COMP2017 & COMP9017 Week 1 Tutorial

Introduction to Linux

Introduction to Linux

What is Linux

Linux is a family of operating systems that derive from the original AT&T Unix developed in the 1970's at Bell Labs. As an operating system it manages the computer's hardware and software and provides a common interface for applications to use as they manipulate elements of the system.

Unix follows a philosophy that revolves around designing small, well defined programs that integrate well together. This can be summarised as follows:

- Write programs that do one thing and do it well
- Write programs to work together
- Write programs to handle text streams, because that is the universal interface

Everything in Unix is either a file or a process. A file is any collection of data, such as a series of configuration strings, an image, or compiled source code, while a process is a program that is currently being executed. Unix then revolves mostly around passing and parsing streams of data between different processes and files.

The Kernel and the Shell

The Unix kernel is a master control program that handles low level tasks such as starting and stopping programs, managing the file system and peripheral devices and passing instructions to processors or graphics cards. The kernel is complimented by a series of small programs that perform common tasks such as listing the files in a directory, changing directories and executing compiled code. Other tools can then build off these functions to provide search functions, document readers and image processors such as desktop environments.

Terminal Basics

The terminal is the main way of interacting with the Unix environment as opposed to the graphical user interface GUI in operating systems like Windows. At first, the terminal may seem unusual and inconvenient but you will soon learn that it is much more powerful and offers greater flexibility.

Keyboard shortcuts

Here are several keyboard shortcuts that are helpful to know when using the terminal

- ctrl + c terminates the current running program
- ctrl + d sends end of file EOF to the running program
- ctrl + a moves the cursor to the beginning of the prompt
- ctrl + e moves the cursor to the end of the prompt

Wildcards and tab completion

Wildcards and tab completion allow you to reduce the amount of keyboard input for commands. The wildcard `*` is a special character interpreted by the shell that is replaced with the names of matching files as if the user had typed the filenames explicitly. For example, suppose we are in a directory containing the files *a.txt, b.txt* and *c.txt*. The following three commands are equivalent:

```
$ cat a.txt b.txt c.txt
$ cat *.txt
$ cat *
```

When typing out a command, your shell also provides a tab completion feature where commands and filenames are guessed from the first few letters that you have typed. For example, at the prompt type "ec" and then press the tab key, your shell will guess that you want to type "echo" and complete the name of the command for you.

In addition, you can use tab completion to complete the path of a particular file or folder you want to reference. Often there are times when a file is hidden deep within many folders away from you. Instead of typing out each intermediate directory, which you may not know the names of anyway, your shell can assist you by expanding the path to the file with the closest matching filename, stopping mid path if there is more than one possible match. Simply type a few more letters, enough to resolve any ambiguity and then press the tab key again to continue completing the path.

Getting help

One of the most useful linux commands is **man**. This brings up the **man**ual pages for a given command and shows options and example usages if you are unsure of the syntax. To **q**uit the man page, press **q**. As an example, enter the command **man ls** to see the manual pages for the **ls**t command. This command shows the names of all the files and folders in the current directory. You should see in the man page the option **ls -a** which shows all files.

Quit out of the man page and try both **ls** and **ls -a**. You should notice that the second command lists quite a few more files than the first, all of them prefixed with a full stop.

mkdir is the **m**a**k**e **dir**ectory command. As the command suggests it creates a new directory with whatever name you specify. For example **mkdir comp2017** creates a new directory in the current folder called 'comp2017'. Directories can be removed with the **rmdir** command.

Next we want to be able to navigate between directories. For this we have the **c**hange **d**irectory command **cd**. To move to the comp2017 directory you would simply **cd comp2017**.

Tab Completion

Typing out the full names of each folder every time is somewhat redundant and to work around this Linux features tab completion. Pressing tab will automatically complete your current command or path if it can be uniquely determined. If it cannot be uniquely determined, pressing tab twice will list all possible completions.

