UNIX Administration Course

Copyright 1999 by Ian Mapleson BSc.

Version 1.0

`mapleson@gamers.org`
`Tel: (+44) (0)1772 893297`
`Fax: (+44) (0)1772 892913`
`WWW: http://www.futuretech.vuurwerk.nl/`

Detailed Notes for Day 2 (Part 2)

UNIX Fundamentals: Shell scripts.

It is an inevitable consequence of using a command interface such as shells that one would wish to be able to run a whole sequence of commands to perform more complex tasks, or perhaps the same task many times on multiple systems.

Shells allow one to do this by creating files containing sequences of commands. The file, referred to as a shell script, can be executed just like any other program, though one must ensure the execute permissions on the file are set appropriately in order for the script to be executable.

Large parts of all modern UNIX variants use shell scripts to organise system management and behaviour. Programming in shell script can include more complicated structures such as if/then statements, case statements, for loops, while loops, functions, etc. Combined with other features such as metacharacters and the various text-processing utilities (perl, awk, sed, grep, etc.) one can create extremely sophisticated shell scripts to perform practically any system administration task, ie. one is able to write programs which can use any available application or existing command as part of the code in the script. Since shells are based on C and the commands use a similar syntax, shell programming effectively combines the flexibility of C-style programming with the ability to utilise other programs and resources within the shell script code.

Looking at typical system shell script files, eg. the bootup scripts contained in /etc/init.d, one can see that most system scripts make extensive use of if/then expressions and case statements. However, a typical admin will find it mostly unnecessary to use even these features. In fact, many administration tasks one might choose to do can be performed by a single command or sequence of commands on a single line (made possible via the various metacharacters). An admin might put such mini-scripts into a file and execute that file when required; even though the file's contents may not appear to be particularly complex, one can perform a wide range of tasks using just a few commands.

A hash symbol '#' in a script file at the beginning of a line is used to denote a comment.

One of the most commonly used commands in UNIX is 'find' which allows one to search for files, directories, files belonging to a particular user or group, files of a special type (eg. a link to another file), files modified before or after a certain time, and so on (there are many options). Most admins tend to use the find command to select certain files upon which to perform some other operation, to locate files for information gathering purposes, etc.

The find command uses a Boolean expression which defines the type of file the command is to search for. The name of any file matching the Boolean expression is returned.

For example (see the ’find’ man page for full details):

```
find /home/students -name "capture.mv" -print
```

Figure 25. A typical find command.

This command searches all students directories, looking for any file called ’capture.mv’. On Indy systems, users often capture movie files when first using the digital camera, but usually never delete them, wasting disk space. Thus, an admin might have a site policy that, at regular intervals, all files called capture.mv are erased - users would be notified that if they captured a video sequence which they wished to keep, they should either set the name to use as something else, or rename the file afterwards.

One could place the above command into a executable file called ’loc’, running that file when one so desired. This can be done easily by the following sequence of actions (only one line is entered in this example, but one could easily enter many more):

```
% cat > loc
find /home/students -name "capture.mv" -print
[press CTRL-D]
% chmod u+x loc
% ls -lF loc
-rwxr--r--    1 mapleson staff          46 May  3 13:20 loc*
```

Figure 26. Using cat to quickly create a simple shell script.

Using ls -lF to examine the file, one would see the file has the execute permission set for user, and a ’*’ has been appended after the file name, both indicating the file is now executable. Thus, one could run that file just as if it were a program. One might imagine this is similar to .BAT files in DOS, but the features and functionality of shell scripts are very different (much more flexible and powerful, eg. the use of pipes).

There’s no reason why one couldn’t use an editor to create the file, but experienced admins know that it’s faster to use shortcuts such as employing cat in the above way, especially compared to using GUI actions which requires one to take hold the mouse, move it, double-click on an icon, etc. Novice users of UNIX systems don’t realise until later that very simple actions can take longer to accomplish with GUI methods.

Creating a file by redirecting the input from cat to a file is a technique I often use for typing out files with little content. cat receives its input from stdin (the keyboard by default), so using ’cat > filename’ means anything one types is redirected to the named file instead of stdout; one must press CTRL-D to end the input stream and close the file.

