Linux-Fu

- Just enough command line knowledge to make you dangerous

R.I. Ferguson[1]

1 Abertay University

Dedicated to all those who took the time to pass on their Unix/csh/ksh/bash/Linux - fu to me. Take a bow Chris Bowerman, Alan Lumsden, Arthur Wyvill, John Cartledge, Kevin Hayton, Dave Webster, Ian Gordon, Kenny Duffus, Matt Pollard.

Table of Contents

1 Introduction

1.1 Basic concepts

“This is about how to use the Linux command line.”

“But I use Windows. Why am I doing this?”

“ Because this is how pros do stuff.”

The whole idea of WIMPS (Windows, Icons, Menus, Pointers) interfaces (or GUIs) is that they don't require the user to have a 'mental model' of what's going on 'behind the scenes' in a computer system - they try not to put to much load of a users short-term memory and cognitive skills.

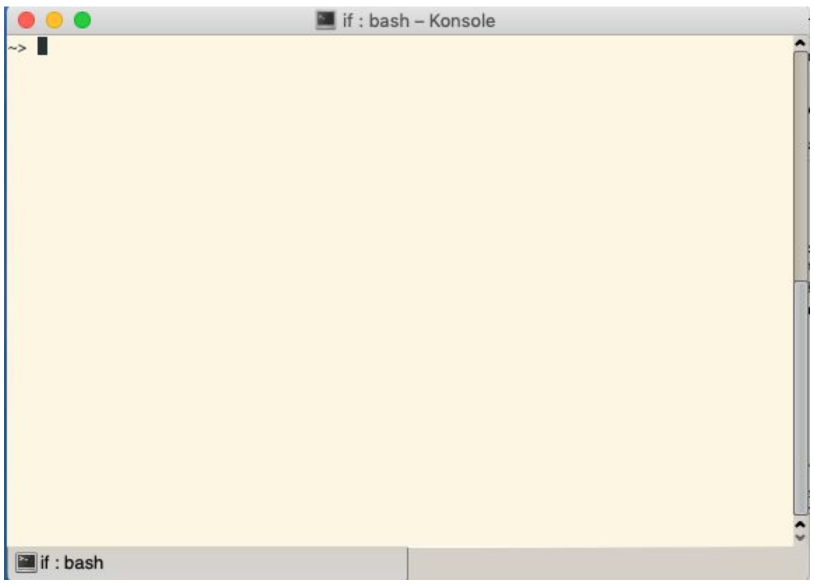
Command line interfaces are the reverse. They expect the user to know what is happening and to remember the commands and syntax thereof. If you are going to be a professional in any form of IT, then you DO need to know what's happening, so why not use a CLI (command line interface)?

Whilst there is nothing wrong with the WIMPs paradigm, once you have overcome the steep learning curve associated with the command line, you can expect to be faster at accomplishing many of the routine tasks associated with setting up, managing and using computer systems and to have a powerful, flexible tool for solving many kinds of information systems problems, literally, at your fingertips.

1.1.1 The Command Line and Commands

The basic idea behind the command line is that you type in the name of a command and this causes the command to be run, but before you do that, you'll need to use the GUI to run a 'terminal' in which there is a command line interpreter (or 'shell') that 'listens' to your typing.

There are a number of different terminals (e.g. terminal, xterm, konsole), available depending on which version of Linux you are using.



```
~>
```

Terminal running under Ubuntu

It's worth knowing that you are dealing with two programs here – one is the terminal (the graphical window on the screen, responsible for appearance etc.) and the shell, the thing that deals with the text.

Try it now:

Activity: Run a terminal

1. In the GUI find and run the 'terminal' program.

1.1.2 First Commands

Activity: Existential angst

1. Type `whoami` and hit return.

You should be rewarded with your username.

2. Type pwd and hit return. (pwd = print working directory)

You'll see the path to the directory you are currently working in.

3. Type `ls` and hit return. (ls = list)

You should see a list of all the files and sub-directories in your current working directory.

1.1.3 Fine tuning commands - Parameters

Many commands take parameters to modify their behaviour. The ls command can take the name of a directory that you want a listing of.

Activity: Listing a different directory

1. Try running `ls /etc`

Note that when you use ls WITHOUT giving it the name of a directory it assumes you want the current working directory. This is known as a default.

1.1.4 Fine tuning commands - switches

Many commands allow you to 'fine tune' their output by specifying 'switches'. You've already seen the default format of the output from the 'ls' command. More information can be obtained by adding the '-l' switch.

Activity: More info from `ls`

1. Try running `ls -l`

1.1.5 Getting Help

There are a lot of commands in Linux and they take a lot of switches. Soon enough, you'll remember the common ones but it's useful to be able to look up info on a given command. You could use Google (Other search engines are available. Really? Does anyone actually use bing?) however, there is a command for this: man (short for manual).

Activity: Using the manual pages

1. Type:

```
man ls
```

You'll see the manual entry for the `ls` command including all of the switches and parameters is supports. Don't worry about reading all of them now but do remember the man command.

2. Use the up and down cursor keys to scroll through the text
3. Type 'q' to quit man.

Note: Linux is a case-sensitive operating system so 'LS' means something different to 'ls'.

And whatever you do, don't make the mistake of type sl instead of ls..........

2 Files and directories

2.1 Tree structures

You'll be familiar with the idea of files and directories (or folders if your from the Windows world) but formally, information is stored in a hierarchically (tree!) structured database known as a filesystem.

Directories in Linux all have names and a path-based scheme is used. Unlike Windows there are no drive letters and the root directory a Linux is known simply as '/'.

A key concept working with Linux is the idea of a current working directory. You can only be 'in' one directory at any time. When you first log in and run a terminal you will be the 'home' directory of your account. You can make any other directory the current directory by 'changing directory' (or cd-ing) to it:

Activity: cd-ing

1. Type: `cd /`

You are now in the root directory.

2. You can prove this. How? Hint: You've already seen the command to tell you the working directory.

3. List the contents of the root directory.
4. Return to your 'home' directory: Type `cd`

'cd' without any parameter defaults to your home directory

Note that the 'prompt' (the text that appears at the start of the current line) should also give you a clue as to which directory you are in.

Now that you can move from one directory to another, we can turn our attention to storing information.

2.2 Creating and removing directories – `mkdir`, `rmdir`

You can create new directories. There's a command for that: mkdir. Let's create some directories within your home directory. Any directory that exists with another directory is said to be a sub-directory, so you're about to create three sub-directories of your home directory.

Activity: Making directories with mkdir

1. Make sure you are in your home directory. (Hint: cd)
2. Type:

```
mkdir uniStuff
```

3. cd to uniStuff by typing:

```
cd uniStuff
```

4. (Rocket science huh?)

5. Type:

```
mkdir year01
mkdir year02
mkdir year03
mkdir year04
```

6. List the contents of your home directory.
7. Make year02 your current directory with cd
8. Type:

```
mkdir CMP209
mkdir CMP210
mkdir BSW211
```

9. List the contents of the current directory

Oops! Mistake! I don't take the class BSW211. I need to get rid of it. No problem, the command rmdir (remove directory) is your friend.

