THE UNIVERSITY OF SYDNEY

INFO1910 S1 2023 Week 1 Tutorial

Introduction to Linux

Introduction to the Course

Ed
We will be using Ed this semester for discussions, answering questions, and some assessments. Check Ed at least once a day, feel free to try to answer questions posted there, it's a good test of whether you understand the content.

On the topic of asking questions, do this. When you're stuck, search things up, and when you're still stuck ask.

Canvas
You can find your lecture recordings here, in particular the Advanced lecture recordings. Your current marks will also be recorded on Canvas under the marks tab.

Git
We will be learning and using git extensively in this course. Git is a version control system, it tracks changes to collections of files and allows you to share your modifications between different computers. Git will be used to handle your assignment submissions.

Linux
You will **need** your own local installation of Linux, either on bare metal, dual booted or in a VM, OSX is a stopgap measure if you don't want to dual boot and your hardware is insufficient to run a VM, but will lead to problems in future subjects. WSL and Git Bash will not be sufficient. If you have difficulty doing this please ask sooner rather than later. For installation instructions, see the 'Week 0' tutorial sheet.

Text Editors
You will need to be familiar with a text editor (not an IDE). The recommendation is for a combination of vim or emacs, and sublime text or atom, but other options are out there. For this unit we will be trying to avoid any interface with a automatic 'run' button. Again see the 'Week 0' tutorial sheet for more details.

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.

Optional Challenge

If you already have some prior exposure to Linux, or the command line, consider the following 'hard mode' version of this tutorial. Upon reaching the login screen press **Ctrl + Alt + F2**, your screen should turn black and you should be shown a login prompt. You will not have GUI for the rest of the tutorial. If you're not running Linux, this won't work, but you can try to restrict yourself to the terminal anyway.

Terminal Basics

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.

Question 1: Dot Prefixing

- What does it mean when a file is prefixed with a dot? (The manual pages for `ls` might shed some clues)
- Is this true on other operating systems?

`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 info1910` creates a new directory in the current folder called 'info1910'. Directories can be **rem**oved 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 info1910 directory you would simply `cd info1910`. Along with chanding directories, we can print our **p**resent **w**orking **d**irectory with the `pwd` command.

Question 2: 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.

You can also revert to the directory you were immediately previously in with `cd -`.

Tab Completion

Typing out the full names of each folder every time is somewhat redundant. 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. Fancier shells also contain inbuilt completions for arguments passed to commands along with device drivers and other useful features.

From the info1910 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. Once you've entered this command use tab completion to navigate back to your info1910 directory.