An even lazier way of creating the file, if just one line was required, is to use echo:

```
% echo ’find /home/students -name "capture.mv" -print’ > loc
% chmod u+x loc
% ls -lF loc
-rwxr--r--    1 mapleson staff          46 May  3 13:36 loc
% cat loc
find /home/students -name "capture.mv" -print
```

Figure 27. Using echo to create a simple one-line shell script.

This time, there is no need to press CTRL-D, ie. the prompt returns immediately and the file has been created. This happens because, unlike cat which requires an ’end of file’ action to terminate the input, echo’s input terminates when it receives an end-of-line character instead (this behaviour can be overridden with the ’-n’ option).

The man page for echo says, "echo is useful for producing diagnostics in command files and for sending known data into a pipe."

For the example shown in Fig 27, single quote marks surrounding the find command were required. This is because, without the quotes, the double quotes enclosing capture.mv are not included in the output stream which is redirected into the file. When contained in a shell script file, find doesn’t need double quotes around the file name to search for, but it’s wise to include them because other characters such as * have special meaning to a shell. For example, without the single quote marks, the script file created with echo works just fine (this example searches for any file beginning with the word ’capture’ in my own account):

```
% echo find /mapleson -name "capture.*" -print > loc
% chmod u+x loc
% ls -lF loc
-rwxr--r--    1 mapleson staff         38 May  3 14:05 loc*
% cat loc
find /mapleson -name capture.* -print
% loc
/mapleson/work/capture.rgb
```

Figure 28. An echo sequence without quote marks.

Notice the loc file has no double quotes. But if the contents of loc is entered directly at the prompt:

```
% find /mapleson -name capture.* -print
find: No match.
```

Figure 29. The command fails due to * being treated as a metacommand by the shell.

Even though the command looks the same as the contents of the loc file, entering it directly at the prompt produces an error. This happens because the * character is interpreted by the shell before the find command, ie. the shell tries to evaluate the capture.* expression for the current directory, instead of leaving the * to be part of the find command. Thus, when entering commands at the shell prompt, it’s wise to either use double quotes where appropriate, or use the backslash \ character to tell the shell not to treat the character as if it was a shell metacommand, eg.:

```
% find /mapleson -name capture.\* -print
/mapleson/work/capture.rgb
```

Figure 30. Using a backslash to avoid confusing the shell.

A -exec option can be used with the find command to enable further actions to be taken on each result found, eg. the example in Fig 25 could be enhanced by including making the find operation execute a further command to remove each capture.mv file as it is found:

```
find /home/students -name "capture.mv" -print -exec /bin/rm {} \;
```

Figure 31. Using find with the -exec option to execute rm.

Any name returned by the search is passed on to the rm command. The shell substitutes the {}

symbols with each file name result as it is returned by find. The \; grouping at the end serves to terminate the find expression as a whole (the ; character is normally used to terminate a command, but a backslash is needed to prevent it being interpreted by the shell as a metacommand).

Alternatively, one could use this type of command sequence to perform other tasks, eg. suppose I just wanted to know how large each movie file was:

```
find /home/students -name "capture.mv" -print -exec /bin/ls -l {} \;
```

Figure 32. Using find with the -exec option to execute ls.

This works, but two entries will be printed for each command: one is from the -print option, the other is the output from the ls command. To see just the ls output, one can omit the -print option.

Consider this version:

```
find /home/students -name "*.mov" -exec /bin/ls -l {} \; > results
```

Figure 33. Redirecting the output from find to a file.

This searches for any .mov movie file (usually QuickTime movies), with the output redirected into a file. One can then perform further operations on the results file, eg. one could search the data for any movie that contains the word ’star’ in its name:

```
grep star results
```

A final change might be to send the results of the grep operation to the printer for later reading:

```
grep star results | lp
```

Thus, the completed script looks like this:

```
find /home/students -name "*.mv" -exec /bin/ls -l {} \; > results
grep star results | lp
```

Figure 34. A simple script with two lines.