Activity: Deleting directories with rmdir

1. Type:

```
rmdir BSW211
```

2. List the contents of the current directory
3. cd into CMP209

2.3 Navigating Directories - cd

After that last exercise, you should be in CMP209. Suppose you want to be back in year02. You could cd to your home directory and then cd to uniStuff and then cd again into year02 but that's waaaaaay to much typing. Instead you can use a bit of Linux shorthand: two dots together means 'the parent directory' so cd .. means go up one level

Activity: cd-ing with ..

1. Type:

```
cd ..
```

2. Use pwd to confirm you are where you think you are.
3. Go to your home directory (anyway you like).

Oh d*mn! I need to do some more stuff in CMP209. cd uniStuff, cd year02, cd CMP209....... tedious. Instead you can just give a complete path:

Activity: cd-ing by giving a path as a parameter

1. Type:

```
cd uniStuff/year02/CMP209
```

2. Use pwd to confirm you are where you think you are.

That's a bit better but still to much typing for the lazy amongst us (or those who want to to stuff fast!) One of the best kep secrets for using the CLI fast is 'command auto-completion' . Watch this carefully.......

Activity: Saving a shi......shed-load of typing

1. Go to your home directory (anyway you like).

2. Type:

```
cd un
```

3. Press the tab key

Shi...er... Magic happens!!

4. Type:

```
y
```

5. Press the tab key

More magic happens

6. Type:

```
2
```

7. Press the tab key

More magic happens (you get the idea now?) This is even more magical:

8. Press the tab key again
9. ...and again....
10. Type:

```
0
```

11. Press the tab key again
12. Hit return
13. Use pwd to confirm that you are indeed where you think you are.

Command completion is a really powerful way to speed up your interaction with Linux. In modern Linux distributions it even knows which command/programs you have installed and will a) complete the name of the command and b) only fill-in files which are appropriate to that command.

2.4 Getting a quick view of a directory structure - `tree`

Okay, so that was quick way of cd-ing around the place. Now here is a quick way of getting an overview of what you have stored: The tree command

Activity: Can't see the wood for the trees?

1. Go to your home directory:

```
cd
```

2. Type:

```
tree
```

HINT: Don't use spaces in directory names or filenames – it confuses Linux, the CLI, command line completion, your lecturer and probably you. Instead of calling a file 'my info.txt' it's better to use something like myInfo.txt

HINT: Now whilst `tree` and the CLI are wonderful, don't make the mistake of turning the CLI into a religion! You might just want to use the file manager to view the directory structure you've just created and confirm that it looks the way you thought. Sometimes the fastest way to do things is by combining the CLI with the GUI. But sometimes all you've got is a CLI.......

2.5 Creating and removing files

2.5.1 `touch`

All the stuff we've done so far is great for getting organised and dividing your work/life/info into different compartments, but we haven't actually stored any information yet. Time to create some files. There are many ways you can do this in Linux and we'll start with one that probably isn't the way you'll do it most of the time, but it's kind of worth seeing once. We're going to use the touch command. e.g. touch myInfo.txt

Activity: You've got the `touch`

1. Go to your CMP209 directory

2. Type:

```
touch myInfo.txt
```

3. Get a directory listing, but use the -l switch to the ls command:

```
ls -l
```

4. Type:

```
touch myInfo.txt
```

5. Get a directory listing, but use the -l switch to the ls command:

```
ls -l
```

6. Can you see the difference between the two listings?

The `touch` command is really for updating the timestamp on a file, it just happens to have the side-effect of creating the file if it doesn't exist in the first place in which case it creates an empty file.

2.5.2 Removing files with `rm`

The command to delete a file is rm (for remove). It takes a parameter: the name of the file to remove.

WARNING: this isn't like the waste bin in Windows/Macos. Once you `rm` something, it's gone (more or less).

Activity: rm

1. Type:

```
rm myInfo.txt
```

2. Use ls to verify that it's really gone.

You can remove multiple file with one rm command by passing a 'wildcard' as a parameter e.g. rm * will remove ALL the files in the current directory.

2.6 Editing a text file – `pico`, `nano`, `vi` and `emacs`

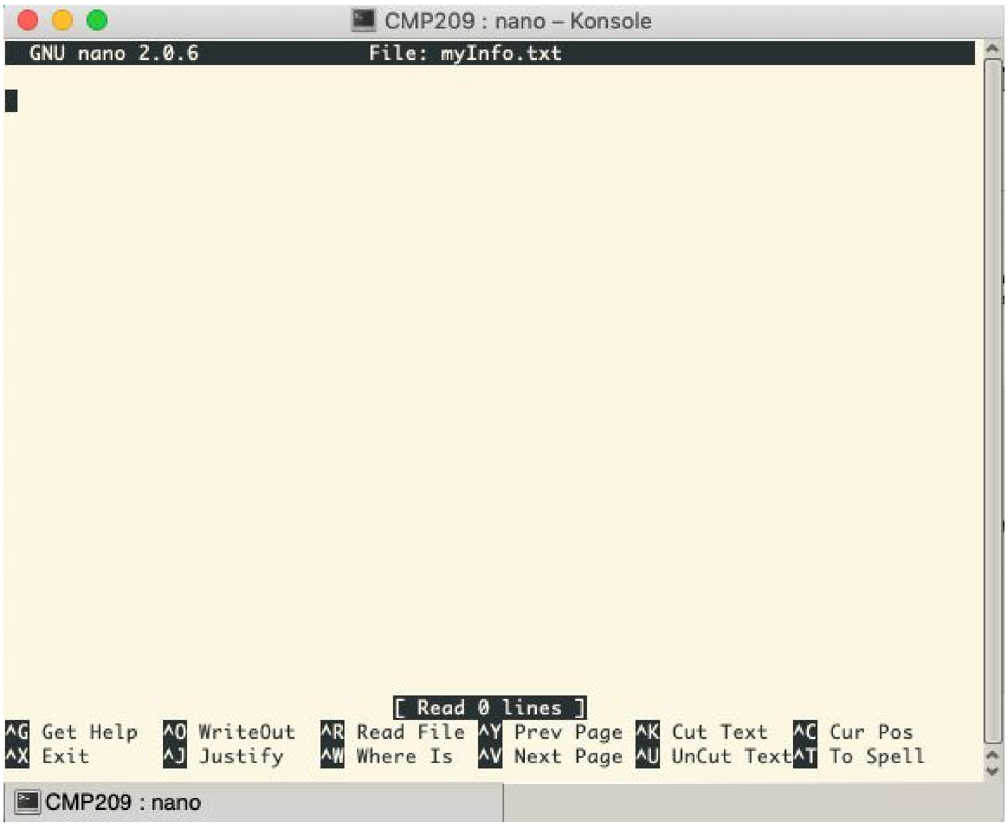
Suppose we now want to put something into that file. There are approximately 1,303,867.9 ways to do this in Linux. Let's go for an easy one first. We'll edit the file you just created. Oh! You just deleted it! Never mind, pico will make a new one for us.

Activity: Editing with pico?

1. Type

`pico myInfo.txt` (Remember you can use completion to enter that name)

You should now see that pico has taken over your entire CLI window.



```
CMP209 : nano – Konsole
  GNU nano 2.0.6               File: myInfo.txt

                              [ Read 0 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
CMP209 : nano
```

Editing with pico

Pico is a character-based text editor. You can just type and your text will appear. To save the file, press ctrl-o (for write-Out). For help, ctrl-g and to exit, ctrl-x.

