Lecture 02: Unix

I am going to make a few assumptions about your knowledge of UNIX. As a second year, I am assuming that you have used it a bit, perhaps not a lot, but at least in workshops, practicals, and tutorials. However, being able to use UNIX and really understanding what it is, or really what an operating system is, are two completely different ideas. A bit like speaking a language and understanding the grammar (many can do the former but not the latter).

If you think you already know all this, please just read the section on **chmod** as it will have catastrophic consequences for your assignments.

Operating Systems

An operating system is effectively some low-level software designed to deal with low-level problems. These include talking to devices, scheduling/running programs and managing tasks. Sometimes, when writing a higher-level program, you want to be able to interfere with some of these things. That is really the essence of this course.

Unix

So, what makes Unix different from other operating systems? Why would you bother with it instead of using Windows or Mac OS. Unix is a bit more friendly to programmers and advanced users. There is something inherently a bit lower level in Unix than either of the other two which means that if you understand what is going on, then you have a lot more control over things. Windows and Mac are designed with naïve end users in mind (which is not a bad thing) but does mean for advanced users there can often be painful roadblocks to getting the system to do what you want it to do (Microsoft and Apple have other ideas). Some of this comes down to Unix's design philosophy.

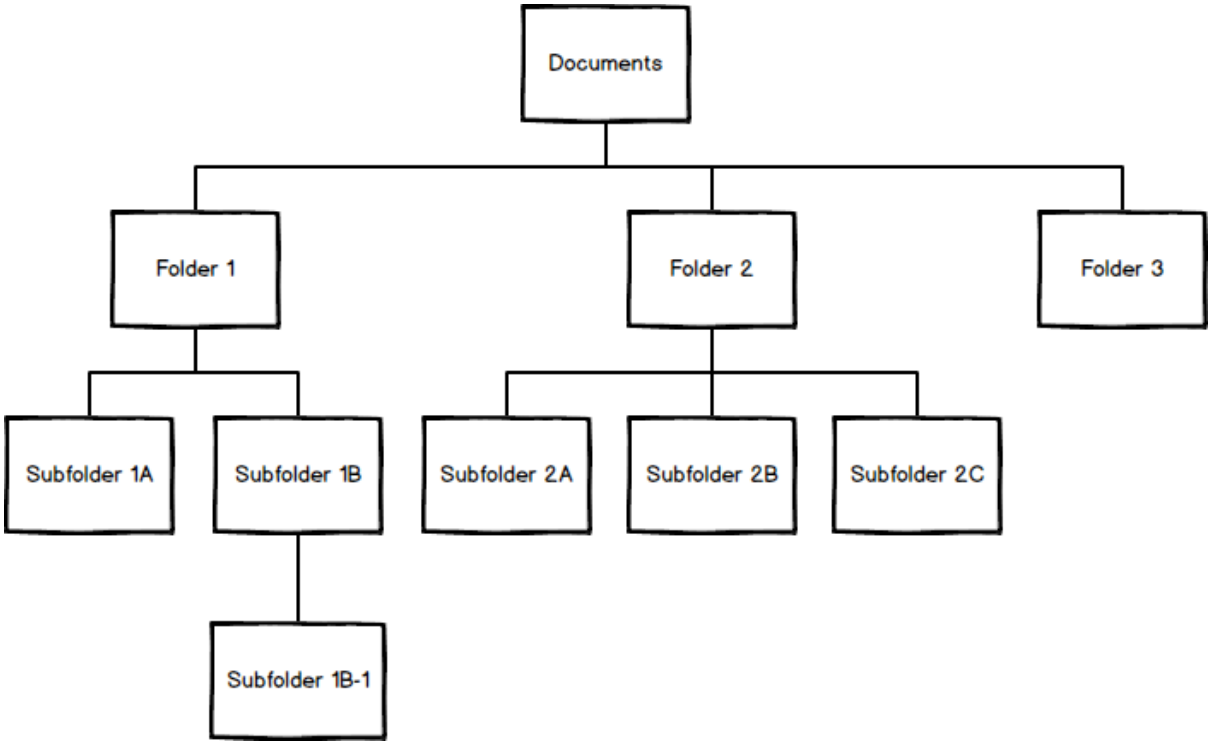
Unix Design Philosophy

Everything is a File.

This sounds simple enough, but it likely extends a bit further than you might expect. Obviously, files are files, but so are directories, executables, and devices. Even running processes can be treated like files. What this really means will be discussed later, but for now, the main takeaway is that by standardizing the interface (i.e., treating everything the same) there are certain advantages and far less 'exceptions'.

The File System

If everything is a file, then everything can neatly be catalogued. Directories (which are files) are merely lists of other files. If everything is a file, then a simple tree structure (see below) allows the operating system to know where everything is. It is all pretty neat. Even things like executables (which are files) are contained within this structure.