Only two lines, but this is now a handy script for locating any movies on the file system that are likely to be related to the Star Wars or Star Trek sagas and thus probably wasting valuable disk space! For the network I run, I could then use the results to send each user a message saying the Star Wars trailer is already available in /home/pub/movies/misc, so they’ve no need to download extra copies to their home directory.

It’s a trivial example, but in terms of the content of the commands and the way extra commands are added, it’s typical of the level of complexity of most scripts which admins have to create.

Further examples of the use of ’find’ are in the relevant man page; an example file which contains several different variations is:

```
/var/spool/cron/crontabs/root
```

This file lists the various administration tasks which are executed by the system automatically on a regular basis. The cron system itself is discussed in a later lecture.

WARNING. The Dangers of the Find Command and Wildcards.

Although UNIX is an advanced OS with powerful features, sometimes one encounters an aspect of its operation which catches one completely off-guard, though this is much less the case after just a little experience.

A long time ago (January 1996), I realised that many students who used the Capture program to record movies from the Digital Camera were not aware that using this program or other movie-related programs could leave unwanted hidden directories containing temporary movie files in their home directory, created during capture, editing or conversion operations (I think it happens when an application is killed of suddenly, eg. with CTRL-C, which doesn’t give it an opportunity to erase temporary files).

These directories, which are always located in a user’s home directory, are named ’.capture.mv.tmpXXXXX’ where XXXXX is some 5-digit string such as ’000Hb’, and can easily take up many megabytes of space each.

So, I decided to write a script to automatically remove such directories on a regular basis. Note that I was logged on as root at this point, on my office Indy.

In order to test that a find command would work on hidden files (I’d never used the find command to look for hidden files before), I created some test directories in the /tmp directory, whose contents would be given by ’ls -AR’ as something like this:

```
% ls -AR
.b/  .c/  a/   d/
./.b:

./.c:
.b   a

./a:

./d:
a
```

ie. a simple range of hidden and non-hidden directories with or without any content:

- Ordinary directories with or without hidden/non-hidden files inside,
- Hidden directories with or without hidden/non-hidden files inside,
- Directories with ordinary files,
- etc.

The actual files such as .c/a and .c/.b didn’t contain anything. Only the names were important for the test.

So, to test that find would work ok, I executed the following command from within the /tmp directory:

```
     find . -name ".*" -exec /bin/rm -r {} \;
```

(NB: the -r option for rm means do a recursive removal, and note that there was no -i option used with the rm here)

What do you think this find command would do? Would it remove the hidden directories .b and .c and their contents? If not, why not? Might it do anything else as well?

Nothing happened at first, but the command did seem to be taking far too long to return the shell prompt. So, after a few seconds, I decided something must have gone wrong; I typed CTRL-C to stop the find process (NB: it was fortunate I was not distracted by a phone call or something at this point).

Using the ls command showed the test files I'd created still existed, which seemed odd. Trying some further commands, eg. changing directories, using the 'ps' command to see if there was something causing system slowdown, etc., produced strange errors which I didn't understand at the time (this was after only 1 or 2 months' admin experience), so I decided to reboot the system.

The result was disaster: the system refused to boot properly, complaining about swap file errors and things relating to device files. Why did this happen?

Consider the following command sequence by way of demonstration:

```
     cd /tmp
     mkdir xyz
     cd xyz
     /bin/ls -al
```

The output given will look something like this:

```
     drwxr-xr-x    2 root     sys            9 Apr 21 13:28 ./
     drwxrwxrwt    6 sys      sys          512 Apr 21 13:28 ../
```

Surely the directory xyz should be empty? What are these two entries? Well, not quite empty. In UNIX, as stated in a previous lecture, virtually everything is treated as a file. Thus, for example, the command so commonly performed even on the DOS operating system:

```
     cd ..
```

is actually doing something rather special on UNIX systems. 'cd ..' is not an entire command in itself. Instead, every directory on a UNIX file system contains two hidden directories which are in reality special types of file:

```
     ./        - this refers to the current directory.
     ../       - this is effectively a link to the
                 directory above in the file system.
```

So typing 'cd ..' actually means 'change directory to ..' (logical since cd does mean 'change directory to') and since '..' is treated as a link to the directory above, then the shell changes the current working directory to the next level up.