Activity: Have a play with pico

2. Type some random junk in just to prove you can edit the file.
3. Save it.
4. Exit pico
5. Go back in just to check its all still there.
6. Exit pico
7. Check the file size via the ls -l command.

There are several character-based text editors available under Linux. CLI die-hards will tell you to learn 'vi' (prononced 'vee-aye') as you are pretty much guaranteed that wherever there is a command line, vi will also be installed. Pico (and it's big-brother nano) aren't always available though they are a little more 'friendly' than vi. There's also a phenomenal piece of bloatware called emacs which started life as a text editor but has grown into an all singing, all dancing behemoth: It has a mode for playing tetris, a mode for being a psychotherapist and for all I know, a mode for ironing your socks.

Activity: Playing tetris via emacs

1. At the CLI type:

```
emacs
```

2. then press escape followed by x
3. type

```
tetris <return>

You're on your own after that Ctrl-x Ctrl-c will get you out.
```

Activity: Psychotherapy via emacs

1. At the CLI type:

```
emacs
```

2. then press escape followed by x
3. type

```
doctor <return>

Ctrl-x Ctrl-c will get you out.
```

2.7 GUI text editors – `gedit`, `kwrite`

Of course, things have moved on since command line text editors (this is the Century of the Fruitbat after all) so a more comfortable way of editing text files is to use a GUI editor. Depending on the flavour of Linux you are using, there a several text editors likely to be available including gedit, kwrite, kate. All of these can be run from the menu system, but it can be useful to start them from the command line.

Activity: Comfortable text edit

4. Type:

```
 gedit myInfo.txt
```

5. Make a few changes – just because you can.
6. Save them.
7. Note the whilst the editor is running you can't type anything else on the command line.
8. Close the editor down.
9. Note that the command prompt returns .
10. Type:

```
gedit myInfo.txt &
```

What strange magic is this? The editor is running AND the command line is usable. In a word, multi-processing. The '&' caused the editor to be run as a 'background' task so you now have two programs running simultaneously – the command shell and the editor. In Linux terminology you 'forked' the editor process from the command shell. Indeed that pretty much defines what a shell is – something that takes commands from a user and forks other processes to carry out those commands.

2.8 Quickly viewing a file – `cat`, head and `tail`

It doesn't really take long to run an editor, but sometime you don't need to change a file – you just need to see what is in it. Three commands worth knowing are cat, head and tail. Each of them take a parameter – the name of the file you want to see. `cat` copies the whole file to the terminal, `head` just shows you the first few lines whilst `tail` shows you the last few.

Activity: Meow

1. Linux distributions contain a dictionary (of sorts) in the file /usr/share/dict/words. View it by typing:

```
cat /usr/share/dict/words
```

2. Waaaaayyyy to much: Show me just the start:

```
head /usr/share/dict/words
```

3. Predictable. Show me the end:

```
tail /usr/share/dict/words
```

4. Check the man pages for head and tail to see the switches that they take. Give them a try

2.9 Speeding things up

Can't be bothered with all that typing of long paths? Here are some ways of saving typing:

If you want to run the last command again just type !! and hit return

or you could just press the cursor-up key and it will repeat the last command.

Actually you have access to the last few commands via the history buffer – press the up-cursor key a few times to see what I mean.

Alternatively, just type history <ret> to see what you've been doing. To repeat any command you can type ! followed the number of the command you wish to repeat.

Did you make a mistake and type a command wrongly e.g. car myfileWithAVeryLongName.txt?

No need to type it all again: ^car^cat will repeat the last command with the given alteration.

2.10 Getting out of trouble

Something gone wrong? Try hitting Ctrl-c to stop the current command. Ctrl-d and Ctrl-z might also help you out – but you really need to know a bit more about processes before using them.

2.11 Copying and Moving (cp and mv)

Two particularly useful commands are `cp` and `mv`. Basically they both take two parameters – the name of the file you'd like to copy (or move) and where you'd like it copied (or moved) to.

Activity: Copying

1. Make sure CMP209 is your current directory
2. Copy the myInfo.txt file to the CMP210 directory

```
cp myInfo.txt /home/student/uniStuff/year02/CMP210/myInfo.txt
```

3. Use your file manager to check it ended up where it should have.

mv works exactly the same way, except it doesn't leave the original copy where it was.

2.12 Referring to file and directories – paths

You'll have noticed in the previous activity that you typed the full path to the destination. Actually there are several ways you could have saved yourself some work. All of these are equivalent:

```
cp myInfo.txt /home/student/uniStuff/year02/CMP210
cp myInfo.txt ~/uniStuff/year02/CMP210
cp myInfo.txt ../CMP210
```

Let's look in a little more detail at how they work:

```
cp myInfo.txt /home/student/uniStuff/year02/CMP210
```

This one just misses of the name of file in the destination parameter. If you do that, cp (and mv) assume you want the filename to be the same as the source filename

```
cp myInfo.txt ~/uniStuff/year02/CMP210
```

Here the /home/student/ has been replaced with a tilde (~). The tilde is shorthand for your home directory.

```
cp myInfo.txt ../CMP210
```

This is a 'relative' path. The .. means go up one directory from where you are.

Activity: Copying with a relative path

1. Try using `cp` with a relative path to copy the myInfo.txt file to your uniStuff directory.

2. Use your file manager to check it ended up where it should have.

2.13 A more detailed look at `ls`

This is a good time to look in more detail at the output of ls

Activity: details from ls

1. Go to you uniStuff directory
2. Create some more files:

```
touch usefulData.txt
touch todo.txt
touch results.txt
```

3. Run ls with -alF switches:

```
ls -alF
```

4. You should see something a bit like this (but not exactly the same)

```
drwxr-xr-x    9 if  staff   288 Jan 11 11:48 ./
drwxr-xr-x+ 177 if  staff  5664 Jan  9 15:15 ../
-rw-r--r--    1 if  staff     0 Jan 11 11:48 results.txt
-rw-r--r--    1 if  staff     0 Jan 11 11:48 todo.txt
-rw-r--r--    1 if  staff     0 Jan 11 11:48 usefulData.txt
drwxr-xr-x    2 if  staff    64 Jan  9 11:06 year01/
drwxr-xr-x    5 if  staff   160 Jan  9 11:07 year02/
drwxr-xr-x    2 if  staff    64 Jan  9 11:06 year03/
drwxr-xr-x    2 if  staff    64 Jan  9 11:07 year04/
```

Let's try and decipher that. Firstly what's with the -alF switches?

-l simply means long format - one file per line with plenty of info about the file

-a means all so its shows hidden files as well (see next section)

-F - I think I'll let you look that up on the ls man page.

Now to the output of the command:

The first column (all the drwx-xr-xr stuff) Are the file permission – we'll look at these in detail later. For now all you need to know is that they control who can read, write and execute a given file.

Column 2 is the number of links to the file. Again we'll cover that later.

Column 3 is your username (in my list it's if, your's should be student.)

Column 4 is the group the file belongs to – in my case staff

Column 5 (for files) is the size of the file in bytes.

Then follows the time/date that the file was last modified.

Finally the file name

Activity: Read the man page for `ls -alF`

Like it says.....