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

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 **co**p**i**ed using `cp` **m**o**v**ed with `mv`, and **re**m**o**ved with `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!"
```

Question 3: Recursive Remove

Make a new directory titled 'delete_me', and in it create some empty files with the `touch` or `echo` command. Move back to your home directory and using a single bash command, remove the 'delete_me' directory and all it's subdirectories and files. The manual pages will likely be a great help here.

Piping and Redirection

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 >> symbols 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 controls in a similar manner to the `man` command you saw earlier.

Question 4: Dictionary Search

Linux comes with selection of dictionaries they can be found at **usr/share/dict**. Grep, pipes and word counts will help you here.

- Find all words in the dictionary containing the substring 'them', count how many of them there are.
- Help me at hangman by searching that file for all words that contain no vowels. Put them in a file for me.

Question 5: Repeat

Linux comes with a number of configuration files that control the behaviour of just about everything on the computer. You might have noticed that the 'up' and 'down' arrow keys allow you to view previously entered commands. This feature is supported by the '.bash_history' file (you may have a slightly different file depending on your shell, check your home directory). Use `cat` to have a look in the file and see that your previously entered commands are all there.

- Count how many times you've used the grep command.
- Using the `head` and `tail` commands, along with `/bin/bash`, write a single line of bash that re-runs the last entered command.

For future reference, the shortcut `!!` expands to the previously entered command. Try it.

Question 6: Manipulating Data

For this problem we would like to print the sizes in bytes of the largest three files in the current directory.

You will find the following commands useful. As ever the `man` pages will provide more information on their use.

- `awk` - `awk '{print $5}'` will print the 5th column from the stream
- `sort` - Sorts the stream
- `head` and `tail` - Displays the first or last lines of the stream

Vim and Emacs

To edit a file from the command line, there exist two common text editors. These are **vim** and **emacs** and you should learn to use at least one of them.

Vim is smaller, and somewhat less featured while Emacs rivals operating systems in size but is more bloated.

Before you open and use either of them, it is incredibly useful to know in advance how to close them again. In vim you will be looking to press the escape button and then to enter the ':q' command, or the ':q!' command if you're really stuck.

In Emacs you're going to need to press 'Ctrl + X Ctrl + C', then either save or clear the buffer and answer 'yes' if prompted whether you'd like to exit anyway. In the emacs prompt control is abbreviated as a capital C.

To use either to edit a file simply enter **vim my_file** or **emacs my_file**.

In vim press the 'i' key to enter 'insert' mode, and type what text you need to. Then press escape to return to the 'command mode' from where you can save with ':w' and quit with ':q'. You can also combine these commands to ':wq' to save and close the file and vim.

In emacs you can immediately type your text into the 'buffer' which will be periodically auto-saved to the file. You can also force it to save using 'C-x C-s' or 'Ctrl + x Ctrl + s', and then quit.

This is a very brief introduction to these two editors and you should practice using them and learning other commands and shortcuts for each in your own time.

Directories, Devices and other Files

Folders or Directories are also files, opening a directory using vim or emacs will show some metadata about how the system interprets the directory along with a listing of the path to the directory and the files within it. The path to the current working directory of your terminal can be displayed using the **pwd** command.

Not all files are physically stored on disc. For example the current time is stored entirely in memory as a file found at */proc/uptime*. Use cat to view the contents of that file. What happens when you cat the file a second time?

Devices are also treated as files, with **lspci** showing devices connected on a PCI port, **lsusb** showing devices on a USB port and **lsblk** showing block devices. These commands also show the path to the files that represent the connection to these devices. By reading and writing to these files it is possible to read and write to and from the device itself.

Another useful command is **du** which will show the current disk usage of different files and folders, allowing for easy cleaning when the disk starts to fill.

Question 7: Drives and Network Shares

Have a look at the */etc/fstab* file (don't try to change it!). This file tells Linux what devices are to be mounted automatically, what the UUID of the device is so that it can be uniquely identified and what format the device takes.

- How many drives are currently mounted on your computer? Where is your home directory actually located?
- Do you have a SWAP partition, what is it used for?

Question 8: Global Regular Expression Print

You may haved noticed earlier when using grep that the output is displayed in red. Have a look at the */etc/profile.d/colorgrep.sh* file.

- 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?

Question 9: Mounting and Unmounting

If you have a USB, plug it into your machine and using *lsblk* and the *mount* command, mount and access the contents of the USB. The *umount* command can be used to safely unmount a mounted device. Be careful not to unmount the drive containing the operating system! It is a common convention to mount temporary devices to */tmp*. You may need the `sudo` command here!

- Where is the mount command located?
- What are the permissions on the mount command? What looks odd about these permissions?

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.

Processes

A process is simply a currently executing program. Each process is assigned a unique process ID (PID) which can be used as a handle to start and stop it. A currently running process in the foreground can also be stopped using *Ctrl + C*.

The **ps** command displays a list of currently running processes under the current terminal process. The **-au** option for ps will list all processes on the system, including those of other users. For a more interactive display the top or htop commands can be used, these will also show the load on the CPU, the amount of RAM in use, the number of threads currently running and other useful diagnostic information.

Processes are spawned by passing a file with execution permissions to the kernel. The terminal you are currently using is itself a process and the commands you type are interpreted by the process as paths to executable files that are then run.

Your particular Linux distribution comes with a number of executable programs that perform common tasks such as changing directories, making and reading files. Some more advanced processes can be run in the background so that you can continue to use your terminal as they run. In order to background a process simply end the line containing the command' with an amperstand]it &. A backgrounded process can be foregrounded again using the **fg** command.

Question 10: Starting, Killing and Backgrounding Processes

The **ping** command is a small network utility that sends packets to a target IP address and waits for a response. It is quite useful for checking if another computer is connected to the network. You can check your own IP address on each of the networks that your computer is connected to using the **ip addr** command.

- Ping your own computer using the ping command, if you are unsure how to use ping use the **man** command.
- Stop the ping process using *Ctrl + C*.
- Start ping as a background process.
- Use the **kill** command with the process ID from the ping command to kill the ping process without foregrounding it.

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. From here you can surmise that 'executing' a file is really just about reading the hashbang on the first line, then passing the file to the relevant program.

Question 11: Shell Script Basics

- Convert your solution to the 'last command' problem into a script called 'last.sh'. Don't forget to give your script execution permissions!
- Write a shell script that prints your username by looking at the path to your home directory. You may find ~or $HOME and the `sed` command helpful here.

Shell Scripts Continued

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
```

