WEEK 1.3 INTRODUCTION TO UNIX

Unix is a key operating system in the history of communications. Here, you will gain experience in using Unix both in command line and with a graphical interface. Most importantly, you will learn programming in shell script.

You have one 2-hour supervised laboratory to work towards the tasks below. You can continue in your own time, as long as you finish before next week's laboratory.

Form **groups of four**

TASK 1: ACCESS UNIX USING PUTTY (SSH)

i. In windows, from the programs menu select > Host Access > Putty (SSH) > ssh student.cms.gre.ac.uk

ii. When prompted, enter your user id and then your password.

iii. To keep a record of what you are doing, type **script task1** and hit **enter**. This will create a file named **task1** where the text on the screen will be saved. You will need this

iv. Examine the effect of the following commands
cal 3000
cal 6 3000
cal 9 1752

v. Add a redirection by typing
cal 6 3000 > june3000.txt
This sends the results of a command to a file instead of the screen. To check that the file was created, type **ls** which lists all files.

vi. Make a copy of the **june3000.txt** file using the **cp** command
cp june3000.txt june3k.txt
Type **ls** again to see the result.

You may think that the password prompt "doesn't work" because you don't see anything on the screen. That's normal. Just type the whole of it and hit enter.

*The filename **task1** is just an example. Use any name you find convenient.*

Why is the result strange for the third command?

*You won't see anything on the screen, as the results of the command you typed have been sent to the **june3000.txt** file you created instead of the screen.*
Your Unix home can be accessed from Windows on U:

It's 'l' for lima, not "I" for India

vii. Type the following command to show the same list of files but with more detail:

ls -l

See the example below:

After "-", it's 'l' for lima, not '1' the number.

Permission Flags
The first letter specifies a directory (d) or not (-). The rest are three groups of three letters stating the read (r), write (w) and execute (x) permissions for the owner, the group and everybody else, if some permission is denied, then a dash "-" is used instead.

```
-rw-------. 1 root root 23918 Nov  8 15:18 .bash_history
-rw-r--r--. 1 root root    18 May 20  2009 .bash_logout
-rw-r--r--. 1 root root   264 Aug 19 06:06 .bash_profile
-rw-r--r--. 1 root root  1333 Sep 12 04:18 .bashrc
-r--------. 1 root root    59 Oct 27 06:28 .cloudfuse
drwx------. 3 root root  4096 Oct 29 05:57 .config
-rw-r--r--. 1 root root   100 Sep 22  2004 .cshrc
drwx------. 2 root root  4096 Jul  2 06:49 .elinks
```

Directory? | owner | group | everybody = **Permissions** | **Directories** | **Group** | **Size** (in Bytes) | **Date** and **time** | Name of **File** or **Directory**

Use **cd** to change to a different directory. Try this:

cd /home/sp02/unixfiles

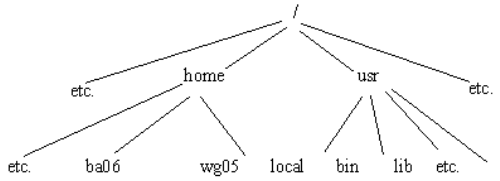
And show the contents of this new directory with

ls -l

Return to your home directory with

cd

UNIX uses a hierarchical file structure. Files, directories and even I/O devices are treated uniformly. There is no common conception of a 'drive' as in windows. All UNIX files 'hang' from one root, called /.



Typical directories might be:
/home/ga606/ccs
/usr/local/work

viii. Make a directory called ccs and change to it

mkdir ccs
cd ccs

Now, give the command to copy all the files from the directory /home/sp02/unixfiles/ to your own area. For 'all files', use the wildcard character ***** as in:

cp /home/sp02/unixfiles/*.* .

Note that this is 'space' and then 'dot'. The destination dot (.) denotes 'the current directory'. Alternatively, you could have stayed in your home directory, and instead given the command **cp /home/sp02/unixfiles/*.* ccs**

Use **ls -l** to find the sizes and owner(s) of these files.

ix. Display the file **poem.txt** using **cat poem.txt** and then using **more poem.txt**.
To redirect the output to a file (rather than the screen), try **cat poem.txt > poem2.txt**. Use **ls -l** to see if the new file poem2.txt was created.

Now try a slightly different redirection:

cat poem.txt >> poem2.txt

What will happen? Use **ls -l** and then **more** to check.