2.14 Hidden Files

A hidden file in Linux is one that doesn't show up in ls unless you use the -a switch. Any file whose name begins with a dot, is a hidden file.

Activity: Create a hidden file

1. touch .myHiddenFile.txt
2. Check that you can't see it with ls.....
3. ….and that you can with ls -al
4. use mv to change the name to myUnhiddenFIle.txt

5. Check that you can see it with ls.....
6. ….and that you can with ls -al
7. Use mv to turn it back into .myHiddenFile.txt

2.15 `rmdir` redux

Okay, let's suppose (and this is really crazy - you may have difficulty imagining this one) that you'd decided you didn't like CMP209 and that you've dropped the module. I know – absolutely unbelievable scenario, but stay with me. You want to delete everything in the CMP209 directory and the directory itself.

You could try cd-ing to the year02 directory and running the command rmdir CMP209. Let's do just that:

Getting rid of CMP209

1. First lets make a backup of that dir just in case you change your mind....
2. <ake year02 your working directory
3. Type:
4.

```
cp -r CMP209 CMP209Backup
```

The -r switch to the cp command is short for 'recursive' which means go through everything in the directory any sub-directories and copy it as well.

5. Ok, Let's get rid of CMP209.

```
rmdir CMP209
```

Oh dear! It won't let you. One of the safety features of rmdir is that it won't get rid of a directory with something in it. We have two options. We could either delete everything in the CMP209 directory first (easy enough How would you do it?) or we can deploy the nuclear option:

```
rm -rf CMP209
```

-r = recursive – go through all sub-directories removing everything in them.

-f = force – Override all protection (e.g. read-only permission) on files and trash everything.

WARNING: This command is as dangerous as it sounds and you should always count to 10 and check before you press return. You *could* type `rm -rf *` but make sure where you are before you do it. Anyone who has spent any time in Linux has typed it in the wrong place once and yes, I did once do it in the root directory of a system I was working with.

2.16 Other hints for speeding things up

Bash implements the concept of a directory stack. To be honest, I don't find it that useful, but others swear by it so here goes:

Pushd/popd allow you to record the current directory at the same time as changing to another directory. It can be useful if you need to go somewhere else and come back to where you were working.

Activity: Pushding and popding

1. Use pwd to verify your current working directory.
2. Type:

```
 pushd /etc
```

3. Use pwd again

4. Type:

```
pushd /var
pwd
popd
pwd
popd
pwd
```

3 Users, Groups and Permission

Back in the mists of time, before the PC (so that'll be around 1984 then) computers with a Unix-based operating system on them tended to be the size of a desk and would have many users logging into them via physical terminals. For this reason a system of accounts, account names and permissions was introduced. That system is still with us today even though PCs are personal and tend to only have one user. As security under Linux is based on this system it's worth spending some time becoming familiar with the basics of user administration.

3.1 Users

So who has permission to log into your system? How many accounts are there on it?

Listing all the users on your system

1. Have a look at the file /etc/passwd - that's a list of all users. I'm not going to give you the command to view the file, you should know that by now.

How many accounts are there? Well, there's one account per line so if we count the lines we'll know. Wait! Count the lines? You must be kidding - I have computers to do that kind of thing for me......

The unfortunately-named (to British-English ears anyway) 'wc' command (it stands for word count) comes in handy for this kind of thing:

2. Type:

```
wc /etc/passwd
```

3. `wc` will give you three numbers. Use `man wc` to work out what those numbers mean.

3.2 Who's who?

There are probably more users there than you expected, especially given that no extra ones have been added since installation. Many of them are associated with particular services (e.g. the printing service) but one of interest is the 'root' account or superuser. Root is a privileged account which is allowed direct access to various aspects of the system (including hardware devices). You'll notice that each of the accounts listed in /etc/passwd has a unique number associated with it – the UID or User Identity. It's worth knowing that the system identifies you using this number and that your username is just a 'alias' for this. The root UI is always 0.

Sometimes (especially when performing sysadmin tasks) you'll need to be root in order to run certain commands. There are several ways of doing this in Linux – You can log in as root rather than student, you could use the su command to asSUme the identity of root, but the safest way is to use the sudo command. This allows you to be root for the duration of one command.

Activity: Be root (temporarily)

1. Type:

```
sudo whoami
```

Hey, I feel powerful! Omnipotent!

2. Type:

```
whoami
```

Oh! I'm only me again.

3.3 Adding a new user - `useradd`

One of the things that root can do is add new users to the system (and just as easily remove them). Let's try that:

Activity: Be root and add a user

1. Type:

```
sudo useradd -m catnip
```

2. You need to set a password for the user before they will be able to login

```
sudo passwd catnip
```

Note: you'll be asked three times for a password: first for the sudo password, then for catnip's password and then to confirm catnips's password.

3. Check the /etc/passwd file again to ensure the new user has been added.

3.4 Creating groups - `groupadd`

Linux has the concept of groups of users (think students, staff, admins, etc.) Let's set up a staff group. It's just possible that the staff group already exists on the machine your using. If it does, use your imagination and think of another name for the group. Just like users, groups have unique identify numbers – the gid and like the users,

Activity: Be root and add a group

1. Type:

```
sudo groupadd staff
```

3.5 Adding a user to a group - `usermod`

Activity: Add user to group

1. Type:

```
sudo usermod -a -G staff catnip
```

2. Linux maintains a list of which users are in which group. It can be found in the file /etc/group. Check that file now to verify that catnip is a member of staff.

3.6 Sudoers

One group which is of particular interest is the sudoers group. Only users that are a member of this group are trusted to use the sudo command.

3.7 Passwords - `passwd`

Passwords can be changed with with `passwd` command

Activity: Changing your password

1. Log out of the student account completely
2. Login as catnip and change your password

```
3.      Go back to being student
```

3.8 Permissions - chmod

We've briefly looked a the permissions system when we saw at the output of the ls -l command. Now that we have seen the concepts of users and groups, it's time to look at permissions in more detail.

Consider the output of the ls -l command that we saw earlier:

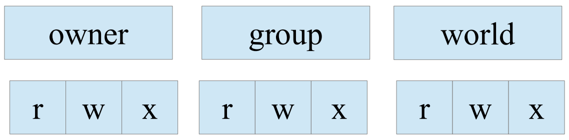
```
rwxr-xr-x    9 if  staff   288 Jan 11 11:48 ./
drwxr-xr-x+ 177 if  staff  5664 Jan  9 15:15 ../
-rw-r--r--    1 if  staff     0 Jan 11 11:48 results.txt
-rw-r--r--    1 if  staff     0 Jan 11 11:48 todo.txt
-rw-r--r--    1 if  staff     0 Jan 11 11:48 usefulData.txt
drwxr-xr-x    2 if  staff    64 Jan  9 11:06 year01/
drwxr-xr-x    5 if  staff   160 Jan  9 11:07 year02/
drwxr-xr-x    2 if  staff    64 Jan  9 11:06 year03/
drwxr-xr-x    2 if  staff    64 Jan  9 11:07 year04/
```

– chmod, chown, chgrp sudo

The bit of interest is the 'permission' field which is the first 10 characters of each line.

