CS240 Operating Systems, Communications and Concurrency – Dermot Kelly
Practical 2 Command Shell and Shell Scripts

Last week's practical introduced the Unix environment, how to login, how to open a command window and enter some basic commands, how to open an editor to create text files or programs and how to navigate the file system by creating and moving files between directories. We experimented with altering your current directory as you move up and down the file name space and navigating from the root to your CS240 directory.

There are two ways to interact with the operating system, either by using a windows oriented graphical user interface(GUI) or by using a command interpreter program called a shell. When navigating the file system, organising files and launching applications it is probably easier to use the mouse and GUI. The purpose of the GUI is to make the system easy to use, but the kinds of things you can do with it are limited to what appears in its menubars and drop down boxes.

When you interact with the operating system through the Command Shell, although typing text based commands is more cumbersome, it may be more powerful because you can tailor the commands, extend them and compose them with others to do custom tasks as well as being more in control of where the input and output for those tasks is sourced and directed.

The shell is a command interpreter program which has its own mini-language that it understands allowing the user to execute commands, and redirect input and output to those commands and link commands together to create more complex functions. There are a few alternate shell programs, which may use different command syntax, but we are going to use a shell program called the "bash" shell which uses the "$" prompt character where user input is expected.

We will look at how we can put a series of commands for the shell into an executable file known as a shell script, so that we can execute a complex series of commands simply by executing that single script file under the control of the shell.

The shell is an interactive user interface program which interprets your commands according to a specific syntax and communicates with the operating system to create processes to execute them.

Remember when using the Unix system the online manual is available to explain the details of any unix commands and the parameters taken by them.

man <command>

You can exit the online manual by typing 'q' at any time.

Write out the commands you use as you are going along and be ready to hand them up at the end if required.

The Shell

The shell program reads from a single input stream and writes to a single output stream (by default the input stream is the keyboard and the output stream is the xterminal window associated with the shell). If you have an xterminal window open and a $ prompt displayed then a bash shell is running. We are going to create a new process using the shell as follows:-

Enter the command **bash** at the $ command prompt.

Your original bash shell process will now start a new bash shell process which will wait (with the $ prompt) to accept your commands.

All your keyboard commands will now be read and executed within the environment of the new bash shell.

Try "**ps -l**" (the –l or "long" flag given to the process listing command gives more detailed information about running processes)

You should have three processes active, one for the original bash shell, one as a result of the **bash** command you just entered and one as a result of the **ps -l** command which is itself a process at the time when the process list is obtained.

Note that to execute a command, the shell 'forks' a separate child process, to execute that command, and then waits until the child process terminates. When the command is completed, the child process executing it exits and the shell process resumes by redisplaying the "$" prompt.

Look at the three process Ids (PID field) from the output of the **ps** command. The PID corresponding to the **ps** command is listed, but this process only existed while the **ps** command was executing. Once the list of current processes has been produced, this process no longer exists.

The other processes corresponding to the two running **bash** shells still exist however. To terminate either process enter "**kill -9 pid**", where pid is the process id of one of the **bash** processes. Choose the higher process id first or you will kill both. The **kill** command sends a signal to the process executing that **bash** command (signal 9 indicates to a receiving process that it should terminate).

Notice that if you chose the higher process id to kill, that control should return back to your original shell process now that the child shell has died. "Try **ps –l"** again. Now there should only be two processes listed.

You can also terminate a shell process by entering CTRL-D (i.e. press CTRL key and d together). This indicates an end of input condition on the input stream and so the command interpreter of the shell cannot continue. Try entering CTRL-D now. What happened? You can open a new shell terminal window under the applications/accessories menu using the GUI.

Shell Scripts
A Unix shell script is simply a list of one or more Unix shell commands contained in a file which may be executed by a shell program. This saves us having to retype complex commands which we may create for routine tasks. A script can be used to customise an existing command or to create new more complex and more powerful commands by combining existing ones, using communication pipes, as needed.

Make **cs240** your current directory.
Put the command "**echo hello world**" into a file called script1 using an editor preferably, or using the shell echo command and output redirection symbol below:-

```
echo 'echo hello world'  > script1
```

The **echo** command usually writes the string parameter to the current output stream (i.e. the xterminal window). In this case however, we are telling the shell to redirect the output to go to a file called script1. So a new file called script1 will now exist with the string **'echo hello world'** in it.