[by contrast, 'cd ..' in DOS is treated as a distinct command in its own right - DOS recognises the presence of '..' and if possible changes directory accordingly; this is why DOS users can type 'cd..' instead if desired]

But this can have an unfortunate side effect if one isn’t careful, as is probably becoming clear by now. The ".*" search pattern in the find command will *also* find these special ’./’ and ’../’ entries in the /tmp directory, ie.:

- The first thing the find command locates is ’./’
- ’./’ is inserted into the search string ".*" to give "../*"
- find changes directory to / (root directory). Uh oh...
- find locates the ./ entry in / and substitutes this string into ".*" to give "../*". Since the current directory cannot be any higher, the search continues in the current directory; ../ is found next and is treated the same way.
- The -exec option with ’rm’ causes find to begin erasing hidden files and directories such as .Sgiresources, eventually moving onto non-hidden files: first the /bin link to /usr/bin, then the /debug link, then all of /dev, /dumpster, /etc and so on.

By the time I realised something was wrong, the find command had gone as far as deleting most of /etc. Although important files in /etc were erased which I could have replaced with a backup tape or reinstall, the real damage was the erasure of the /dev directory. Without important entries such as /dev/dsk, /dev/rdsk, /dev/swap and /dev/tty*, the system cannot mount disks, configure the swap partition on bootup, connect to keyboard input devices (tty terminals), and accomplish other important tasks.

In other words, disaster. And I’d made it worse by rebooting the system. Almost a complete repair could have been done simply by copying the /dev and /etc directories from another machine as a temporary fix, but the reboot made everything go haywire. I was partly fooled by the fact that the files in /tmp were still present after I’d stopped the command with CTRL-C. This led me to at first think that nothing had gone awry.

Consulting an SGI software support engineer for help, it was decided the only sensible solution was to reinstall the OS, a procedure which was alot simpler than trying to repair the damage I’d done.

So, the lessons learned:

- Always read up about a command before using it. If I’d searched the online books with the expression ’find command’, I would have discovered the following paragraph in Chapter 2 ("Making the Most of IRIX") of the ’IRIX Admin: System Configuration and Operation’ manual:

  > "Note that using recursive options to commands can be very dangerous in that the command automatically makes changes to your files and file system without prompting you in each case. The chgrp command can also recursively operate up the file system tree as well as down. Unless you are sure that each and every case where the recursive command will perform an action is desired, it is better to perform the actions individually. Similarly, it is good practice to avoid the use of metacharacters (described in "Using Regular Expressions and Metacharacters") in combination with recursive commands."

I had certainly broken the rule suggested by the last sentence in the above paragraph. I also did not know what the command would do before I ran it.

- Never run programs or scripts with as-yet unknown effects as root.

  ie. when testing something like removing hidden directories, I should have logged on as some ordinary user, eg. a ’testuser’ account, so that if the command went wrong it would not have been able to change or remove any files owned by root, or files owned by anyone else for that matter, including my own in /mapleson. If I had done this, the command I used would have given an immediate error and halted when the find string tried to remove the very first file found in the root directory (probably some minor hidden file such as .Sgiresources).

Worrying thought: if I hadn’t CTRL-C’d the find command when I did, after enough time, the command would have erased the entire file system (including /home), or at least tried to. I seem to recall that, in reality (tested once on a standalone system deliberately), one can get about as far as most of /lib before the system actually goes wrong and stops the current command anyway, ie. the find command sequence eventually ends up failing to locate key libraries needed for the execution of ’rm’ (or perhaps the ’find’ itself) at some point.

The only positive aspects of the experience were that, a) I’d learned alot about the subtleties of the find command and the nature of files very quickly; b) I discovered after searching the Net that I was not alone in making this kind of mistake - there was an entire web site dedicated to the comical mess-ups possible on various operating systems that can so easily be caused by even experienced admins, though more usually as a result of inexperience or simple errors, eg. I’ve had at least one user so far who has erased their home directory by mistake with ’rm -r *’ (he’d thought his current working directory was /tmp when in fact it wasn’t). A backup tape restored his files.