The first character indicates whether that line represents a directory (d) or a file(-)

The next 9 characters are organised like this:



Where:

r = read

w= write

x = execute (or the ability to cd into it in the case of a directory)

Taking the file usefuldata.txt as an example, it can be seen that it can be read and written by the owner, it can be read by other people in the group (staff) and in fact read by anyone else. The permissions on a file can be changed using the GUI or by the use of the `chmod` command. `chmod` can be used in two different ways – symbolically or numerically. Personally, I find the numeric way easier butYMMV.

To set permissions using the numeric method you would type something like:

```
chmod 600 myInfo.txt
```

but where does the 600 come from? What does it mean?

Well the 6 corresponds to the owner's permission, the first zero to the group and the second zero to the world but the best way to think of this is in binary. Think of the owner permissions as 3 bits – a read bit, a write bit and an execute bit

If you want the owner to be able to read, write but not execute the file you would set the read bit to 1, the write bit to 1 but the execute bit to 0 – so you'd end up with the binary number 110_2 – which is 6 in decimal – so the owner permission is 6. You don't want the group to be able to access it at all, so read = 0, write = 0 and execute = 0 so 000_2 or 0 in decimal. Same for the world permission 0 – Putting that all together gives 600.

Activity: Setting permissions with chmod

1. Using `chmod` set the permissions of the `results.txt` file to allow the owner to read and write it, members of the staff group to be able to read it and the rest of the world to have no access.
2. Using chmod set the permissions of the year02 directory to allow the owner to read it, write it and change directory into it, members of staff to read it and cd into it and the world to have no access.
3. Using chmod set the permissions of the todo.txt file to allow everyone to do everything to it.
4. Why is 3 a bad idea?
5. Change todo.txt back to its original permission.

3.9 Changing ownership of a file - chown

It's possible to give a file to another user:

Activity: changing ownership

1. Try it:

```
chown catnip myInfo.txt
```

Ah! You're not allowed to. But root is.

2. Try that again but this time put a `sudo` in front of it.

3.10 Changing the group of a file - `chgrp`

You can set the file to belong to another group (that you are a member of).

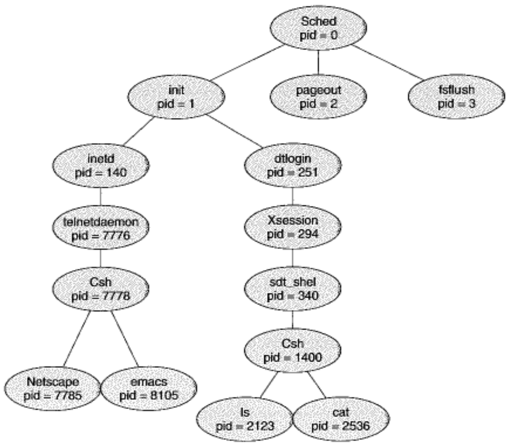
e.g. `chgrp` *filename*

4 Processes and Jobs

4.1 Process models

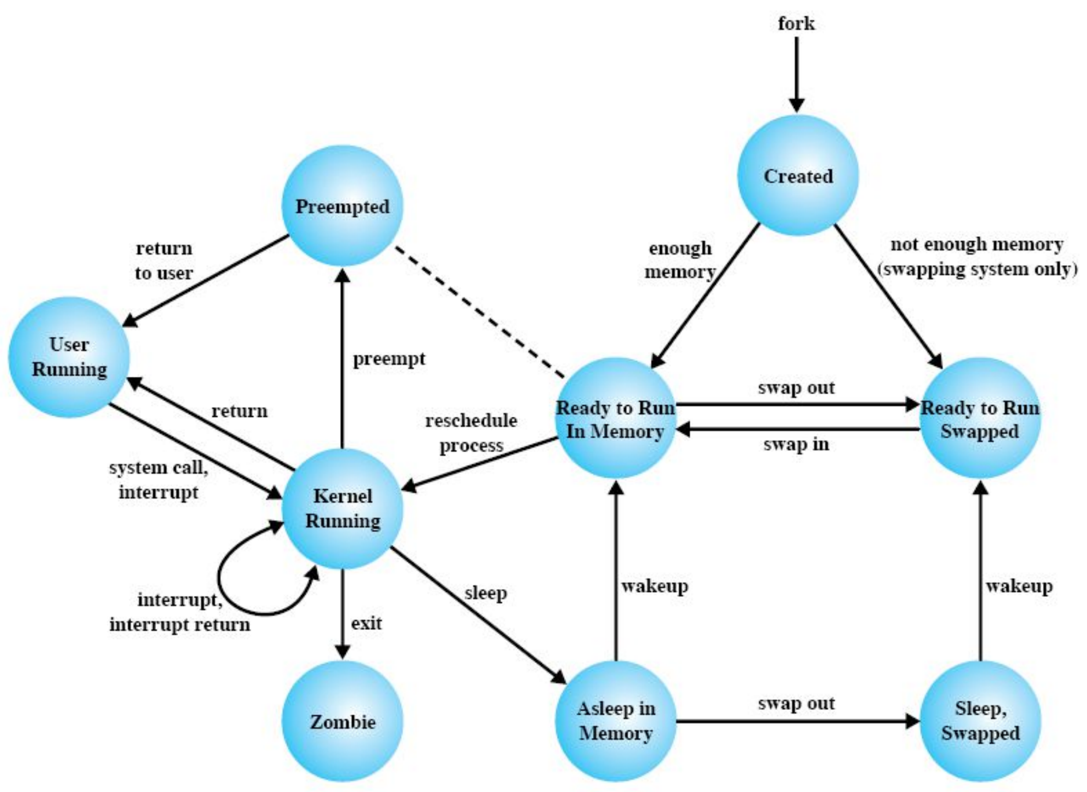
When you first ran the GUI based text editors in the previous chapter you used the & symbol at the end of the command line to fork the command as a new process. It's time now to look at the idea of processes in more detail.

You will have looked at the ideas of concurrency in the first year so the idea that a computer can multi-process or appear to be be running more than one program simultaneously should be familiar to you. You may even remember the idea that a parent process can create child processes (or sub-processes) so that the the set of process running on a given machine is a tree structure with a root which is the first process that starts at boot time.



A process tree

To understand the following exercises, it will also be useful to recall that process can be in a variety of 'states':



The unix process states

4.2 Running stuff, &

We know that simply typing the name of a command causes it to run (no need to practise that) and that adding an & after the command causes the shell to detach the process and allow it to run independently. What happens though if you run a program, forget to add the & and then you want your command prompt back? Here's a useful trick:

4.3 Backgrounding a running job

Run the texteditor:

3. gedit

You'll now have a texteditor window and you can type in it, but if you try and type on the command line – you can't.

4. Try it....

See? What if you NEED that command line back?

At the command line, type ctrl-z – you should get the prompt back and all is well.

You can get on with your work. Wait! Does the text editor still work?

5. Try it. Oh dear! The text editor seems to have gone to sleep. It's in a suspended state. Can we get it to work at the same time as having the command line back? Yes. There are two ways to do this: The easy way is simply to type:

```
bg (short for background)
```

and “Hey Presto!” it'll work again. `bg` puts the last process that you ctrl-z'ed into the executing in the background state. Alternatively you could have done it like this:

6. Run another instance of the text editor:

```
gedit
```

7. ctrl-z it
8. Type:

```
jobs
```

9. …..and it'll give you a list of the jobs you've created. Each job is preceed with a number and its state – you'll see that one job is in the 'stopped' state in this case it should be job 2. To put it into the background type:

```
bg 2
```

4.4 Listing processes with ps

Occasionally you'll want to see what your computer is up to – i.e. what programs are running on it. You can obtain a list of processes with the ps command.