***>>** works like **>** except that it appends data to the end of a file if it already exists.*

x. To delete a file you need to remove it using rm:

rm poem2.txt

You will be asked if you are sure. Type **y** and press **enter.**

xi. Stop recording the script file by typing

exit

Copy the script file to your Unix area on the PC network and insert the script file contents into a word document. Name the document 'Introduction to Unix'

xii. Logout by typing **logout** and hitting enter.

Step (xi) will work only if you have done (iii) correctly.

Your Unix area is mapped to drive U: in your PC account. You can copy the content of any files you created here into your lab document. If you do not have a U drive, you can create it running the map-stu-udrive.bat file on T:

TASK 2: ACCESS UNIX USING A GRAPHICAL INTERFACE (XMING)

Xming is a program which runs on a Microsoft Windows PC to communicate with Unix/Linux servers and provides a graphical user interface for using these servers.

Are you ON THE GREENWICH CAMPUS?

(ON CAMPUS STUDENTS ONLY)

i. From the programs menu select > Host Access > X Windows > CMS Student Solaris Server and login to UNIX. This will then launch the Sun Java Desktop (based on Gnome - a desktop for Linux or UNIX platforms).

ii. From the Launch button (bottom left corner) select Applications > Accessories > Text Editor. The editor is reasonably intuitive for opening and saving files. Change all occurrences of 'Shalott' in poem.txt to Onion (poem.txt is in the **ccs** directory)
Change all abbreviations such as **'d** into **ed**
Save the edited files under different names.

iii. On the desktop, find the icon 'This Computer 'and double click it. In the Location bar, enter /home/your_userid (e.g. /home/aa001). This will display all of your files in your Unix home area.

iv. Right click on the desktop and select Open Terminal. In this terminal window type **firefox** (all lower case) and browse the web.

v. Copy the new version of the poem and web page you visited and append to the document 'Introduction to Unix'.

(FOR STUDENTS STUDYING OFF CAMPUS)

i. Download and install Xming, Start Xlaunch, choose 'Multiple windows' then Next. Select 'Start a program' then Next, In the Run Remote select 'Using Putty'. Now enter student.cms.gre.ac.uk as the 'Connect to computer', your userid and password then Next. Continue by clicking next you can save the configuration and click finish. This will launch Xterm on a local Xserver.

ii. In the xterm, type **gedit &**. The editor is reasonably intuitive for opening and saving files. Change all occurrences of 'Shalott' in poem.txt to Onion.
Change all abbreviations such as **'d** into **ed**
Save the edited files under different names.

iii. In the xterm, type **nautilus &**. Then in the Location bar enter /home/your_userid e.g. /home/aa001. This will display all of your files in your Unix home area.

iv. In the xterm, type **firefox &** and browse the web.

v. Copy a webpage you surfed to the word document 'Introduction to Unix'.

In UNIX, one can run files that contain executable code (usually the result of compiling a program from a high-level language such as C into machine code) or files that are called *shell scripts* (which contain a sequence of commands in the Shell programming language).

TASK 3: RUNNING C PROGRAMS IN UNIX

*You can use **gcc** to compile C programs in Unix or Linux:*

gcc filename.c -o filename.out

i. Start a script file: **script task3**

ii. From the ccs directory compile the C program hello.c using **gcc hello.c -o hello.out** and run it by typing **./hello.out** You should see the familiar *'Hello World'* displayed.

iii. Use the text editor to change the Hello world program so it will now display *'Hello Your Name'* instead

iv. Compile the C program **oddeven.c** and run **oddeven.out**

v. Append the script file to the document 'Introduction to Unix'.

gcc stands for GNU C Compiler Further details on using gcc and other C compilers: http://unix.cms.gre.ac.uk/software/compilers.html

The "dot-slash" ./ in front specifies that the file to be executed is in the current directory

TASK 4: RUNNING SHELL SCRIPTS IN UNIX

Before working on this task, you need to read pages 6-7 and replicate the examples of pages 8-11, so as to learn how to create and run Korn Shell scripts.