Navigation

Navigating this tree structure is fairly straight-forward. Open a shell. Type "pwd". The shell should print out your current location within this tree structure. The top-level directory is merely described as '/'. So, if you are in /home/Bernard, it translates to you being in the base directory, then in the home directory, then in the Bernard directory (essentially you are two levels down). A simple analogy is that you are current in /Earth/Australia/Adelaide (well maybe not but you know what I mean). A useful exercise is to use the following three commands:

- pwd — Prints out where I am
- ls — (L for Lima) Prints out what is in my current directory
- cd <dir_name> — Changes the current directory to the specified one

Using these you can find your way around. Of course, you need to know a few more tricks for this to really work because it is easy enough to enter a directory then see what other directories are available, but eventually you will reach the bottom and be in a directory with no new directories. Interestingly in Unix, each directory contains two special files. One, entitled '.' (i.e., just a single full stop) and another entitled '..' (i.e., two full stops). '.' represents the current directory, '..' represents the parent directory (i.e., the directory in which the current directory is found). Clearly, '..' is quite useful. The command 'cd ..' takes you up one directory. However, '.' seems at first glance to be not that useful. Using 'cd .' does nothing. It should be noted that 'cd' isn't the only command available and that sometimes you want to specify the current directory as a parameter to a command. This is

when '.' is useful as you can essentially say 'some_function .' which means use the current directory as the parameter for the function. The copy command (cp) is a good example (copy something from somewhere else to 'here').

While it may not have been entirely obvious up until now, there are two ways of navigating. One is via a relative path (i.e., 'cd my_dir') and the other is by absolute path (i.e. 'cd /home/Bernard/dir1/dir2/my_dir'). Both have their uses. Using absolute paths can make code less portable (because things may be in different locations) but as we will learn soon, every bash script should contain at least one absolute path in it.

Finally, if in doubt, cd '/' is a thing. This takes you to the root directory.

File Permissions

Now it may not be an obvious leap, but if everything is a file, then there are some questions about who can view, edit, or execute different files. One can imagine that the original operating systems (very old) did not consider this problem at all, but once multiple people started using a single computer it certainly would have ("Who keeps messing with my stuff!?!?"). Clearly the operating system itself has certain files it does not want users to access (as it would crash the whole system). Well, perhaps it would be ok for an administrator who understands the consequences of what they are doing, but usually a big red flag saying 'this is a bad idea' would be good here.

What about different users? What if you have an admin, and a few other users. Obviously, there are things that only the admin should touch, but also individual users probably want to segregate their own files. Unix handles this quite simply.

- **Users**
  - Firstly, there are users. When you open a shell, you are someone (an ID) and the operating system has a list of 'someones' (IDs). The operating system can use this ID to check whether a file is accessible or not.
- **Groups**
  - Unix also has a concept of group which is a set of users. This allows files to be shared within a group but not between groups. Any individual user can be a member of 15 different groups.
- **Other**
  - Finally, there is 'other' which is basically everyone and anyone.

Unix also divides things into three kinds of operations which are fairly intuitive.

- **Read**
  - You may look at the contents of this file.
- **Write**
  - You may change the contents of this file.
- **Execute**
  - You may treat this file as a program.

Read-Only files make a lot of sense as well as files that may not be executed. One quirk of this is that you can have a file that you can **write to** but cannot **read from**… which is strange.

Before using 'chmod' (the command that lets you fiddle with these permissions) it is probably a good idea to have a look at '**ls -l**'. Like 'ls', this lists the files in a directory. Unlike 'ls' it will give a lot more

information about each file. The first 10 characters of which will look a bit opaque until you realise it is file permissions (plus the directory flag). A classic example will be:

-rwxr-xr--

I have colour coded the above for convenience. In essence, the black dash says that the file is not a directory (it would be a 'd' otherwise). The red '**rwx**' represents read, write and execute permission for the user (i.e., you). The green '**r-x**' indicates read and execute permissions for the group (i.e., if you aren't the owner of the file but are in the same group) but no write access. Finally, the blue '**r--**' indicates that all users can read the file but can neither write nor execute it. It is important for assignments to give global execute access for the file otherwise the marker script will fail.

chmod

Warning

*When submitting an assignment, remember that your bash scripts **must have global execute permissions** otherwise the marking script will not be able to execute your code (a bit unsurprising really). Unfortunately, **svn** maintains the permissions of the file even if you modify it locally. This means that if you 'add/check in' a file with no global execute, then changing it may require deleting the file from svn and re-adding it. You have been warned.*

The 'chmod' function takes two basic parameters (it can take more but in its simplest form it is two). The second is the file you want to change and the first is what you want to do with it.