Activity: Get a list of processes

1. Type:

```
ps
```

Unfortunately, that will only show you processes that you have created, so it might be rather a short list at the moment.

2. To see the list of processes created by everybody on the machine, try

```
ps -e  (where -e stand for everbody)
```

One final refinement is to add the -f switch (for full-path):

```
ps -ef
```

Now you can see the full command line that was used to start every process on the machine

Read the man page for ps to find out what the other columns in the output mean.

TO MUCH INFORMATION: Perhaps the output of ps -ef is a little to verbose and goes scrolling off your scren. OK, so you can use the scrollbars on the window to 'go back up', but what if you weren't in a wondowing environemnt – suppose you've just managed to get a shell on something you are penetration testing. How can you limit the amount of stuff you see? Try the piping the output of ps-ef through the less command: `ps -ef | less`

You can use the cursor keys to scroll up and down the output.

4.5 When things go wrong

Linux isn't perfect. And neither are you. Sooner or later one of the two are going to make a mistake. Suppose you have managed to run something and can't stop it – maybe it's crashed and is ignoring ctrl-c maybe you accidental fork-bombed yourself. I did it once on a 500 seat Oracle server installation – I wasn't popular. One way to deal with this is to open another terminal window, find the process number of the rogue process and kill it:

Activity: Terminate with extreme prejudice

1. Make sure you have two terminal windows open (File | new window)
2. In one, run the evil piece of malware known as xeyes
3. xeyes
4. Now we could just close xeyes cleanly or we could ctrl-c it from the terminal instead....
5. Goto your second terminal window:

6. Use ps to find the process number of xeyes

7. Type:

8. kill <process number>

9. Farewell evil spyware.

10. Now some processes are just plain obstinate and don't die when you kill them. In a case like that, there is a more serious form of kill known as kill-9

11. Run xeyes again

12. get its process number

13. Kill it with:

```
kill -9 <process number>
```

14. If that doesn't work you have a major zombie infestation on your hands. You could try killing its parent......

15. Run xeyes again

16. Get the process number of the terminal you ran it from. MAKE SURE YOU GET THE RIGHT ONE!

17. kill that one, NOT THE ONE YOUR TYPING IN.

18. If things have gone really wrong then you may have no choice but to reboot:

19.

```
sudo shutdown -r now
```

-r = reboot. You can try this now if you want or wait to the end of the lesson.

4.6 `xkill` – Killing via graphics

If you have a GUI based program that refuses to die, then xkill is your friend. Run `xkill` from the command line, and click in the window you want to kill.

Activity: xkill

1. Give it a try on xeyes or gedit

4.7 Getting a process to keep running even after you log out - nohup

As you've just seen, killing a parent process usually kills all the children of that process. You may however not want that to happen. A common scenario is that you want a program to keep running even after you've logged out – this can be achieved with the `nohup` (short for no hang-up) command:

```
nohup <your command> &
```

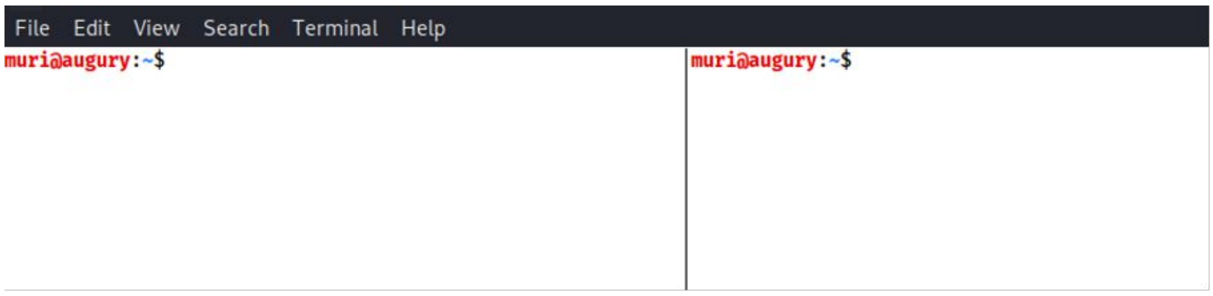
4.8 `screen`

`screen` is a very powerful way to achieve the same kind of thing. With `screen` you can have multiple programs running and switch between them giving each one access to your screen when needed. Screen also allows you to leave processes running on a machine and reconnect to them again next time you login. It's a bit beyond the scope of this document, but I thought I'd better mention it as something you can investigate yourself.

4.9 tmux[2]

Like screen, tmux is a type of program called a "terminal multiplexor". This is a fancy way of saying that it can be used to split the terminal into different sections, but (as with screen), it can also be used to create sessions which can be left to run in the background – an extremely useful trick if you are working on a remote server and need to log out for a bit! As with most other command line programs, tmux can be started by typing "tmux" in the terminal, although there are many other switches available for starting it in different ways or for attaching to backgrounded sessions.

Tmux works on a system of "prefix key bindings". This means that you press a key combination called the "prefix" (by default this is Ctrl + B), then type another character to make the program do something. For example, you could press Ctrl + B, then % to split the terminal along the vertical axis:



```
muri@augury:~$
```

```
muri@augury:~$
```

Or, importantly, you could press Ctrl + B, then the letter "d" to detach from the tmux session and have it run in the background. Once detached from a session, you can list the currently open tmux sessions by typing "tmux ls" in the command line and pressing enter. With a session selected, you can use "tmux a -t SESSION_NAME" to reattach to a session of your choice (substituting in the correct session name).

The default key bindings are widely considered to be rather clunky. Fortunately, tmux is extremely easy to configure. When tmux is started it looks for a file in your current user's home directory called ".tmux.conf". This file contains any custom configurations you have set for the program. The number of available configuration options makes it impractical to go into detail on how this could be done, but a sample configuration file can be found here: https://github.com/MuirlandOracle/linux-config/blob/master/Tmux/.tmux.conf.

As a multiplexor, tmux has three ways to separate your terminal: "sessions", "windows", and "panes". These are in hierarchical order: a session can contain many windows, a window can contain many panes. Sessions are created when you start tmux – they are the overall containers for all your work within tmux. By default, when you start a new session, a single window (containing a single pane) is created. As shown previously, you can split this window into further panes using Ctrl + B then "%" to split the screen vertically, or Ctrl + B then the double quote symbol (") to split the screen horizontally. This is great if you need to see multiple terminals at once! You can move between these panes using Ctrl + B then the arrow keys. New windows can be created by pressing Ctrl + B then the letter "c" – a list of windows in the current session is shown in the bar at the bottom of the screen; to switch between them use Ctrl + B, then the number next to the window (e.g. Ctrl + B then 0).

Tmux is an extremely powerful tool with far more functionality available than is covered in the brief overview provided here. Further investigation into this topic in your own time is highly recommended. A good practical resource for learning more can be found here: https://tryhackme.com/room/rptmux, in addition to a superb YouTube introduction to the topic from IppSec: https://www.youtube.com/watch?v=Lqehvpe_djs.