From the comp2017 directory type **cd ../D** and then press tab twice. You should see the following output.

```
Desktop/ Documents/ Downloads/
```

Changing this to **cd ../Doc** and then pressing tab once should complete the command to **cd ../Documents**. Once you've entered this command use tab completion to navigate back to your comp2017 directory.

If your screen is looking full you can use the **clear** command or *Ctrl + L* to clear the screen.

Question 1: Terminal Navigation

Using **cd ..** navigate upwards from your current directory to the topmost or 'root' directory, pay attention to the path that you took and then try to navigate back to your home directory. If you get lost **cd ~** will automatically take you back.

```
$ cd /        go to the root directory
$ cd ~        go to the home directory
$ cd .        go to the current directory (loop backs)
$ cd ..        go to the parent directory
```

Files

As stated above, in Unix everything is either a file or a process. Files can be read, written to and executed. To see what permissions a file has use the **-l** option for the **ls** command.

```
-rw-r--r-- 1 root root   387 Dec 10 10:17 Makefile
```

Here this Makefile has read and write permissions for the user, and read permissions for group and anyone. File permissions can be changed using the **chmod** command.

There are a number of commands that can be used to read and write to files, the most basic of which are **cat** and **touch**. The cat command con**cat**enates files and prints the output, allowing the easy reading of plaintext files while touch changes the timestamps of a file, if the file doesn't exist then it creates it. **less** also reads text files, but places the user in an environment that is virtually identical to the one for **man**. It can be escaped using *q* in the same manner.

Existing files can be copied using **cp** moved **mv**, and removed **rm**. Directories are removed using **rmdir**. Though there are a few flags that can be passed to the rm command to grant it the same utility. Hidden files are prefixed with a . and can be viewed using the -a flag with the ls command. These hidden files are also called 'dotfiles' and tend to be configuration files for various programs.

To add text to a file we can use the **echo** command. Echo prints whatever input it is given, for example

```
echo "Hello World!"
```

We can redirect the output of any Linux command to any other Linux command, or to a file.

As per the Linux philosophy, text can be directed between processes and to files. In this case the **>** and **»** commands write and append to files respectively.

```
echo "Hello World!" > hello.txt
```

As all inputs and outputs from processes are text based, the output from one command can be 'piped' as the input to another command.

```
cat my_file.txt | less
```

The above command takes the output of cat and displays it using the 'less' command, which works in a similar manner to the **man** command you saw earlier.

Question 2: Global Regular Expression Print

You may notice that the thing you are searching for using grep is displayed in red. Have a look in */etc/profile.d/colorgrep.sh*.

- What is really happening when you type the grep command?
- Create or edit your *.bashrc* file to create a new alias similar to the ones you saw in colorgrep.sh for a grep_no_colour command. The "–color=NONE" option will be helpful here.
- Run **source .bashrc** to run the commands in your .bashrc profile and try your new aliased command, have you de-coloured grep?

The UNIX Filesystem

In the root directory you can find a number of different folders with specific functions.

- *home* Contains the home directory for each user. Your home directory contains your Documents, Downloads and other directories. When logged in the system variable $HOME is associated with your home directory. Try **cd $HOME** to see it being used.
- *bin* contains binary executable files that execute most of the processes on the system. Having a look in here you should see a number of core unix commands such as ls, touch, cat and su.
- *boot* is often a separate boot partition rather than just a directory, you can check this using **lsblk**. This partition manages how the kernel is loaded when the computer boots. You should not modify anything in this folder unless you are sure that you know what you are doing, a mistake can render your operating system non-functional.
- *lib* contains common shared libraries and kernel modules, such as firmware and cryptographic functions.
- *dev* is the device folder. This contains files that communicate between the operating system and any connected devices. If you plug a usb into your Linux system you will find and be able to mount it from here.
- *root* The home folder for the root user, otherwise identical to a regular user's home folder.