Use the text editor **gedit** to create the following Korn Shell Scripts:

i. Write a shell script to prompt the user for a variable, test the value of the variable, if it = 1 then write to the screen, "The value was 1", else write to the screen "The value was not 1".

ii. Write a shell script program that accepts three parameters and displays their average to the user. *Note: you will probably have to do two calculations, one for the total and one for the division.*

iii. Write a shell script program that accepts a variable number of parameters and displays their average, and the number of numbers input, to the user.

iv. Write a shell script program that accepts a file name from the user, displays the file to you using the less command, then calls the editor (command is gedit) to allow you to edit that file, once you quit the text editor, your shell script will continue to run. It should then display the file using the less command again. Make some changes in the text editor so you can see that the file has changed. Do not forget to test if the file exists before you attempt to do anything with it. If it does not exist then output an appropriate error message.

v. Write a shell script program to provide the following facilities to a user in the form of a menu:

1. Allow the user to use an editor e.g. gedit (which can be called from the command line using gedit filename) to write a poem.
2. Run a word count on your poem using wc
3. Display the output to the screen and display a page at a time using more or less (ask the user which one they want to use).
4. Sort the output using the UNIX sort command and put the results into a file, then display the output of the sorted file using cat or more. (ask the user which one they want to use)
5. Exit

Append the programs and the script files demonstrating the operation of the programs to the document 'Introduction to Unix'.

TASK 5 (OPTIONAL): Install Linux on your home computer

Linux is a UNIX type Open Source computer operating system designed primarily for the PC but also available for a wide range of other systems.

To install Linux on you home machine, follow the instructions at
http://staffweb.cms.gre.ac.uk/~wd18/linux_install/

TASK 6: REFLECTION (as a group, not as individual members)

Identify the sections of the laboratory you have understood and demonstrate your understanding - beyond the simple level of completing the laboratory - through cognitive processes such as analysing, explaining, interpreting, and evaluating. Illustrate, by the use of examples how the laboratory contributed towards your understanding and your Degree programme.

For the sections of the laboratory in which you struggled with, or were uncertain of, identify why this was the case. Evaluate the effectiveness of your learning strategy, including factors such as, motivation, preparation, commitment, time management, communication, constraints and support. With reflection to past experience, identify how you could improve your learning and performance to overcome the barriers encountered in this laboratory such that they do not infringe upon the next laboratory you undertake.

With relation to the sections of the laboratory you encountered difficulty with, state how, and by when you intend to gain competence in these areas.

Critically appraise the laboratory; identify sections you thought were positive, facilitated your understanding and contributed to your Degree programme; identify sections that require improvement and state how and why would you change the laboratory to improve the laboratory for the next year's students.

Attach the marking scheme on the front of your document, print it and hand it in at the laboratory next week. ONE printout to be handed in per team.
*In addition, **ALL** team members need to individually upload their documents into **week 1.3 (in PDF format)** by midnight the day before their next laboratory.*
Even for your individual uploads, make sure that the marking scheme has the names of all the members of your team.

UNIX Shell Programming

Up to now you have issued commands only using the keyboard. The shell is a command language that provides a user interface to UNIX. It executes commands typed in at the terminal or from a file. By providing the facility to read and execute a sequence of commands, you can create your own commands with the same status as system commands like **ls**. For example, suppose you create a text file called **setupusers** with the necessary UNIX commands to set up accounts for 100 users. If you then type **setupusers** this will be executed in the same way as any other Unix command like **ls**.

The shell is both a command language and a programming language that provides you with the ability to set up an environment tailored to you or a group's needs. It can link applications together that would normally have to be initiated by separate commands from the command line (like a batch file in DOS). For example, it could link a compiler with an editor, such that if there were errors from your compilation it could automatically throw you back into an editor.

As a programming language the shell provides you with facilities above those provided by the raw UNIX system such as programming constructs, variables, wildcards etc. A file that contains shell commands is called a shell script.

There are three popular shells: Bourne Shell (bash), Korn Shell (ksh) - superset of the Bourne Shell – and C Shell (csh). We shall concentrate on the Korn shell as the syntax is slightly easier.

To generate a shell script:
1. Create a text file containing your desired set of commands
2. Use **chmod 700 filename.ksh** to make it ‘executable’
3. Run it by simply typing **filename.ksh**

File Execute permissions

At the UNIX prompt, you will see the permission flags for each file when you type ls -l e.g. -rwxr--r-- The first letter specifies a directory or not, the remaining are three groups of three letters stating the read, write and execute permissions for the owner, the group and everybody else.

Changing the permission of a file requires three numbers: the first is the permissions for the owner, second for the group and third for everyone.