2 Thanks to Allan Goodwill (CMP209 – 2021) for the section on tmux

4.10 Return codes

Still to do! See this for now: https://shapeshed.com/unix-exit-codes/

5 Redirection, pipes, filters

One thing that Linux excels at is quickly processing information held in text files. One of the means it achieves this by is allowing the output of one program to be easily fed into the input of another through what are known as pipes. In this chapter we'll look at how you can use pipes to rapidly build complex processing structures. Before we use pipes though, it is necessary to understand the concepts of redirection, stdin, stout and stderr.

When you run a command, it looks as though it prints its output on the screen. Actually it doesn't, it sends it to a place called stdout (or standard output). It just happens that stdout is, by default, directed to the screen. You can easily tell a command to send its output somewhere else e.g a file. This is known as redirecting stdout. Let's use the 'ps -ef' command from earlier as an example:

Activity: Redirect to file

1. Type:

```
ps -ef
```

You'll see all the text output on your terminal

2. Now try:

```
ps -ef > processes.txt
```

This time, instead of the output appearing on the screen its been redirected to the file process.txt in your current directory.

3. You can check this by using

```
cat processes.txt
```

or

```
cat processes.txt | less
```

or even

```
gedit processes.txt &
```

HINT: If the file you've used already exists, note that using one > will replace the file with whatever that command output. If you use two >>, it'll add it to the end of the file if it already exists. If It doesn't exist, then it behaves just like a single >

5.1 Stderr

When a linux program outputs an error message, you'll usually see it on the terminal (or possibly in a log file – more on them later). In fact the program has written it to a place called stderr (or standard error) which just happens to be directed to the screen. Again you can redirect stderr to a file and avoid seeing the error messages on the screen - we'll be using this when we come to use the find command.

To redirect stderr, we use the 2> notation e.g.

myprog 2> errors.txt

You may (under some circumstances want to simply throw the error messages away. You can send them to a special file called/dev/null (AKA God's great bit bucket in the sky) and they'll never be seen again

myprog 2> /dev/null

5.2 Connecting process with pipes – the '|' symbol

|Pipes are used to connect the stdout of one program to the stdin of another. This is best learned by example:

Activity: Piping

We're going to produce a complete list of all the words that Shakespeare ever used. There's no good reason for this, it just makes a nice example.

1. Retrieve the complete works of William Shakespeare (let's have a bit of culture in this class!)

```
wget https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt
```

2. Quickly read it all ;-)

```
cat t8.shakespeare.txt
```

Great but we want just a list of words not all that poetry.

If we translate every space in the file into a new line we should get a list of words. We'll use the tr command.

3. Use cat to print the file and pipe the output of cat into the input of tr which does the translation and sends it output to the screen:

```
cat t8.shakespeare.txt | tr '[:space:]' '\n'
```

Ok, we have a list of words. It would be good it they were in some kind of order. Let's sort them (This might take a while).

4. This following command pipeline shows the idea of chaining command together so we have a cat sending its output to tr sending its output to sort

```
cat t8.shakespeare.txt | tr '[:space:]' '\n' | sort
```

You should end up with an alphabetically ordered list of every word that Shakespeare ever used. But wait! (What light from yonder window breaks? 'Tis Juliet lighting her ciggy.) Every occurrence of the word appears multiple times. Wouldn't it be nice it each word only appeared once in our list. There's a filter for that – uniq (short for unique).

5. Let's add another bit to our pipeline

```
cat t8.shakespeare.txt | tr '[:space:]' '\n' | sort | uniq
```

6. And if we wanted, we could save the results of all of that by redirecting the output to a file:

```
cat t8.shakespeare.txt | tr '[:space:]' '\n' | sort | uniq >
wordlist.txt
```

Challenge time. Using the approach above and one command you've already seen in this course answer the following:

7. How many different words did Shakespeare use?

5.3 Using tee to create branches from pipelines

tee takes it's input from stdin and passes it straight to stdout but also to any files named on the command line. It's useful for dumping the results of intermediate processing steps of a long pipeline of commands to disk.

e.g. dump the unsorted Shakespeare to a file:

```
cat t8.shakespeare.txt | tr '[:space:]' '\n' | tee
unsortedShakespeare.txt | sort | uniq > wordlist.txt
```

6 Searching with `find` and `grep`

Being able to search files and the data they contains is a key skill for Linux users be they software developers, digital forensic investigators or pen-testers. In this chapter we'll look at finding files (using the `find` command) and finding things within files (using `grep` and family).

6.1 find

`find` is powerful and has a strange syntax: You need to tell it where to start searching, what to look for, what you want it to do with what it has found and often what to do with any errors. Let's look at a simple example:

```
find . -name “fred.txt” -print
```

The '.' means start looking in the current directory (this directory and all sub-directories will be searched). Instead of '.', you could have typed the full path to any directory where you want to start a search. A common place is '/' which is obviously the root directory which means searching the whole filesystem – useful but it can take a while. There is however a problem with that as some of the directories you'd then be trying to search aren't yours and you won't have permission to read them. It can therefore be useful to run a find starting in '/' as root by prefixing 'sudo' onto the front of the command:

```
sudo find / -name “fred.txt -print
```

This to has a problem, in that is will search in all kinds of special places where although root has permission to search, its doesn't make much sense (the best examples are the /dev and /proc directories – more of which, later). Find will complain loudly by dumping reams of error messages into your output obscurnig the results that you relly want to see. The best way around this is simply to redirect the error strema (stderr) to /dev/null where it will be ignored.

```
sudo find / -name “fred.txt -print 2> /dev/null
```

The -name switch means search for files by name in which case you have to give it a filename pattern to look for – in this case “fred.txt”. If you only that the file you were looking for was a .jpg file, you could use a wildcard pattern to search for them:

```
find . -name “*.jpg” -print
```

The -print simply means print the name of anything you find.

6.1.1 Find by name

Activity: Simple find

1. Do you have a file called smb.conf anywhere on your system?
2. Where is the file fstab?

6.1.2 Find by name with wildcards

Activity: Find with wildcard

1. Find all the .jpg files on your system.
2. How many are there?

6.1.3 Find by type

The -type switch allows you to specify (in a very limited way) what kind of file you want to search for. Check it's man page and find out what:

```
-type d
```

means.

Activity: Find with -type

1. How many sub-directories are there under the /usr directory?

6.1.4 Find by date and by permission

Check out the -ctime and -perm flags in man.

Activity: Find with -type and -ctime

1. How many executables are there under /usr that were last modified after Dec 31 2016?

6.2 `grep` – the 'global reporter'

The `find` command is great for locating whole files, but what if you don't know which file you need? What if you need to see whether a file contains a particular string? `egrep` is your friend.

Grep takes two parameters, a *regular expression* (or regex) specifying what you are looking for and a filename (which can be a wildcard) e.g.

```
egrep 'Macbeth'  t8.shakespeare.txt
```

It will return all the lines in the file which match the regex. It can be useful to specify the `-i` switch which makes the matching case-insensitive.

Activity: What does Macbeth think of physics?

1. `egrep` for the word 'physic' in the complete works of Shakespeare.

6.2.1 Regular Expressions

(See lecture)

6.2.2 grep, egrep fgrep

There are different 'flavours' of the grep command under Linux. I tend to use egrep (extended-grep) as it supports a more powerful version of regular expressions, but for this tutorial, any of them will do.