Most UNIX courses explain how to use the various available commands, but it’s also important to show how *not* to use certain commands, mainly because of what can go wrong when the root user makes a mistake. Hence, I’ve described my own experience of making an error in some detail, especially since ’find’ is such a commonly used command.

As stated in an earlier lecture, to a large part UNIX systems run themselves automatically. Thus, if an admin finds that she/he has some spare time, I recommend using that time to simply read up on random parts of the various administration manuals - look for hints & tips sections, short-cuts, sections covering daily advice, guidance notes for beginners, etc. Also read man pages: follow them from page to page using xman, rather like the way one can become engrossed in an encyclopedia, looking up reference after reference to learn more.

A Simple Example Shell Script.

I have a script file called ’rebootlab’ which contains the following:

```
rsh akira init 6&
rsh ash init 6&
rsh cameron init 6&
rsh chan init 6&
rsh conan init 6&
rsh gibson init 6&
rsh indiana init 6&
rsh leon init 6&
rsh merlin init 6&
rsh nikita init 6&
rsh ridley init 6&
rsh sevrin init 6&
rsh solo init 6&
#rsh spock init 6&
rsh stanley init 6&
rsh warlock init 6&
rsh wolfen init 6&
rsh woo init 6&
```

Figure 35. The simple rebootlab script.

The rsh command means ’remote shell’. rsh allows one to execute commands on a remote system by establishing a connection, creating a shell on that system using one’s own user ID information, and then executing the supplied command sequence.

The init program is used for process control initialisation (see the man page for details). A typical use for init is to shutdown the system or reboot the system into a particular state, defined by a number from 0 to 6 (0 = full shutdown, 6 = full reboot) or certain other special possibilities.

As explained in a previous lecture, the ’&’ runs a process in the background.

Thus, each line in the file executes a remote shell on a system, instructing that system to reboot. The init command in each case is run in the background so that the rsh command can immediately return control to the rebootlab script in order to execute the next rsh command.

The end result? With a single command, I can reboot the entire SGI lab without ever leaving the office.

Note: the line for the machine ’spock’ is commented out. This is because the Indy called spock is currently in the technician’s office, ie. not in service. This is a good example of where I could make the script more efficient by using a for loop, something along the lines of: for each name in this list of names, do <command>.

As should be obvious, the rebootlab script makes no attempt to check if anybody is logged into the system. So in practice I use the rusers command to make sure nobody is logged on before executing the script. This is where the script could definitely be improved: the command sent by rsh to each system could be modified with some extra commands so that each system is only rebooted if nobody is logged in at the time (the ’who’ command could probably be used for this, eg. ’who | grep -v root’ would give no output if nobody was logged on).

The following script, called ’remountmapleson’, is one I use when I go home in the evening, or perhaps at lunchtime to do some work on the SGI I use at home.

```
rsh yoda umount /mapleson && mount /mapleson &
rsh akira umount /mapleson && mount /mapleson &
rsh ash umount /mapleson && mount /mapleson &
rsh cameron umount /mapleson && mount /mapleson &
rsh chan umount /mapleson && mount /mapleson &
rsh conan umount /mapleson && mount /mapleson &
rsh gibson umount /mapleson && mount /mapleson &
rsh indiana umount /mapleson && mount /mapleson &
rsh leon umount /mapleson && mount /mapleson &
rsh merlin umount /mapleson && mount /mapleson &
rsh nikita umount /mapleson && mount /mapleson &
rsh ridley umount /mapleson && mount /mapleson &
rsh sevrin umount /mapleson && mount /mapleson &
rsh solo umount /mapleson && mount /mapleson &
#rsh spock umount /mapleson && mount /mapleson &
rsh stanley umount /mapleson && mount /mapleson &
rsh warlock umount /mapleson && mount /mapleson &
rsh wolfen umount /mapleson && mount /mapleson &
rsh woo umount /mapleson && mount /mapleson &
```

Figure 36. The simple remountmapleson script.

