Introduction to the Unix Command Line

Bob Dowling `rjd4@cam.ac.uk`

Julian King `jpk28@cam.ac.uk`

13 October 2010

Copies of these notes may be found at
`http://www-uxsup.csx.cam.ac.uk/courses/UnixCLI/`

Table of Contents

Notation

Warnings

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

[5 minutes]

Exercise 0.
Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.
An indication is given as to how long we expect the exercise to take. Do not panic if you take longer than this. If you are stuck, ask a demonstrator.
Exercises marked with an asterisk (*) are optional and you should only do these if you have time.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: “`Lorem ipsum`”. (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the “A key” will be written “[A]”. Note that the return key (pressed at the end of every line of commands) is written “[↲]”, the shift key as “[⇧]”, and the tab key as “[↹]”. Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as “[⇧]+[A]”. Note that pressing [A] generates the lower case letter “a”. To get the upper case letter “A” you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                  This is a comment about the line.
```

1 The example text here is the famous “lorem ipsum” dummy text used in the printing and typesetting industry. It dates back to the 1500s. See `http://www.lipsum.com/` for more information.

Booting & logging in

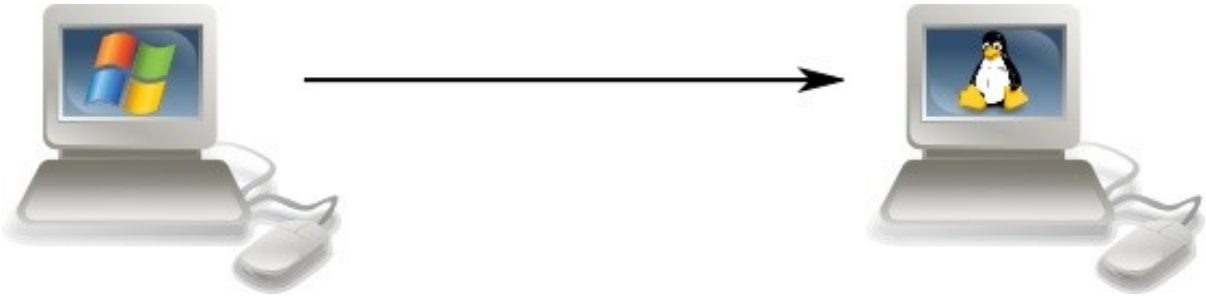
“Booting” is the name given to the process of turning a dead chunk of metal, plastic and silicon into an activated, functioning computer ready to help you with your work. It takes its name from “bootstrapping”, the magical ability to lift yourself up by your own boot straps.

In the simplest case a computer will boot into its one and only operating system, be it Microsoft Windows™

or Linux™:

Some computers, such as those in this classroom, can be set up so that they can boot into either Windows or Linux, with the choice being made by the user at boot time.

If the computer is already running Windows then we need to reboot from Windows to Linux.



Rebooting from Windows to Linux on the PWF

1. Log out from Windows to return to the login window.
2. Click the “Shutdown” button.
3. Select “Shutdown and restart” from the menu offered.
4. As the system boots you will be offered a menu of PWF Windows and PWF Linux.
5. Use the down arrow [↓] to select PWF Linux.
6. Press Return, [↲].
7. Once the booting is complete you will see the RIPA[2] message. You should read this once, but don't need to every time you log in.
8. Click the [OK] button to continue.
9. You will then be asked for your user name. This is also known as your login ID, your user ID or, in Cambridge, your CRS ID[3]. In the course you will be using IDs of the form ynnn. The demonstrator will be using y250; you should each have your own individual ID.
10. Enter your ID and press return [↲].
11. You will be asked for your password. For this course you have all been given the same (not very good) password. Enter it and press [↲] again. (For your own account you should have a better password and should not share it with anybody else.)
12. After about 15 seconds you should be logged in and should see the “message of the day” window in the middle of your screen.

Exercise 1.
Reboot into Linux and log in with the course id you have been given.

[5 minutes]

2 Regulation of Interception Powers Act
3 CRS: Common Registration Scheme. This is how we keep the various IDs in sync across the computers in Cambridge.

Terminal windows and text consoles

To launch a text console in PWF Linux, select Applications→ Unix Shell→ Gnome Terminal. This can be done as often as you want for as many text consoles as you want. Each runs an independent command line interpreter (“shell”).

Alternatively you can dispense with the graphical environment entirely and go back to the “good old days” of text-mode Unix. Press [Ctrl]+[Alt]+[F2] to get a pure text login console. If we enter our PWF login and password again we can log in here too. Note that we can be logged in both through the graphical interface and the text interface(s) simultaneously. In fact, we could be logged in to one interface and our neighbour could log in through another. Unix is a fully multi-user operating system. Identical interfaces are available by using [F3], [F4], [F5], or [F6] in place of [F2]. [F1] has a text console that tends to be more colourful.

When you switch between consoles there is often a several second black screen as the console switches over. Please be patient and let the switch complete before you start switching back again. If you try to switch while the consoles are in mid-switch you can jam things badly.

Using [Ctrl]+[Alt]+[F7] returns you to the graphical interface.

You will be faced with a set of text that looks like this in the graphical terminal window or the plain text console:

```
pcphxtr01:~$
```

The text at the start of the line is called the “prompt” and its purpose is to prompt you to enter some commands.

The prompt can be changed (see below) but the default prompts on PWF Linux have these components:

`pcphxtr01`	Your computer
`:`	separator
`~`	The directory your session is “in”, also known as the “current working directory”. “~” is shell short hand for “your home directory”.
`$`	Final separator.

To issue a command at the prompt simply type the name of the command and press the Return key, [↵]. For example, the `ls` command lists the files in the current working directory:

```
pcphxtr01:~$ ls
Appscfg.PWF  Desktop  Library  My Music  My Pictures  My Video  Unix Intro

pcphxtr01:~$
```

Note that the prompt is repeated after the `ls` command has completed.

Logging out

Once we are finished with a terminal or a terminal window we need to quit. We will illustrate three ways to do this.

Close the window

In the graphical environment the terminal window is just another window. At its top right corner are the three buttons for minimising, maximising and closing. If you click in the [×] button the window is closed and the session cleanly ended.

The exit command

In either a terminal window or a text console you can issue the command “`exit`”; this will end the session. In the graphical environment ending the session running in a window closes the window too[4]. In a text console

4 This is the case on PWF Linux and most traditional Unixes. On MacOS X the terminal window is left open

the console is typically cleared and a fresh login prompt presented.

[Ctrl]+[D]

Recall that "[Ctrl]+[D]" means to press down the [Ctrl] key at the same time as the [D] key. In practice we press the [Ctrl] key down, press and release the [D] key, and then release the [Ctrl] key.

On a Unix system [Ctrl]+[D] means "end of input". We will meet it later when we are entering data nto a command and want to signal that we have finished. Here it signals to the shell that we have no more input for it so it might as well quit. And quit it does.

[5 minutes]

Exercise 2.

1. Log in to the graphical interface if you are not already logged in.
2. Start up two terminal windows. (Recall: Applications→ Unix Shell→ Gnome Terminal)
3. Switch to the [Ctrl]+[Alt]+[F2] console and log in there too.
4. Repeat for the [Ctrl]+[Alt]+[F3] console.
5. Run the command "w". This shows who is logged in and where. You should get something like this:

```
pcphxtr01:~$ w
 10:20:00 up 16 min,  5 users,  load average: 0.01, 0.08, 0.16
USER  TTY   LOGIN@  IDLE   JCPU   PCPU   WHAT
y250  tty2  10:10   10:00  0.05s  0.05s  -bash
y250  tty3  10:10   0.00s  0.07s  0.00s  w
y250  tty7  10:06   10:07  3.39s  0.47s  /usr/bin/gnome-session
y250  pts/0 10:09   10:27  0.02s  0.02s  bash
y250  pts/1 10:09   10:20  0.04s  0.04s  bash
pcphxtr01:~$
```

The user column will show your ID rather than `y250`, of course.
Log out of one of the text consoles when you are done.

It is easy to forget to log out of sessions that aren't right in front of your eyes. Logging out of the graphical interface will not log you out of any of the text interfaces.

Just for interest

The letters "`tty`" in `w`'s output stand for "**t**ele**ty**pe". This was the name of the old, clunky paper printing terminals that used to be plugged into the backs of the old mainframe computers. Today we don't have that set up but instead we have multiple teletypes all provided through the same console and switched between using the [Ctrl]+[Alt]+[F*n*] key combinations.

The teletype "tty**2**" corresponds to [Ctrl]+[Alt]+[F**2**], and "tty**3**" to [Ctrl]+[Alt]+[F**3**]. The seventh teletype (tty**7**) is dedicated to the graphical interface managed by a program called `gnome-session`. This is why you use [Ctrl]+[Alt]+[F**7**] to return to the graphical environment.

The graphical terminal windows are not "real" teletypes. These are managed by a "**p**seudo-**t**erminal **s**ervice", or "`pts`" for short. The first two terminal windows you create are assigned pts/0 and pts/1 respectively.

[5 minutes]

Exercise 3.

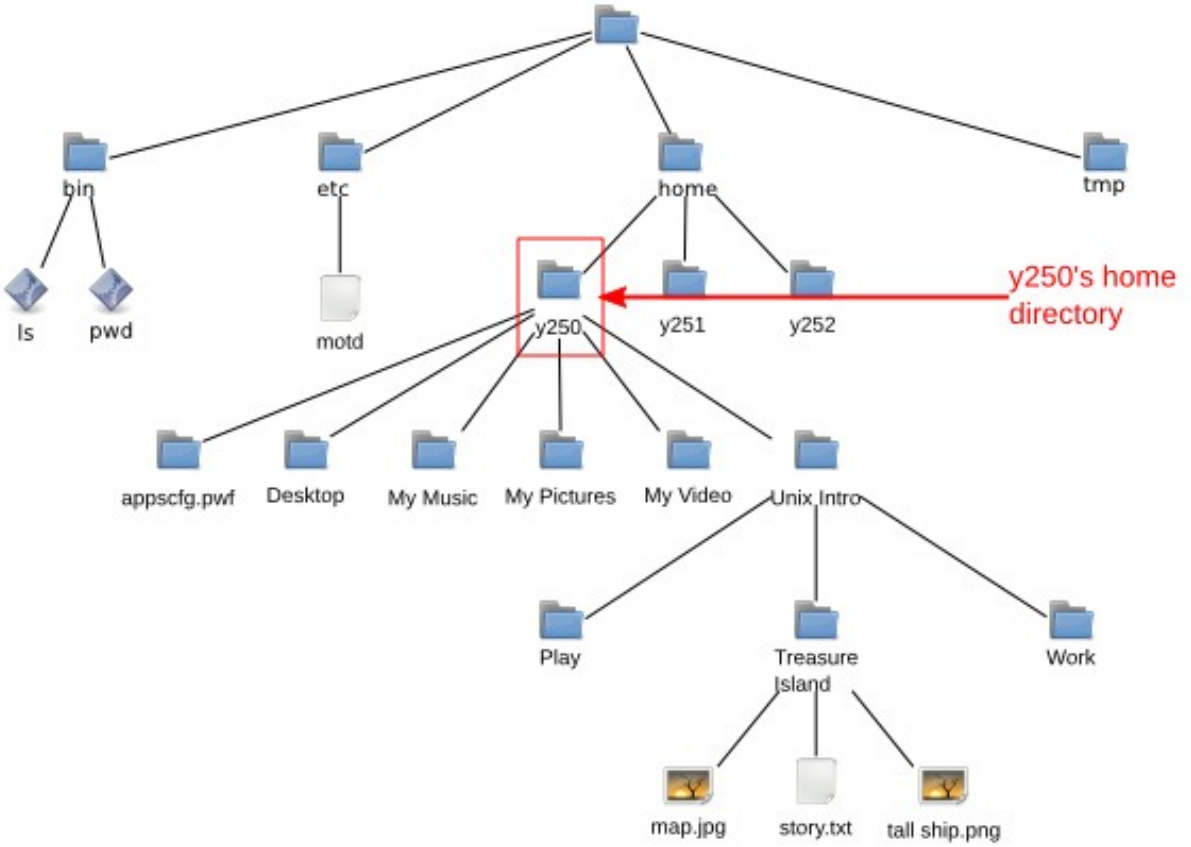
The `w` command summarizes various bits of data which can also be seen individually. Try the "`who`" and "`uptime`" commands as well.

and the window needs to be closed explicitly.

Navigating the file system in the CLI

All the elements of a Linux system form a hierarchy called the “file system”. This hierarchy is a tree of folders (typically called “directories” in the Unix world) and files containing content. (Actually, there are other types of thing in the file system but we don't need to worry about them here.)

A subset of that hierarchy belongs to you and takes the form of a subset of the hierarchy hanging off your “home directory”. The various logins you have done so far all start you off in your home directory. This is the standard place to start. If you go looking for files or creating them then the path to them starts here.



Directories

Working directory

To know what directory we are in we use the command `pwd`, “print working directory”:

```
pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$
```

Directory contents

To see what is in the current directory we use the `ls` (“**ls**t”) command:

```
pcphxtr01:~$ ls
Appscfg.PWF  Desktop  My Music  My Pictures  My Video  Unix Intro

pcphxtr01:~$
```

These are all directories. The `ls` on PWF Linux uses colours to indicate types of files, and directories are coloured blue. Also note that with the exception of "`Desktop`" all the names of directories have spaces in them.

We can get more information from `ls` about the contents of the directory by asking for its "long" output. We do this by adding an option, "`-l`" (for "**l**ong"), to the command. Note that there is a space between the "`ls`" and the "`-l`":

```
pcphxtr01:~$ ls␣-l
total 3
drwxr-x--- 1 y250 y250 512 2009-04-28 13:11 Appscfg.PWF
drwxr-xr-x 1 y250 y250 512 2009-04-28 12:51 Desktop
drwxr-x--- 1 y250 y250 512 2009-04-28 13:11 My Music
drwxr-x--- 1 y250 y250 512 2009-04-28 13:11 My Pictures
drwxr-x--- 1 y250 y250 512 2009-04-28 13:11 My Video
drwxr-xr-x 1 y250 y250 512 2009-04-28 13:22 Unix Intro

pcphxtr01:~$
```

(We will not usually put in spaces explicitly as "␣" in future.)

We can analyse the output of `ls -l` by looking at the `Desktop` line in detail. It will be easier to explain if we work through it right to left, though.

`Desktop`	This is the name of the file or directory.
`2009-04-24 11:37`	This is the date and time of the last update to the file, or the date it was created if it has not been updated since. The format of this time stamp varies between Linux distributions; some spell out the date less numerically.
`512`	This is the number of bytes taken by the file. Directories have a database structure within them and their sizes are typically multiples of 512 bytes. General files will have arbitrary sizes.
`y250 y250`	The first instance of `y250` is the owner of the file. This is typically the user who created the file (or for whom it was created in this case). Users can be lumped together into groups and on PWF Linux we place each user into a group of their own. The second `y250` is the group associated with the file. In this specific case the user `y250` is the only member of the group `y250`. On some Unix systems all the users are placed in a group called "`users`" and that group is used.
`1`	A property of the Unix file system is that more than one name can correspond to the same file. This number, called the "reference count", is the number of names that correspond to this file or directory. In our simple case our files all have just one name.
`rwxr-xr-x`	These are the permissions on the file or directory. They form three triplets, each a letter or a dash. The letter shows the permission is granted and the dash that it is denied. We can consider the permissions one triple at a time:
`rwx`	The first triple identify the permissions granted to the owner of the file (user `y250`). The user may **r**ead from, **w**rite to and e**x**ecute the file. Read and write permissions mean exactly what they say. "Execute" permission means that the owner may run this file as a program. In the case of a directory, the read permission means that the owner may look to see the names of the files within the directory, the write permission means that the owner may add or remove things in the directory, and the execute permission means that the owner can change directory into it.
`r-x`	The second triple indicates the permissions granted to members of the group who are not the owner. In the case of PWF Linux there is no one who matches this description. The middle dash means that the write right is not granted.
`r-x`	The third triple indicates the permissions granted to any user who is not in the group or the owner of the file.
`d`	The first character indicates that this is a directory. If it was a plain file the symbol would be a dash.

Changing directory

Now we know we have some directories, let's use one of them, "`Unix Intro`". The command to **c**hange **d**irectory is `cd`. But things don't go quite as planned:

```
pcphxtr01:~$ cd Unix Intro
-bash: cd: Unix: No such file or directory

pcphxtr01:~$
```

What has gone wrong here is that the shell is using the space character to split up the various bits of the command line just as it split the command `ls` from its option `-l`. So here the `cd` command is being given two words to work on (called "arguments" in the jargon). It's only expecting one, the directory to change to, and ignores the second. It tries to change to a directory called "`Unix`" and fails because no such directory exists.

The error message can be understood as a series of reports:

`bash`	The error came from the shell (`bash`).
`cd`	`bash` was running `cd`.
`Unix`	`cd` had a problem with the `Unix` directory.
`No such file or directory`	The problem was that it didn't exist.

There are two "proper" ways round this and one neat trick that will avoid the problem for ever.

The proper ways involve informing the shell that the space between "`Unix`" and "`Intro`" is not the same as the space between "`cd`" and "`Unix`".

Quoting

The first way to do this is to place the name of the directory in quotes. The quotes tell the shell to treat everything inside them as a single word and not to split it up on spaces.