Activity: Derive and test regular expressions

Derive (i.e. don't just Google them!) and test regular expressions for:

1. A credit card number
2. An md5 hash
3. A hostname (e.g. www.bbc.co.uk)
4. A URL
5. An email address. (Warning, hard! A 90% solution will do)

6.3 Find on the contents of files

Where we've used grep (or egrep etc.) in the examples above we've always known the file we want to search through. What if you want to find all files that contain a particular search term? We can combine find and grep into a powerful search tool – but the syntax is unusual to say the least:

```
find . -name “*” -exec grep “my search term” {} \; -print 2>/dev/null
```

Roughly translated that means starting the current directory (the '.') find all files that are called anything (the '*') and run grep on each one of them – the '{}' means the current filename.

6.3.1 -exec'ing something else

There's nothing magic about the grep in the last section. The sysntax of find just allows us to run (-exec) any command we like on all files returned by the find. We could have tried:

```
find . -name “*” -exec grep md5sum {} \; -print 2>/dev/null
```

Which would be a good answer to the programming problem set in week 2.

6.4 Bigger Exercise – Find the missing link

Activity - Find the missing link

HTML files implement hyperlinks with the <A> and </A> tags (short for anchor).

1. Use this fact, along with the find and grep commands to list all of the hyperlinks you can find on your system.

ANSWER:

find . -name '*.html' -exec grep -E -o "<H1>.*?</H1>" {} \; -print

6.5 Putting it all in context..

You can ask grep to print out lines **b**efore (with the -B switch) and **a**fter (with the -A switch) the line with the match:

```
egrep -B2 -A5 'MACBETH.*physic '  t8.shakespeare.txt
```

Adding a -n switch will precede the printed lines with line numbers:

```
egrep -B2 -A5 -n 'MACBETH.*physic '  t8.shakespeare.txt
```

6.6 Processing columns of data with awk

`awk` is a full programming language: There a full textbooks just about `awk`. Teaching is all is beyond the scope of this introduction to Linux but there are some handy `awk` 'one-liners' that are useful to know about. Suppose your using the 'ls -l' command to get a directory listing. It might produce something like this:

```
-rw-r--r--@ 1 if  staff   195623 Jan 21  2020 linux-fu.odt
-rw-------@ 1 if  staff   354292 Jan 21  2020 linux-fu.pdf
-rw-r--r--  1 if  staff  5458199 Mar 10  2009 t8.shakespeare.txt
-rw-r--r--  1 if  staff   100528 Feb  5 14:47 tmp.txt
-rw-r--r--  1 if  staff   573881 Jan 23  2019 wordlist.txt
```

Too much information! Suppose all you want are the file owner, the file size and the file name – in other words column 3 of the output, column 5 and column 9. If you pipe the output of '`ls -l`' to `awk`, you can write a one-line program which will be applied to every line of the input:

```
ls -l | awk '{ print $3, $5, $9}'
```

This gives you:

```
if 195623 linux-fu.odt
if 354292 linux-fu.pdf
if 5458199 t8.shakespeare.txt
if 100528 tmp.txt
if 573881 wordlist.txt
```

Much better! And, yes, I know there are ways you can do that kind of thing just with ls.

That showed awk running as a filter in a pipeline of commands. You can also apply it directly to a file:

Activity - Using awk on a file

Create a file containing a list of current processes:

```
ps -ef> procList.txt
```

Take a look at the file:

```
cat procList.txt
```

Too much information – I only want to see the PID (column 2) and the command used to run the process (column 8)

```
awk '{ print $2, $8}' procList.txt
```

6.7 Capturing a session with `script`

You can use the script command to record a terminal session:

Activity - Logging interactivity with script

To start logging simple type

```
script 2021-02-05-session01.txt
```

All input and output in the that terminal will not be recorded in the file 2021-02-05-session01.txt until you type

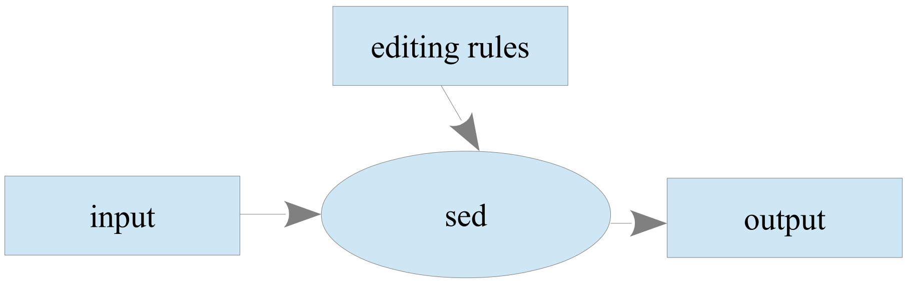
```
exit
```

to stop the recording.

Script is ideal for recording what you are doing during a Digital Forensic investigation. It provides a great way of recording the necessary detail of what's going on.

6.8 Stream editing - sed

A stream editor is a text-editing program that reads input a line at a time from its stdin, makes some change to that input specified by some editing rules and then writes the output to stdout.



Activity - Stream editing with sed

create an input file called input.txt:

```
I like cheese:
cheese sandwiches
cheese on toast with added cheese
```

cheese rolls

Bored with cheese – want sausage!

```
cat input.txt | sed s/cheese/sausage/
```

the output is

```
I like sausage:
sausage sandwiches
sausage on toast with added cheese
sausage rolls
```

the `s/cheese/sausage/` bit of command is the editing rule. It tells sed to substitute (s) whatever is between the first two '/'s (in this case cheese) with whatever it between the second pair (sausage).

Looks good. But wait!!!! There's still some cheese in there. I don't want cheese on my sausage (…......hmn...don't I?) sed only changes the *first* occurrence of the search string unless the sed editing rule is changed to:

```
s/cheese/sausage/g
```

(notice the added 'g' for global) The output will now be

```
I like sausage:
sausage sandwiches
sausage on toast with added sausage
sausage rolls
```

What if you want to make multiple edits? You can place multiple editing rule in a file (e.g. edits.txt) and have sed apply them all

```
cat input.txt | sed – f edits.txt
```

You can, of course use redirection to write the results to a file:

```
cat input.txt | sed – f edits.txt > output.txt
```

7 Devices and Filesystems

Mounting, mount points

loopback

- usb devices
- automount

/mnt

/dev

umount

fdisk

/dev/null

/dev/zero

/dev/rnd

8 Hashing

Md5

md5sum

md5deep

sha256 and friends

9 Scripting

Bash

echo

#!

variable assignment

printing

echo

control structures

if

formattingforeach

while

exists

10 Installing software

Apt

synaptic

repositories

add

remove

which packages are installed

which package does a file belong to

11 Networking

Old style

apps telnet, ftp, wget

utils – ping, whois ,finger, nslookup

Modern

ssh

scp, sftp

sshfs

netstat

nmap

tcpdump, tshark

smb

12 Scheduling

At,cron

13 Compiling

Gc, make,config, autoconf

build-essentials

library files/packages

cross compiling

14 Configuration

/etc

fstab

/var

/log

15 References

1. Baron, R. M. F. (2002) ‘A Critique of the International Cybercrime Treaty’, *Journal of Communications Law and Technology Policy*, 10, pp.263-278