Creating files and folders

The **touch** command allows you to create files or update file timestamps if the file exists.

```
$ touch document    creates a file in the current directory
$ touch notes.txt   creates a file with the .txt extension
$ touch ~/numbers   creates a file in your home directory
```

The **mkdir** command allows you to create folders.

```
$ mkdir folder             creates a folder in the current directory
$ mkdir -p ~/photos/2017   creates intermediate folders as required
```

Manipulating files and folders

The **cp** command allows you to make a copy of one or more files or folders.

```
$ cp /usr/share/dict/words ~/            copy file to home directory
$ cp /usr/share/dict/words ~/documents   copy file to documents folder
$ cp <file1> <file2> ... <destination>   copy group of files to destination
$ cp -r <folder> <destination>           copy folder and its contents
```

The **mv** command allows you to rename a file or folder or move it into another folder.

```
$ mv DS1234.jpg photo.jpg   renames the file to photo.jpg
$ mv photo.jpg ~/pictures   moves the file into the pictures folder
```

The **rm** command is one of the most dangerous commands as it removes the given files and folders, and instead of placing these folders in a Recycle Bin it removes them permanently.

```
$ rm photo.jpg        deletes photo.jpg
$ rm one two three    deletes files one, two and three
$ rm -rf ~/pictures   deletes the folder and all of its contents
```

Working with text and files

The **echo** command displays the arguments given to it, including any environment variables.

```
$ echo Hello there   outputs "Hello there"
$ echo $PATH         outputs folders to search through for executable files
```

The **printf** command displays the arguments given to it, and is capable of processing format strings.

```
$ printf "Hello there\n"      outputs "Hello there"
$ printf "Hello, %s" Alice    outputs "Hello, Alice"
```

The **cat** command outputs the contents of each file given to it.

```
$ cat /usr/share/dict/words         outputs a list of words
$ cat /proc/cpuinfo /proc/meminfo   outputs cpu and memory information
```

The **less** command is the opposite of more and is an advanced text viewer that allows you to navigate through the text using the following commands.

```
up arrow, down arrow   up or down, one line at a time
page up, page down     up or down, one page at a time
/<pattern>             find pattern in the file
n, N                   next or previous result
g, G                   start or end of file
h                      help
q                      quit
```

Pipes and redirects

Pipes and redirects allow to combine Unix commands and small programs to produce elegant functional solutions. Redirection uses the following symbols: $<$, $>$, $>>$, and | is used for piping.

$>$ will redirect the output of one program to a file, rather than the terminal. If no file with the specified name exists, a new file will be created. Otherwise, any content of the existing file will be overwritten by the output of the program. This is something you need to be careful about.

```
$ cat /usr/share/dict/words > ~/words
```

$>>$ is very similar to $>$, however if the specified file already exists then it will append the output of the program to that file without overwriting any existing data.

```
$ echo "# List of words" > ~/words
$ cat /usr/share/dict/words >> ~/words
```

$<$ will send the contents of a file as standard input to the program.

```
$ tr a-z A-Z < /usr/share/dict/words
```

You can combine input and output redirection, like so:

```
$ tr a-z A-Z < /usr/share/dict/words > ~/WORDS
```

| will send the output of one program to the input of another. Below, the output of the $grep$ command is sent as input to the wc command, that outputs the number of words starting with the letter a.

```
$ grep "^a" /usr/share/dict/words | wc -l
```

The pipe is powerful because it allows us to combine the functionality of multiple programs in order to solve a problem. Here is a more complicated one liner:

```
$ grep "^a" /usr/share/dict/words | sort -r | less
```

It works like this:

- Find all words starting with the letter a
- Sort these words in reverse alphabetical order
- Opens the result in the $less$ program for viewing.