```
pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$ cd "Unix Intro"

pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro
```

Now that we've completed that demonstration we need to know how to get back to our home directory. If you run the command `cd` with no argument then it defaults to your home directory:

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd

pcphxtr01:~$ pwd
/home/y250
```

Also note how the prompt changes to indicate the current directory too.

Escaping

The second approach is to specifically identify the space character as "nothing special, just another character" so the shell doesn't use it for splitting. We do this by preceding the space with a backslash character, "\". This is called "escaping" the character:

```
pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$ cd Unix\ Intro

pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd

pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$
```

And now the good news: you will never have to remember to type the quotes or backslash again. In fact you're about to do a lot less typing.

File name completion

Let's start again in the home directory. This time we will start to type the name of our double-barrelled directory, "`Unix Intro`", but only get as far as the first letter. Then we hit the Tab key, [↹].

```
pcphxtr01:~$ cd U↹
```

At this point the shell determines that there is only one file or directory present that starts with a "`U`" and automatically extends the file name on the command line, adding any escaping that is necessary:

```
pcphxtr01:~$ cd Unix\ Intro/

pcphxtr01:Unix Intro$
```

In the case of completion of a directory name it even puts on a terminal slash, "`/`", in case you want to continue the quoting of a file within that directory. A trailing slash makes no difference to us so we leave it.

It's not always possible for the shell to determine exactly what file or directory you want. We can see this if we return to the home directory and look at some of the other directories:

```
pcphxtr01:Unix Intro$ cd

pcphxtr01:~$ ls
Appscfg.PWF  Desktop  My Music  My Pictures  My Video  Unix Intro
```

This time we type "`cd M`" and then press [↹].

```
pcphxtr01:~$ cd M↹
```

There are three directories that start with an "`M`". The shell extends the name as far as it can and beeps to let you know that it needs help to go further:

```
pcphxtr01:~$ cd My\␣
```

(We have explicitly marked the space it has added with a "␣".)

We then press one more letter to distinguish between "`My Music`", "`My Pictures`" and "`My Videos`" (an "`M`", a "`P`", or a "`V`") and press [↹] again:

```
pcphxtr01:~$ cd My\␣P↹
```

and the shell can now complete the directory name:

```
pcphxtr01:~$ cd My\␣Pictures/
```

The use of the Tab key and its automatic escaping of file names will save us a lot of typing and also having to remember the backslashes or quotes.

Directories again

Let's return to the `Unix Intro` directory and take another look at directory navigation. We have seen how to enter a directory and to return to our home directory. How can we move up the directory tree from our current location? The answer lies in a couple of “hidden” directories in every directory. We can see hidden files and directories with another option to `ls`, “`-a`” (for “**a**ll”):

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ ls
Play  Treasure Island  Work

pcphxtr01:Unix Intro$ ls -a
.  ..  Play  Treasure Island  Work

pcphxtr01:Unix Intro$ ls -la
total 3
drwxr-xr-x 1 y250 y250 512 2009-04-28 12:07 .
drwxr-xr-x 1 y250 y250 512 2009-04-28 13:27 ..
drwxr-xr-x 1 y250 y250 512 2009-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2009-04-27 19:06 Treasure Island
drwxr-xr-x 1 y250 y250 512 2009-04-27 19:19 Work

pcphxtr01:Unix Intro$
```

The `ls` options `-l` and `-a` can be combined as “`ls -al`”, “`ls -a -l`”, “`ls -la`”, or “`ls -l -a`”. They are all equivalent.

Every Unix directory has the two entries “`.`” and “`..`”; they are created automatically and you can’t remove them. The first, “`.`”, refers to the directory itself, so “`cd .`” has no effect:

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd .

pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$
```

The second, “`..`”, refers to the parent directory. Changing directory to “`..`” takes you up one level in the directory tree:

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd ..

pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$
```

We have just been dipping in and out of one directory directly beneath our home directory. As a result, plain “`cd`” is enough to return us to where we were before. Alternatively we have been able to use “`cd ..`” to get back.

More generally, “`cd -`” will take us back to our previous directory:

```
pcphxtr01:~$ cd Unix Intro

pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd -

pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$ cd -

pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ cd -

pcphxtr01:~$ pwd
/home/y250

pcphxtr01:~$
```

File paths

We don't need to enter a directory to see what's in it. We don't need to only change one level of directory tree at a time either.

We will start in our home directory. We want to run `ls` on the Treasure Island directory in the "`Unix Intro`" directory. We refer to the directory within another directory by separating their names with a slash, "`/`". Note that there are no spaces around the slash:

```
pcphxtr01:~$ ls Unix\ Intro/Treasure\ Island/
map.jpg  story.txt  tall ship.png

pcphxtr01:~$
```

(Also note how few keys had to be hit to enter that command: "`ls␣U↹T↹↵`" . Tab completion is your friend.)

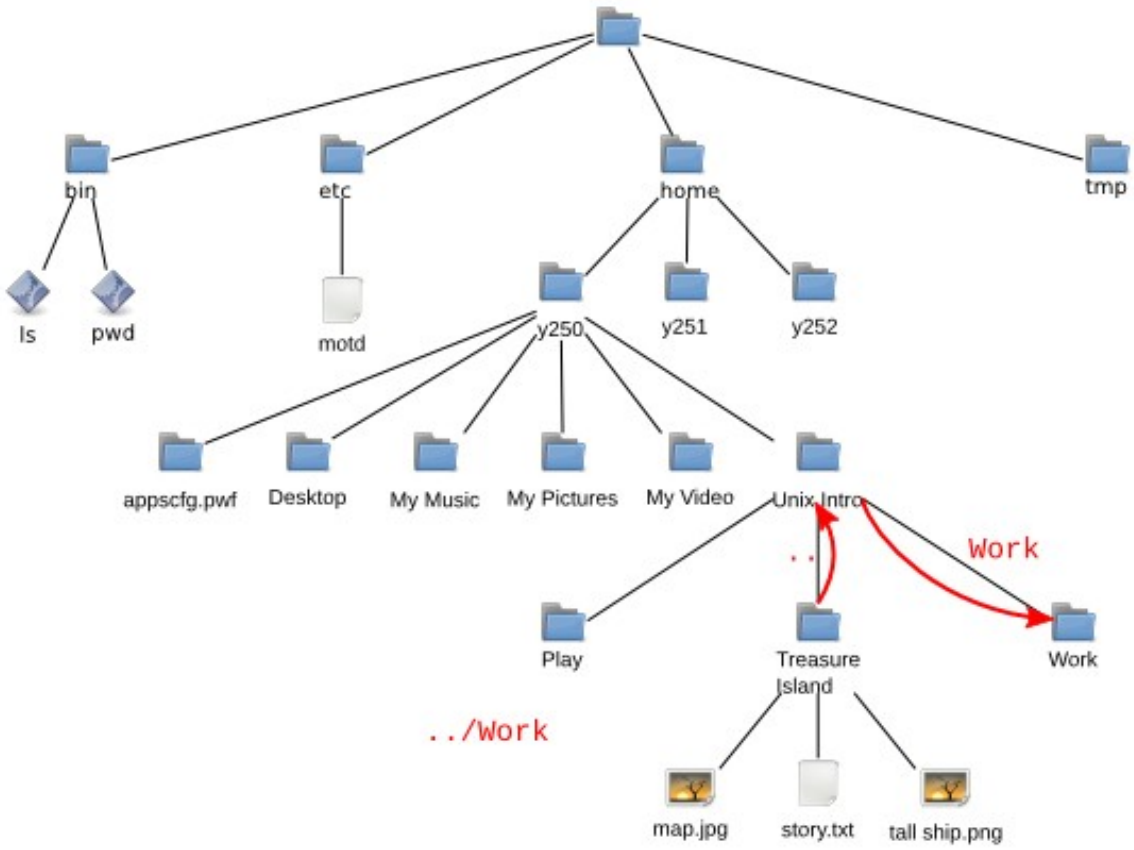
We can also change directory directly rather than having to pass through the intermediate directory:

```
pcphxtr01:~$ cd Unix\ Intro/Treasure\ Island/

pcphxtr01:Treasure Island$
```

We can also use "`..`" in these file paths:

```
pcphxtr01:Treasure Island$ pwd
/home/y250/Unix Intro/Treasure Island

pcphxtr01:Treasure Island$ ls -l ../Work/
total 10
-rw-r--r-- 1 y250 y250   36 2009-04-27 19:12 abc.txt
-rw-r--r-- 1 y250 y250   36 2009-04-27 19:13 def.txt
-rw-r--r-- 1 y250 y250   36 2009-04-27 19:13 ghi.txt
-rw-r--r-- 1 y250 y250 3664 2009-04-27 19:14 lorem.txt
-rw-r--r-- 1 y250 y250 3664 2009-04-27 19:15 nonsense.txt
drwxr-xr-x 1 y250 y250  512 2009-04-27 19:18 Project Alpha
```

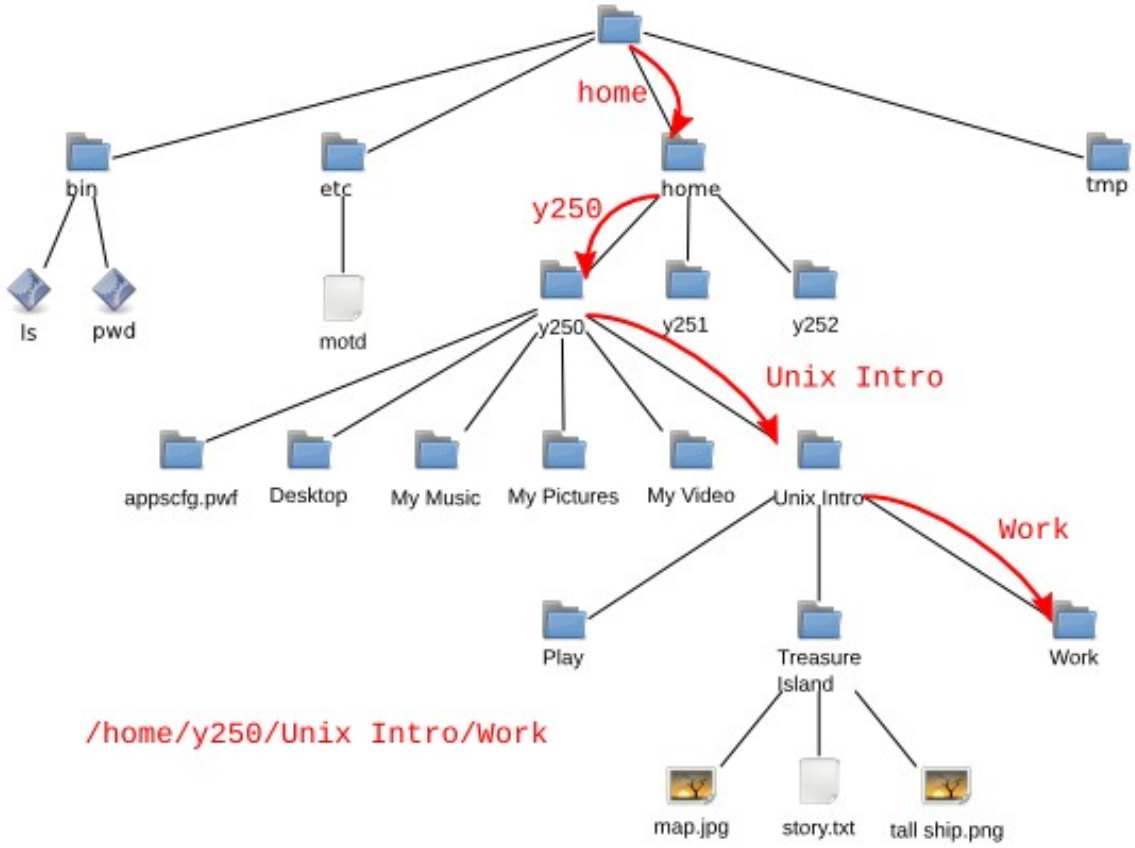
In this last case "`..`" has taken us up one level and then "`Work`" has taken us back down again into another directory:



Note that a *forward* slash, "`/`" is used in file paths to separate components and a *back* slash, "`\`", is used to escape spaces in commands. They are *not* the same.

Our file paths so far have all started with a component in the current working directory. (Recall that "`..`" is technically within the directory even though it points up a level.) These are called "*relative* paths" and refer to files and directories relative to the current working directory.

If a file path starts with a forward slash, "`/`", then it is evaluated relative to the top (the "root") of the file tree. This is called an "*absolute* path".



So if our current working directory is "`Treasure Island`" then we can refer to the "`Work`" directory either as "`../Work`" (relative path) or as "`/home/y250/Unix\ Intro/Work`" (absolute path).

[10 minutes]

Exercise 4.

1. Run the "`cd`" command on its own to start in your home directory.
2. Run "`pwd`" to check where you are.
3. In a *single* `cd` command, using a relative path, change directory to the `Work` subdirectory of the `Unix Intro` directory.
4. Run "`pwd`" to check where you are.
5. Starting from the `Work` directory, move into the `Play` directory with a single `cd` command and a relative path.
6. What do you think the absolute path is for your current directory?
7. Run "`pwd`". Were you right?

Hint: Remember the various ways to allow for the space in `Unix Intro`.

Renaming, creating and deleting file and directories

Now we can manoeuvre around the file system, we need to know how to manipulate it. We will start in the "`Treasure Island`" directory.

```
pcphxtr01:Work$ pwd
/home/y250/Unix Intro/Treasure Island

pcphxtr01:Treasure Island$ ls
map.jpg  story.txt  tall ship.png

pcphxtr01:Treasure Island$
```

Renaming and moving items

Suppose we want to rename the file "`tall ship.png`" to "`hispaniola.png`"[5]. We do this with the `mv` ("**m**o**v**e") command. Note the use of tab completion to avoid having to explicitly escape the space in the file's name.

```
pcphxtr01:Treasure Island$ ls
map.jpg  story.txt  tall ship.png

pcphxtr01:Treasure Island$ mv tall\ ship.png hispaniola.png

pcphxtr01:Treasure Island$ ls
hispaniola.png  map.jpg  story.txt
```

The `mv` command's name is more obvious when used to move a file into another directory:

```
pcphxtr01:Treasure Island$ mv story.txt ..

pcphxtr01:Treasure Island$ ls
hispaniola.png  map.jpg

pcphxtr01:Treasure Island$ ls ..
Play  story.txt  Treasure Island  Work

pcphxtr01:Treasure Island$
```

We can move files between directories and rename them simultaneously:

```
pcphxtr01:Treasure Island$ mv map.jpg ../island.jpeg

pcphxtr01:Treasure Island$ ls
hispaniola.png

pcphxtr01:Treasure Island$ ls ..
island.jpeg  Play  story.txt  Treasure Island  Work

pcphxtr01:Treasure Island$
```

We can use `mv` on directories just as we do for files within our home directory. (Things can get more complicated if you want to move directories to other parts of the system.)

Copying files

To copy a file we use the command `cp` ("**c**o**p**y") just like we used `mv`:

```
pcphxtr01:Treasure Island$ ls
hispaniola.png

pcphxtr01:Treasure Island$ cp hispaniola.png "tall ship.png"

pcphxtr01:Treasure Island$ ls
hispaniola.png  tall ship.png

pcphxtr01:Treasure Island$
```

Note how we still have to use quotes (or backslashes) when describing new files. They don't exist yet so tab completion can't find them.

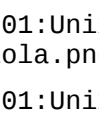
There is a slight wrinkle with using `cp`; it cannot be used to copy directories without an extra option. The option is "`-R`" ("**r**ecursive")[6] which means to copy the directory and everything in it:

5 "Hispaniola" was the ship that took Jim Hawkins, Long John Silver *et al* to Treasure Island.

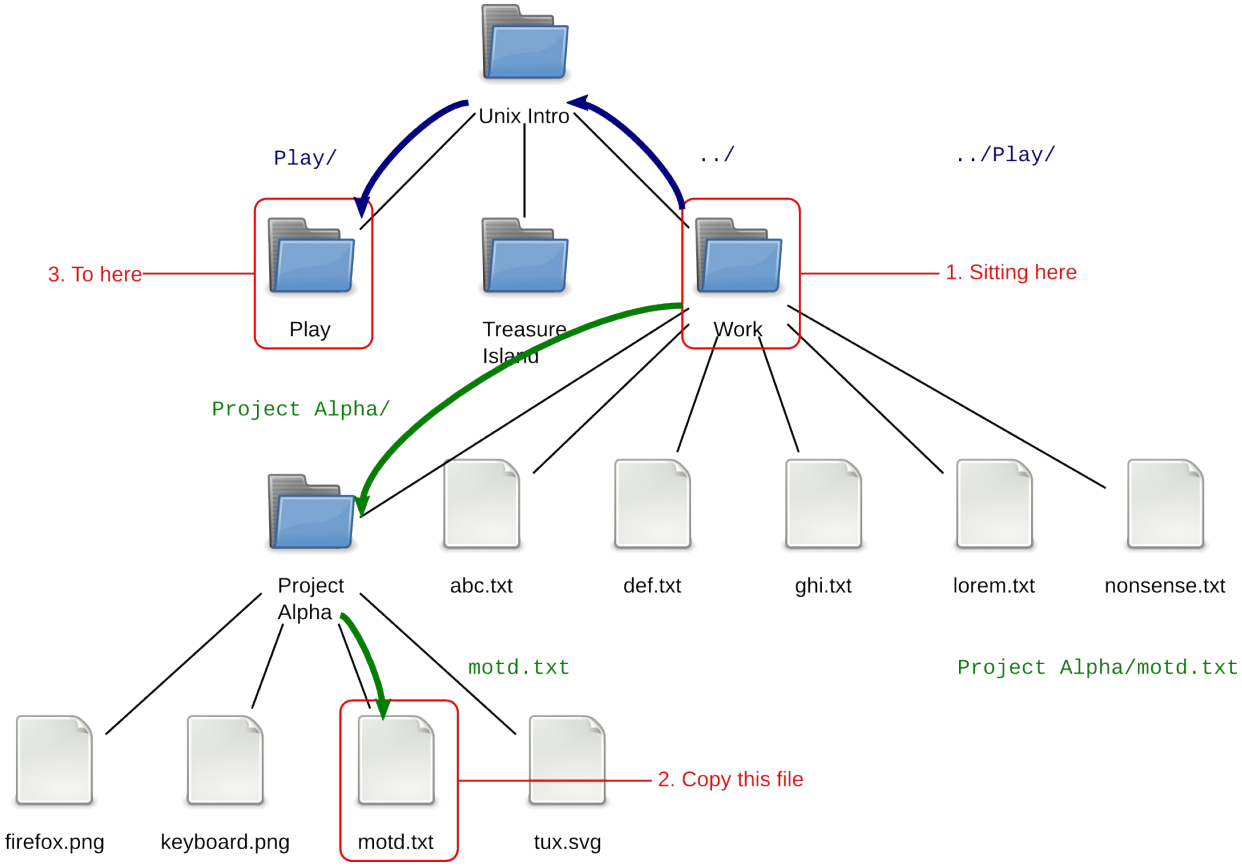
6 On some older versions of Unix this was a lower case letter, "`-r`". However, for consistency between commands capable of recursive behaviour, modern versions have standardised on the upper case "`-R`".

```
pcphxtr01:Treasure Island$ cd ..

pcphxtr01:Unix Intro$ cp Treasure\ Island/ "Copy of Treasure Island"
cp: omitting directory `Treasure Island/'

pcphxtr01:Unix Intro$ cp -R Treasure\ Island/ "Copy of Treasure Island"

pcphxtr01:Unix Intro$ ls Copy\ of\ Treasure\ Island/
hispaniola.png  tall ship.png

pcphxtr01:Unix Intro$
```

The copy and move commands, `cp` and `mv`, take two arguments[7]: the source and the destination. Each of these is evaluated relative to the current working directory. The destination is *not* evaluated relative to the source.



Suppose we were in the `Unix Intro/Work` directory (i.e. that was our current working directory) and we wanted to copy the `motd.txt` file in `Project Alpha` into the `Play` directory then we would refer to the `motd.txt` file as `Project\ Alpha/motd.txt` and its destination location as `../Play/motd.txt`.




```
pcphxtr01:Work$ pwd
/home/y250/Unix Intro/Work

pcphxtr01:Work$ cp Project\ Alpha/motd.txt ../Play/motd.txt

pcphxtr01:Work$
```

7 Actually, they can take more but we'll consider the simple case for now.

Creating directories

To create an empty new directory we use the command `mkdir` ("**m**a**k**e **dir**ectory"):

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ ls -l
total 428
drwxr-xr-x 1 y250 y250    512 2009-04-28 18:07 Copy of Treasure Island
-rw-r--r-- 1 y250 y250  44928 2008-08-21 18:53 island.jpeg
drwxr-xr-x 1 y250 y250    512 2009-04-23 18:20 Play
-rw-r--r-- 1 y250 y250 390927 2009-04-24 11:38 story.txt
drwxr-xr-x 1 y250 y250    512 2009-04-28 18:06 Treasure Island
drwxr-xr-x 1 y250 y250    512 2009-04-27 19:19 Work

pcphxtr01:Unix Intro$ mkdir Fun

pcphxtr01:Unix Intro$ ls -l
total 429
drwxr-xr-x 1 y250 y250    512 2009-04-28 18:07 Copy of Treasure Island
drwxr-xr-x 1 y250 y250    512 2009-04-28 18:13 Fun
-rw-r--r-- 1 y250 y250  44928 2008-08-21 18:53 island.jpeg
drwxr-xr-x 1 y250 y250    512 2009-04-23 18:20 Play
-rw-r--r-- 1 y250 y250 390927 2009-04-24 11:38 story.txt
drwxr-xr-x 1 y250 y250    512 2009-04-28 18:06 Treasure Island
drwxr-xr-x 1 y250 y250    512 2009-04-27 19:19 Work

pcphxtr01:Unix Intro$ ls -l Fun
total 0

pcphxtr01:Unix Intro$
```