One of these variables is the 'path'. When you enter a bash command, it searches all directories stored within the 'path' for executable files matching the name of the command, if one is found it executes it. New commands can then be added to bash either by adding an executable in one of the locations specified by the path, or by adding a new location to the path. Typically this can be modifed just for the current user by changing the path in their *.bashrc* file.

Echo the **PATH** variable and see what locations are currently listed. Use the `which` command to find where some common bash commands are being executed from.

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
```

We can also make use of wildcards rather than explicit commands:

```
for file_var in * ;
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.

If you had sufficient permissions, instead of aliasing your script you could instead provide a symlink from the /bin directory. For the sake of posterity, you can check the word count of your script using the `wc` command, and the line count using the `wc -l` command.

Question 12: Bash Loops

Rewrite your solution to the dictionary search question using explicit loops.

Question 13: File Rename

Fill a directory with `.txt` files with a range of names. Write a bash loop to rename each of the files so that it now ends with `.c` instead.

The following bash commands may assist greatly, start by trying each of them in a loop and echoing their output.

```
${filename##*.}
${filename%.*}
```

Question 14: Automarker

Write a bash script that takes a defined command and tests it for expected input and output. For instance, in a directory 'ls_test_1' containing the file 'qwop', it would be expected that `ls ls_test` would return 'qwop'.

- Use pipes, the diff command and the previous dictionary tasks to test the `grep` command (remember that you can cheat your input and output here, we're really interested in building the marking script)
- Modify the program to read each test from an input file with a '.in' extension, and compare to an output file with a '.out' extension.
- Write a hello world program in another language that you already know. Modify your script to compile and run your program, test the output!
- Modify your program to read a line from standard input and count the number of times the letter 'e' occurs within this line.
- Modify your script to pass input to your program.
- Rewrite your 'e' counter in bash.
- For each word in the dictionary, compare the results of your bash e counter and your e counting program.

Desktop Environments

The window manager itself in Linux is a process that is managed by the kernel and provides a desktop environment with extravagances like mouse support, icons and windows. The window manager can be run using the **startx** command.

Find what the startx command does and what files might you edit to modify the behavior of the window manager. How does this relate to xinit?

If you're a windowed environment, find and kill the xinit and processes. Then run the startx command to return to the safe and familiar land of desktop environments. If you've done the tutorial up to this point purely in a tty session (nice work!) then simply `startx`.

You should by this point see that the Unix operating system is simply a series of files that communicate through processes to run just about everything the operating system does. The flexibility afforded by this scheme means that modifying any of these files allows a user to tailor the system to their own ends and it has been squeezed onto just about every device that has been made.

Homework

There were possibly quite a few new concepts and commands in this tutorial. Without practice it's likely that you will forget them all by next week.

Your homework is this; stop using a file browser. Spend the next week navigating and launching programs exclusively with the terminal while you familiarise yourself with the environment and learn the commands that will be relevant on a day to day basis.

There are many more useful commands out there, that you will hopefully learn and accumulate over time.

Extension Problems

Question 15: Your own Text Scroller (Extension)

Write a bash script that tracks your place for the purposes of viewing a selected number of lines of a text file. Don't forget the hashbang!

- The script should read from a configuration file stored in an appropriately named directory in .config indicating what file it is to be reading from and the last read line.
- The script should have a flag that changes the file that it is reading
- When invoked using the -n <number> flag the script should read the next number of lines specified.
- When invoked using the -s flag the script should re-print the same
- When invoked using the -p <number> flag, the script should print the previous number of lines specified.
- When invoked without a flag, the script should set the new target file to whatever file the user has specified as an argument and reset the last read line to 0.
- Alias your script in .bashrc with the absolute path to the script to use the command without needing to replicate the path from your current directory to where the script is stored