Suppose you wanted to give yourself read, write and execute access and the group and everyone read access only. You need to know that

Read = 4
Write = 2
Execute = 1

If you want to be able to read, write and execute, then you add them up. For example, you add 4 + 2 + 1 = 7 and assign yourself a value of 7. If you want, the group and everyone else to be able only to read, then you give them a 4 and a 4 respectively, eg:

chmod 744 sh1.ksh

Or, to give yourself and the group read and write access (4+2 =6), but none to anyone else (0):

chmod 660 sh1.ksh

To generate a shell script:
1. Create a text file containing your desired set of commands
2. Use **chmod 700 filename.ksh** to make it 'executable'
3. Run it by simply typing **filename.ksh**

KShell examples

Use the text editor gedit to create and run the following example Korn Shell Scripts. Don't forget to change the file permissions using the chmod 700 command. It is important that you use the UNIX editor gedit and do not import files into UNIX from a windows editor, as these files will often have control characters attached and will not function correctly.

Example 1: sh1.ksh

```
#! /bin/ksh
echo "Please enter your name: "
read name
echo "Your name is" $name
```

This tells the system that it is a Korn shell

Note that the echo command is a print statement. To print a blank line, just execute it without text, e.g:

echo
echo hello
echo

To run this example, create a file named sh1.ksh. Type this code into the file, save it and give it execute permission with

chmod 744 sh1.ksh

and then run it with

sh1.ksh

Example 2: sh2.ksh

```
#! /bin/ksh
echo parameter 1 = $1
echo parameter list = $*
```

This shell script expects command line parameters to be passed to it. The parameters are stored as $1, $2, $3 etc. where the number represents the order in which they were typed in. In the example below $1 = Hello, $2 = Peter, $3 = Smith and $* stands for a list containing all the input parameters. An example run of sh2.ksh is:

sh2.ksh Hello Peter Smith

parameter 1 = Hello
parameter list = Hello Peter Smith

KShell examples

Example 3: sh3.ksh

```
#! /bin/ksh
 for i in $*
 do
 echo parameter $i
 done
```

Example run:
sh3.ksh Hello Peter Smith
parameter Hello
parameter Peter
parameter Smith

The syntax of the for loop is **for ... in ... do ... done** where do … done indicate the boundaries of the loop.

Example 4: sh4.ksh

```
#! /bin/ksh
 count=1
 for i in $*
 do
 echo parameter $count is $i
 let count=count+1
 done
```

To initialise and create a variable it is just assigned a value e.g. **count=1** The let command forces evaluation of an assignment statement e.g. **let count=count+1** Note: There must be no spaces in the arithmetic expression i.e. either side of the equals or plus sign in the example below.

Example run:
sh3.ksh Hello Peter Smith
parameter 1 is Hello
parameter 2 is Peter
parameter 3 is Smith

Example 5: users.ksh

```
#! /bin/ksh
 echo There are $(who | wc -l) users on
the system
```

The file is simply used to execute a one line UNIX command (who | wc -l) that is awkward to type on a regular basis. **'who -'** provides a list of the users currently on the system and '**wc -l**' counts the number of lines (requested by -l) and hence the number of users (as who provides one per line). **$(** expression **)** tells the Korn shell interpreter to evaluate the expression within the brackets (before echoing it).

KShell examples

Example 6: menu.ksh

```
#! /bin/ksh
 stop=0
 # test int1 equals int2
 while test $stop = 0
 do
 echo
 echo
 echo Example Menu Program
 echo
 echo 1 : Print the date
 echo 2, 3 : Print working directory
 echo 4 : Directory listing
 echo 5 : Delete *.bak files
 echo 6 : exit
 echo
 echo 'Please enter your choice? '
 read reply
 case $reply in
 "1" ) date ;;
 "2" | "3" ) pwd ;;
 "4" ) ls ;;
 "5" ) if test -f *.bak
 then
 rm *.bak
 echo Files deleted
 else echo No *.bak files found
 fi ;;
 "6" ) stop=1 ;;
 * ) echo illegal choice ;;
 esac
 done
```

while evaluates the condition and if it is true, it executes the statements between the **do** and **done**. It does this repeatedly until the while condition becomes false.