When I leave for home each day, my own external disk (where my own personal user files reside) goes with me, but this means the mount status of the /mapleson directory for every SGI in Ve24 is now out-of-date, ie. each system still has the directory mounted even though the file system which was physically mounted from the remote system (called milamber) is no longer present. As a result, any attempt to access the /mapleson directory would give an error: "Stale NFS file handle." Even listing the contents of the root directory would show the usual files but also the error as well.

To solve this problem, the script makes every system unmount the /mapleson directory and, if that was successfully done, remount the directory once more. Without my disk present on milamber, its /mapleson directory simply contains a file called ’README’ whose contents state:

> Sorry, /mapleson data not available - my external disk has been temporarily removed. I’ve probably gone home to work for a while. If you need to contact me, please call 888026.

As soon as my disk is connected again and the script run once more, milamber’s local /mapleson contents are hidden by my own files, so users can access my home directory once again.

Thus, I’m able to add or remove my own personal disk and alter what users can see and access at a global level without users ever noticing the change.

Note: the server still regards my home directory as /mapleson on milamber, so in order to ensure that I can always logon to milamber as mapleson even if my disk is not present, milamber’s /mapleson directory also contains basic .cshrc, .login and .profile files.

Yet again, a simple script is created to solve a particular problem.

Command Arguments.

When a command or program is executed, the name of the command and any parameters are passed to the program as arguments. In shell scripts, these arguments can be referenced via the ’$’ symbol. Argument 0 is always the name of the command, then argument 1 is the first parameter, argument 2 is the second parameter, etc. Thus, the following script called (say) ’go’:

```
echo $0
echo $1
echo $2
```

would give this output upon execution:

```
% go somewhere nice
go
somewhere
nice
```

Including extra echo commands such ’echo $3’ merely produces blank lines after the supplied parameters are displayed.

If one examines any typical system shell script, this technique of passing parameters and referencing arguments is used frequently. As an example, I once used the technique to aid in the processing of a large number of image files for a movie editing task. The script I wrote is also typical of the general complexity of code which most admins have to deal with; called ’go’, it contained:

```
subimg $1 a.rgb 6 633 6 209
gammawarp a.rgb m.rgb 0.01
mult a.rgb a.rgb n.rgb
mult n.rgb m.rgb f.rgb
addborder f.rgb b.rgb x.rgb
subimg x.rgb ../tmp2/$1 0 767 300 875
```

(the commands used in this script are various image processing commands that are supplied as part of the Graphics Library Image Tools software subsystem. Consult the relevant man pages for details)

The important feature is the use of the $1 symbol in the first line. The script expects a single parameter, ie. the name of the file to be processed. By eventually using this same argument at the end of an alternative directory reference, a processed image file with the same name is saved elsewhere after all the intermediate processing steps have finished. Each step uses temporary files created by previous steps.

When I used the script, I had a directory containing 449 image files, each with a different name:

```
i000.rgb
i001.rgb
i002.rgb
    .
    .
    .
i448.rgb
```

To process all the frames in one go, I simply entered this command:

```
find . -name "i*.rgb" -print -exec go {} \;
```

As each file is located by the find command, its name is passed as a parameter to the go script. The use of the -print option displays the name of each file before the go script begins processing the file’s contents. It’s a simple way to execute multiple operations on a large number of files.

Secure/Restricted Shell Scripts.

It is common practice to include the following line at the start of a shell script:

```
#!/bin/sh
```

This tells any shell what to use to interpret the script if the script is simply executed, as opposed to sourcing the script within the shell.

The ’sh’ shell is a lower level shell than csh or tcsh, ie. it’s more restricted in what it can do and does not have all the added features of csh and tcsh. However, this means a better level of security, so many scripts (especially as-standard system scripts) include the above line in order to make sure that security is maximised.

Also, by starting a new shell to run the script in, one ensures that the commands are always performed in the same way, ie. a script without the above line may work slightly differently when executed from within different shells (csh, tcsh, etc.), perhaps because of any aliases present in the current shell environment, or a customised path definition, etc.