Question 3: Terminal Exercises

- Output “Hello world” to the terminal
- Output your current working directory
- List the contents of your home directory including any hidden files
- Create the folder sun in your home directory
- Navigate into the sun folder you just created
- Create a file with the text “Modern Unix” in a file called unix.txt
- Append the text “ZFS + DTrace + Zones + KVM” into unix.txt
- Rename unix.txt to solaris.txt
- Copy solaris.txt to illumos.txt in the same directory
- List the contents of the sun directory
- Delete the solaris.txt and illumos.txt files
- Use the less command to view the /usr/share/dict/words file
  - Navigate through the file
  - Search the file for the word zythum
  - Jump to the end of the file.
  - Jump to the start of the file.
  - Quit less

Shell Scripts

A shell script is a file with execution permission that contains a series of bash commands. When executed it simply runs each line of the bash script as a command line command. This is a convenient method of re-using bash commands, however the question arises as to how the kernel knows to interpret this executable file as a bash script rather than, for example, a ruby script.

For this linux uses a string prefixed with a hash bang at the start of the file to indicate what command the file is to be run with. In the case of a shell script this is *#!/bin/bash*. The program loader then runs the file with the command located along the specified path, in this case /bin/bash is the binary file that runs the bash terminal.

Question 4: Shell Script Basics

Write a shell script that prints your username by looking at the path to your home directory. You may fine ~or $HOME and the **sed** command helpful here.

Bash

Bash is a Turing complete language in and of itself and comes with a number of the usual programmatic conveniences such as variables, switches and loops. It also has a number of different syntactic conventions and useful pre-defined constants such as $USER and $HOME. Try echoing those constants in the command line to see their current values.

Variables are also indicated by a $ prefix when they are called, but not when they are initialised. As spacing is a strict syntax structure in bash, spaces around variable initialisations cannot be ignored else an error will be thrown. As Linux is built around text streams, all variables can be considered to be strings.

```
foo="bar"
echo foo
echo $foo
```

The dollar sign or grave quotes can also be used to encapsulate another bash command and pass it to a variable. As the output of all commands is a text stream, there are no issues surrounding typing.

```
foo=$(ls)
echo $foo
bar=`ls`
echo $bar
```

Bash switches are bookended by if and fi with the general syntactic structure following ‘if [condition]; then else fi’. Double equals signs are not used, and the terms being compared should be encapsulated in double quotes. Boolean expressions for use in switch statements and loops are encapsulated in square brackets.

```
if [ "$foo" = "$bar" ];
then
        echo sesquipedalian
else
        echo loquacious
fi
```

Bash loops operate over collections in a manner quite similar to Python. They follow a syntactic structure of

```
 for file_var in `ls` ;
 do
         echo file: $file_var
 done
```

Bash also makes use of while and until loops: while [condition]; do done, and: until [condition]; do done. To emulate traditional loops over a range of integers the **seq** command can be used to provide a collection over some range to loop over.

```
for i in seq 1 10;
do
        echo \$i
done
```

Alternatively, the let command can be used to change the value of a variable

```
foo=0
while [  \$foo -lt 10 ];
do
        echo \$foo
        let foo=foo+2
done
```

Once again the lack of spacing when assigning a value to a variable is essential. Bash does support ++, += and related operators.

Bash scripts also accepts arguments, these are automatically set to variables $1 for the first argument, $2 for the second and so on.

Question 5: Shell Exercises

- Output a command line argument from your shell script
- Using a command line argument, output the contents of directory and sort contents in reverse.
- Extend your script to count the number of files that exist in your directory.
- As an extension, recursively output the contents of all files within the folders and count the total number of files contained within a folder.

Question 6: More Shell Exercises

- Create a shell program that will calculate the sum of numbers passed from command line.
- Write a shell program that will output the most frequent character in a given string and the length of the string
- Calculate the hash of a directory using the md5 command (md5sum)
- Use the inotifywait command to monitor for changes in your directory and trigger recompilation of your project (You can use an existing Java, C or Python project).