq` is a very simple utility, shown already in multiple examples. It simply takes its input (which must be sorted) and removes any duplicate lines it encounters. This is handy for situations where you only want the output to contain one instance of that thing (e.g. a username list).

The man pages contain its full instructions, it can optionally display a count of how many times each line was in the input, or only output non-repeated lines.

As shown in section 11.1.9 the program sort can perform the job of a "`sort | uniq`" pipeline by the use of "`sort -u`", and for this reason you will more often use `sort -u` than uniq. However if you have a program that generates already sorted output then uniq can be helpful.

11.1.18 join

The join command is similar to the relational database operator (for those of you familiar with that), essentially it can take either two files, or one file and its input and run a "join" against them, only outputting lines that are in both. For this reason it's important that both files (and any input involved) has been passed through "sort" (see section 11.1.9) previously, as otherwise ordering will cause this command to be useless.

See "`man join`" for more information.