Example

```
chmod +x my_script.sh
```

This will change the execute flag for all users. Intuitively, +r, +w and +x grant read, write and execute while -r, -w and -x remove such access.

You can also be more specific.

```
chmod u=rw,go=r my_script.sh
```

The above will set the user (u) to read and write (rw) while setting the group (g) and other (o) to read (r).

If you feel like being 'clever', there is also a binary way of setting permissions.

```
chmod 755 my_script.sh
```

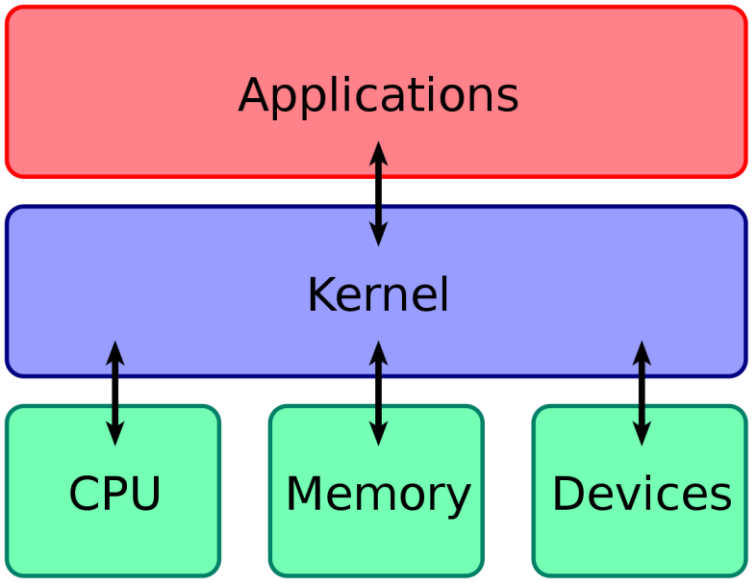
The numbers represent the 'bits' of the command in octal. 7 is equivalent to 1, 1, 1 which is equivalent to +r, +w, +x for the user. 5 then is 1, 0, 1 which is +r, -w, +x for the group. Do I need to explain the second 5? Probably not.

The Kernel

It is odd with computer science that sometimes you can dig your way down deep into a topic (such as files) while simultaneously missing something fundamental. It is a bit like diving down one file structure only to have missed something in a directory in the root directory. When I was in university, I had heard the word 'kernel' thrown around a lot without really having any idea as to what it meant. One does not really need to understand the 'kernel' to be an able practitioner, but it is still useful.

So, what is 'the kernel'. The kernel can be thought of as the base program operating on a machine. When you boot, you are booting the kernel and the kernel is the thing upon which everything else runs. The kernel interfaces to the CPU, memory and peripherals whilst simultaneously talking to applications (i.e., programs you might want to write). If you are talking to the mouse in your code and asking where it is on the screen, you are doing so through the kernel.

The Kernel also manages things like files, users, devices, and processes (which it manages a bit like files).



Even the terminal into which you write commands is a device managed by the Kernel. 'Device?' I hear you say. Well, a terminal was technically an old box with a keyboard. Nowadays you use a 'terminal emulator' (i.e., a purely software version) such as **gnome-terminal** or **xterm**.

For reference some useful functions for looking at devices are:

- lsblk
- lspci Ooh… PCI stuff
- lsusb Ooh… USB stuff

Processes

A quick lesson in computer science terminology.

A program is a series of instructions written by a programmer. A good analogy is a recipe in a recipe book. Multiple people can use the same recipe. You can (if a good cook) follow multiple recipes at once resulting in several dishes being prepared. A recipe without a cook does nothing.

A process is the execution of a program. In our recipe analogy, the process is act of doing the cooking. This means that at an instant of time, you will be at a specific instruction of the recipe (i.e., about to crack eggs or some-such). This kind of means that the operating system is the cook. Which for a good cook (or operating system) means it can cook multiple things simultaneously. Some recipes also have several parts that can be done independently (think of marinades or sauces). If you have a sauce heating while you cut up other food, the operating system (you) is essentially running two processes (cooking and cutting), potentially from the same program (recipe).

An even better analogy for the operating system is that of a head-chef in a restaurant. The head chef will often have access to many kitchen hands and other chefs (resources) which they can assign to different tasks (i.e., processes). The individual chefs can invoke other chefs to do jobs for them (processes spawning other processes) or they can hand their responsibilities over to another chef (different resources performing the same process). In this analogy the head-chef (kernel) is only responsible for managing other processes rather than actually cooking.