Open the script1 file with **gedit, emacs** or type **cat script1** to display the contents of this file in the shell's window.

Note that the file can be used as the basis of a shell script because it contains a valid utility command (**echo**) and a parameter to be echoed (**hello world**).

What happens when you enter the command **script1** at the shell prompt $? (make a screenshot 1)

You may get a command not found error because your current directory is not in the command search path of the shell. $PATH is the name of the shell's environment variable where the search path is configured. Everytime you type a command at the shell, it searches the directories listed in $PATH in the order in which they are listed. If your current directory is not listed, the shell will not find the filenames in it. You can review the current path settings with the command:-

```
echo $PATH
```

If you get a command not found error then you need to specify more explicitly the name of the file you wish to run by supplying a full path name or a more qualified relative path name.

Type **./script1** at the prompt.

This tells the shell that you want to run the file **script1** located in the current directory.

So the shell now should have found your file but noticed that you do not have permission to execute the file **script1**. (make a screenshot 2)

File Access Permissions
In Unix a file can have 3 types of access namely read, write and execute (rwx).

Script1 currently has r and w access only for its owner.

Try "**ls –l"** to verify this. This lists the access permissions and other "long" information for every file in the current directory.

Give yourself execute permission on the file **script1** by typing
chmod   +x   script1

This changes the mode(chmod) in which you and your processes can access the file.

Now that you have execute permission, run it in the current shell by typing
./script1.

The addition of further lines will not cause the execute permission of the file to change.

Add the **date** command to the file **script1** by typing **echo 'date' >> script1**.
The >> symbol indicates to the shell that output of the echo command is to be **appended** to file script1.

Type **cat script1** to see the contents of it.
Now run **script1** as described above and note the change in output.

Background Processes
The ampersand character & can be used to tell the shell to run commands concurrently with the shell program (i.e. not to wait until child processes terminate before accepting new commands). The commands execute in the background and status information about background processes (e.g. initialisation and termination) is passed to the parent shell and displayed on its current output stream. This allows us to have a number of programs running at once, and we still have the facility to enter Unix commands at the original shell prompt while they are running.

Enter the following:-

xterm &	a new terminal window appears
gedit &	a new editor window appears

Note that the shell prompt remains for us to continue launching other processes in our current window.

GREP
We are going to use the Unix grep utility to create a more complex shell script. The grep program searches for text patterns in the named files or in the standard input if no files are mentioned. The text patterns to match are indicated by the regular expression argument. Without options, grep prints each line in the input stream that contains a text pattern that matches the given expression. The options can alter this behaviour.

Make **cs240** your current directory. The output of the second command below will depend on whether you correctly followed the instructions of last week's practical.

What is the output of **grep 'quick' /usr/share/dict/words**
What is the output of **grep 'file' text/***

The simple regular expression appears in quotes as the first parameter and the file(s) to be searched for that pattern are specified in the second parameter.

Some elements which may be included in the regular expression are defined to have the following meaning: (They are just included here for later reference.)

^	beginning of a line
$	end of a line
.	match any single character
[…]	match any one of the characters in …; ranges like a-z are legal
[^…]	match any one of the characters not in …
r*	zero or more occurrences of expression r

Some valid flag options to the **grep** command are listed below. Use the **man** command for entire list.

-c	Produce a count of lines rather than the lines themselves.
-l	List the names of the files that contain text patterns matching the expression.
-n	Print the line number (from the input file) for each match.
-v	Print lines that don't contain text matching the expression rather than lines that do.
-i	Ignore case distinction.

I/O REDIRECTION
One of the main features of the Unix environment is the ease with which input/output to and from commands/processes may be redirected. The output from any command executed on the command line can simply be redirected to a file by placing > **filename** after the command:-

```
grep  -i  '^q.*'  /usr/share/dict/words  >  temp
```

The regular expression above configures grep to output all lines from the **words** file that begin with the letter q followed by any number of occurrences of any other letters. The –i flag tells grep to ignore upper/lowercase distictions, and finally, the output of grep is to be redirected into a file called temp.

cat temp Display contents of **temp** to standard output.