The condition test $stop -eq 0 tests the value of the variable stop and if it is equal to 0, then the condition is true. If it is true then the while statement will enter the loop. Given that set stop = 0 is before the while loop, the condition will always succeed the first time and hence the loop will always be entered.

read accepts input from the keyboard and stores the value in a variable e.g. **read reply**. To refer to a variable in a program it is referenced with the $ sign e.g. **echo $reply**

The **case** statement is like an **if** statement only shorter. What it says is: test the value of the variable called reply against the following values and carry out the appropriate actions on the right side of the) character.

This says:

If there are any files called something.bak then delete them and tell the user that the files are deleted else tell the user that there are no .bak files to be found

*Note: The **if**, **then**, **else** and **fi** must be on separate lines.*

KShell examples

The test command:

There are many parameters available for use with the **test** command, e.g.:
-f True if filename exists as a non-directory
-d True is filename exists as a directory
-r True is filename exists as a readable file
-n True if string contains at least one character
-s True if filename contains at least one character
-w True if filename exists as a writeable file
-x True if filename exists as an executable file
-z True if string contains no characters
str1 = str2 True if string 1 = string 2
string True if string is not null
int1 -eq int2 True if int1 equals int2
int1 -ne int2 True is int1 does not equal int2
int1 -gt int2 True if int1 is greater than int2
int1 -ge int2 True if int1 is greater than or equal to int2
int1 -lt int2 True if int1 is less than int2
int1 -le int2 True if int1 is less than or equal to int2

Alternative syntax for conditional expressions used in e.g. whiles and ifs:

while [$stop = 0]
if [$stop = 0]
while test $stop -eq 0
if test $stop -eq 0

Testing for strings (this is very counterintuitive, you quote the variable in the test, not the string to be tested):

If test $reply=m

How to Access Unix:

From the University Network:
- From the programs menu select > Host Access > Putty (SSH) > ssh student.cms.gre.ac.uk. Enter yes to the security alert ' The server host key is not known'. At the login: prompt, enter your user id. At the password: prompt, enter your password.xqa
-- Via Xming (This is a GUI interface to UNIX via the PC's). From the programs menu select > Host Access > X Windows > CMS Student Solaris Server and login to UNIX.
-- or Host Access -> X Windows -> CMS Stulinux Server running Ubuntu, which is also accessible via "Host Access -> Putty (SSH) ssh stulinux.cms.gre.ac.uk".

From your home machine:
- Download Putty (SSH) from http://unix.cms.gre.ac.uk/software/mswindows.html#putty and login as from the University Network.
- Download the public domain release version of Xming - PC X Server from http://sourceforge.net/projects/xming/files/Xming/6.9.0.31/Xming-6-9-0-31-setup.exe/download.
- To use virtual laboratory desktop follow the instructions at: http://labs.cms.gre.ac.uk/labdesktop/external.asp

More information and help is available on the School's Unix systems at http://unix.cms.gre.ac.uk

History of Unix:

Unix was developed in the early 1970s and has become one of the main operating systems used in industry, available on almost any machine. The vogue in the 1970's was for complex multi-user systems on medium to large mainframe computers. At the same time, cheap minicomputers were being installed in increasing numbers in research laboratories. These could only be programmed in native assembly code, and the operating systems were primitive. Thompson and Ritchie soon spotted the advantages to be had from writing an operating system in a high-level language rather than assembly. Thus, the development of the UNIX operating system, and the C programming language, went hand-in-hand. The use of a high-level language for almost all of the operating system code (a small amount, perhaps 5%, had to remain in the native machine code) meant that UNIX could be put up on any machine with a translator for the C language. Unlike DOS and Windows which run only on Intel processors, UNIX can run on any machine.
The severe limitations on storage of the early UNIX machines forced the developers to think long and hard about UNIX. As a consequence, UNIX is extremely well-designed and well-engineered: its components fit together well, and its powerful shell programming environment permits complex manipulations to be built up from a tool-box of simple components. It consists of a small kernel of essential utilities for maintaining a file system and for running programs in multi-tasking mode. This kernel interfaces directly to the hardware. Surrounding the kernel, between it and the user, is the shell. The shell is the command language that provides the user interface to UNIX. The shell provides a fixed set of commands and will execute commands typed in from the keyboard or from a file. This is similar to a batch file in DOS. Files containing commands (called Shell Scripts) may be created allowing users to define their own, more sophisticated commands. UNIX is a stable operating system which retains a fierce loyalty amongst users, and is extremely flexible and powerful.