Back in computing land, there are a few facts about processes which are useful to know. Each process is associated with a user (i.e., the computer user who is the 'owner' of the process, i.e., maybe who ordered the food in the first place?). Also, generally speaking, if a parent process dies, the child also dies. In our analogy if the head chef cancels an order, all chefs associated with the task stop making the dish. This is not an absolute fact however and processes can be made to outlive their parents.

The Shell

So, we have a **kernel** that runs **processes** which fiddle with **files**. These **processes** are run from **programs**. What is the shell then? A shell is a **process**! But what kind of process? The shell is an interactive process designed to run other processes. It basically exists to simplify starting/stopping programs.

It is kind of useful for testing a few of the above claims too. For example, if you run 'gedit' or something in a terminal, then close the terminal, then 'gedit' will also close (an example of children being dependent on their parent).

The terminal is associated with a user, so file permissions within a terminal are user-dependent. You can change user within a terminal such as using 'su' (which gives admin access… though you will need a password).

That's basically it except for a few other random pieces of information. Firstly, not everything is run from the shell (the Kernel being the most obvious example). These processes tend to be quite independent things that respond to external events.

Basic Unix Commands

There are a lot. Here is a list of some useful ones and a brief description of what they do. For more details either google it or type 'man <instruction>' to get the manual and usage. Learning to read the 'man' pages is a useful skill to be developed as Unix commands are very powerful but a bit opaque as to how they work. I mean, 'grep' was the first acronym I thought about when looking for a 'searching program'… (rolls eyes).

ls	List the contents of the current directory
cp	Copies a file from one location to another
mv	Moves a file from one location to another
rm	Deletes files
pwd	Where am I?
cd	Go here.
less	Show me a bit of the specified file
cat	Concatenate two files… or just show me what is in one
grep	Find the files which contain some text I specified (good for finding config files where you know what it contains but not what it is called)
head	I only want a few lines of this output from the start
tail	I only want a few lines of this output from the end
wc	How many lines/words/characters are in this text
ps	What processes are running
kill	Get rid of a process
cut	Grabs parts of a file you want for some purpose, useful for managing data from .csv files.
paste	Glues files together vertically… so the first line of a file is glued to the first line of another file forming a new first line. Again useful for .csv files.

Redirection

This is a bit of a change of topic but kind of goes with 'basic commands'. One thing you will see a fair amount of in Bash is the use of the '>' and '<' characters. These are used for 'redirection'. Without going into large amounts of detail, programs in bash tend to 'eat lines of text' and 'spew lines of text'. For example 'ls' eats nothing but spews out a list of files in a directory. 'wc' eats a series of lines of text and spews out a formatted output which has number of lines/words/characters.

Redirection is all about choosing what a program eats (input) and where it spews (output). If you want a program to read from a file rather than user input, you can use:

```
my_script.sh < an_input_file.txt
```

Likewise, if you want your script to save all its output to a file, you can use the following:

```
my_script.sh > an_output_file.txt
```

Again, without flooding you with details, remember that in Unix, everything is a file. So, if you can redirect to and from 'files' you can sort of connect anything together. It is very powerful. Also… it can lead to some confusing code.

Piping

There is also piping '|'. This is used to chain commands together.

For example:

```
ls | wc -l
```

'ls' takes no input and simply outputs a set of text detailing the files in a directory. 'wc' takes these lines of text and counts them. With the '-l' flag it simply returns the number of lines.

Often a clever use of in-built commands and pipes can create convenient (although often difficult to read) single-line commands which perform a specific task. Learning some of these can save you a lot of time and effort especially when dealing with large numbers of files.

C and Unix

So why C? It does seem a bit arbitrary that Systems Programming teaches Bash and C. Admittedly Bash is 95% the same as other shell scripting languages like tcsh, zsh etc. However, C is a bit unique when it comes to high level languages. Firstly, the two are highly coupled. Unix, though originally coded in assembly is itself written in C (mostly). As you can imagine this gives C some special advantages when interacting with Unix as the two 'speak the same language'. Many of C's inbuilt commands are designed specifically to directly interface with Unix.

Examples include 'system()' and 'execvp()' (which we will cover in detail later) which are basically stand-ins for typing Unix commands mid-way through a C program.

I'm not going to pretend that I have ever used C for anything in any of my jobs. Most of my programming experience was in Java, Python, C++, Ada and Fortran (so… old). Yet, understanding how C works is one of those things that is useful indirectly. Mostly because when you are talking about C you are usually talking about lower-level tasks like Signals, Pipes, IO and so forth. It is useful to know these things exist and roughly how they work. When using a high-level language, these tasks are often wrapped up in libraries where you do not really get to see what is going on. Ironically, knowing more about C can make you appreciate the conveniences of those languages all the more (and what might be going wrong under the hood).

Update 2022. I have used C in a job! It was for translating code into FPGA 😊