Controlling Output
The **pr** utility is used to paginate or columnate files for printing.

pr -4 temp Print the lines of file temp in four columns to standard output

more is a utility that reads from a single input stream (e.g a file) and writes to a single output stream (e.g. the shell window). It only writes one screen at a time and then waits for the space bar to be pressed before displaying the next screen. Press ‘q’ to exit more at any time.

cat temp
Notice that the file runs off the window.

Try the command
more temp

Pressing the space bar displays the rest of the file.

Connecting processes using Pipes
A pipe is a one way communication channel between two processes. The shell pipe symbol '|' indicates that the output stream of one command is to be used as an input stream to another command.

Try the command and make a screenshot 3

pr -4 temp | more We can now read temp easily in four columns.

Notice the blank lines at the end of the output which were generated by **pr** during processing.

EXERCISE - How could these blank lines be eliminated from the output of pr?

Given that the regular expression for a blank line is '^$', you could pipe the output from **pr** into **grep** to eliminate the blank lines and then pipe the resulting output into **more**. Fill in xxxxx in the following command:-

pr -4 temp | grep xxxxx | more

EXERCISE - Derive a command (using grep) to search the user word dictionary (/usr/share/dict/words) for all words containing the string 'hell'. Test your command and then place the command (not the output) in a file called script2.

Then execute the command(and make a screenshot 4) **sh script2 > scriptout &**

Note that the process id of the new background process appears in the terminal window.

When a done message appears for the background process type **cat scriptout** to see the results. If a done message doesn’t appear after a few seconds press <return> to flush the xterminal output.

Arguments to Shell Scripts

We can pass arguments to our shell scripts by using the shell variables $1, $2, …, $9 within our script. These variables are replaced by string arguments given to a shell script on the command line when it is invoked.

For example, create a file called **script3** containing the text **echo $1**

Make **script3** executable
chmod +x script3

Run it by typing **./script3 hello**

Now type **./script3 there**

Make a screenshot 5

Explain what you observe?

Script Programming Problems

The **du** utility (disk usage) gives the number of kilobytes contained in all files in the specified path and operates recursively on directories contained within that path. See the online manual for details. It will provide a line of output containing the file name and other information for each file it comes across in the given search path.

Try **du -a ***

The parameter * indicates all files in the current directory and is the default. What does the -a flag do (look in the manual)?

Use the **du** and **grep** utilities, pipes and shell variables to write a script called **filefind** which will search for a given file (supplied as a parameter and accessible as $1 within the script) starting at the root directory. Use **du** to traverse the filesystem from the root and then pipe its input into **grep** to search for a given filename and only output lines that contain that filename.

Make **filefind** an executable script. So now you have made a new command by composing simpler commands together. This was the objective of the practical, to show you the power and flexibility of Unix I/O and to show how the command interpreter interface can sometimes be more powerful than a GUI.

Use **filefind** to search for a file of choice (e.g. whose filename contains **'words'**) in the background and redirect output to a file called searchresults. While this is executing try searching for a different file of choice in the foreground. Notice that some information is denied to you when performing the searches from the root directory as some directories are protected. These system error messages will be displayed on the shell's standard error stream.

Write a script called **xword** which scans the dictionary file for a particular word outline (as in hangman style) and returns all matches of words of equal length to the search string and in the number of columns specified, without any blank lines. This is a useful script for solving crosswords.

Example: **xword b..m.. 5** should give

```
Today's Date and Time          Page 1
bowman
bowmen
```

NB. There are not enough words to occupy more than part of one column of one page so the other four columns are empty.

Try **xword b….. 5**
The words should appear in five columns occupying most of one page.

Summary of things you should be able to understand and do at the end of this lab:-

Know the advantage of a command line interface over a GUI

See what processes are running and how to control them

Understand how to run processes concurrently with the shell using &

Understand how the shell finds commands in the filesystem using $PATH

Understand the purpose of the grep, pr, more and cat command/utilities

Know how to create a shell script and make it executable and how to pass arguments to the script

Know how to redirect process input and output streams and how to create applications by composition of processes, i.e. joining processes with pipes

Be able to apply the above to write the shell scripts which solve the two problems at the end

Upload 5 screenshots and 3 **script** on the Moodle Lab2 submission.