Removing files and directories

To remove a file we can use the `rm` ("r**m**ove") command:

```
pcphxtr01:Unix Intro$ rm island.jpeg

pcphxtr01:Unix Intro$ ls
Copy of Treasure Island  Fun  Play  story.txt  Treasure Island  Work
```

Command line removal with `rm` is for ever. There is no "waste basket" to recover files from.

Empty directories can be removed with the `rmdir` ("**r**e**m**ove **dir**ectory") command, but directories with content cannot:

```
pcphxtr01:Unix Intro$ rmdir Fun/

pcphxtr01:Unix Intro$ ls
Copy of Treasure Island  Play  story.txt  Treasure Island  Work

pcphxtr01:Unix Intro$ rmdir Copy\ of\ Treasure\ Island/
rmdir: failed to remove `Copy of Treasure Island/': Directory not empty

pcphxtr01:Unix Intro$
```

If you do want to remove a directory and everything within it you need to use the `rm` command with its `-R` ("**r**ecursive" again) option:

```
pcphxtr01:Unix Intro$ rm -R Copy\ of\ Treasure\ Island/

pcphxtr01:Unix Intro$ ls
Play  story.txt  Treasure Island  Work

pcphxtr01:Unix Intro$
```

[10 minutes]

Exercise 5.

1. Start in the `Unix Intro` directory.
2. Copy the file `lorem.txt` from the `Work` directory into the `Unix Intro` directory.
3. In the `Work` directory, create an empty directory called `Project Beta`.
4. Rename `Project Alpha` as `Project Delta`.
5. Copy the file `abc.txt` into `Project Delta`.
6. Copy `Project Delta` and its contents to `Project Epsilon` with a single command.
7. Remove the `Project Delta` directory and its content with a single command.
8. Change directory into the `Treasure Island` directory.
9. Examine its content. The file `story.txt` should no longer be there but be in the parent directory (if you have been following the demonstrator).
10. Without leaving the `Treasure Island` directory, copy the file `story.txt` into the current directory.

Anatomy of a command

We will start this section in the Unix Intro directory.

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro
```

We have seen the use of `ls` on its own to give a simple listing of the content of the current working directory:

```
pcphxtr01:Unix Intro$ ls
Fun  lorem.txt  Play  story.txt  Treasure Island  Work
```

and its use with the `-a` and `-l` options to give different output:

```
pcphxtr01:Unix Intro$ ls -a -l
total 389
drwxr-xr-x 1 y250 y250    512 2009-11-25 19:39 .
drwxr-x--- 1 y250 y250    512 2009-11-25 19:00 ..
drwxr-xr-x 1 y250 y250    512 2009-11-25 19:37 Fun
-rw-r--r-- 1 y250 y250   3693 2009-11-25 19:39 lorem.txt
drwxr-xr-x 1 y250 y250    512 2009-11-25 19:34 Play
-rw-r--r-- 1 y250 y250 390927 2009-11-25 19:34 story.txt
drwxr-xr-x 1 y250 y250    512 2009-11-25 19:36 Treasure Island
drwxr-xr-x 1 y250 y250    512 2009-11-25 19:40 Work
```

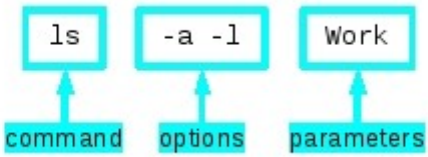
We have also seen it used to peer into another directory:

```
pcphxtr01:Unix Intro$ ls Work
abc.txt  ghi.txt    nonsense.txt  Project Delta
def.txt  lorem.txt  Project Beta  Project Epsilon
```

and we can combine these to give:

```
pcphxtr01:Unix Intro$ ls -a -l Work
total 12
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:40 .
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:39 ..
-rw-r--r-- 1 y250 y250   36 2009-11-25 19:34 abc.txt
-rw-r--r-- 1 y250 y250   36 2009-11-25 19:34 def.txt
-rw-r--r-- 1 y250 y250   36 2009-11-25 19:34 ghi.txt
-rw-r--r-- 1 y250 y250 3693 2009-11-25 19:34 lorem.txt
-rw-r--r-- 1 y250 y250 3664 2009-11-25 19:34 nonsense.txt
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:39 Project Beta
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:39 Project Delta
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:40 Project Epsilon
```

This is an example of the general case for a Unix command



The difference between an option and a parameter is not as clear cut as we might like. There are optional parameters and some commands even have compulsory options, but for our purposes options start with a dash and parameters don't.

Some options take arguments of their own. For example, `ls` has an option `-w` for setting the width of the output (overriding the width of the terminal). This has to be told what width to use, of course:

```
pcphxtr01:Unix Intro$ ls -w 40 Work
abc.txt  lorem.txt     Project Delta
def.txt  nonsense.txt  Project Epsilon
ghi.txt  Project Beta
```

The argument "40" counts as part of the options, not the parameters.

Long options

We have seen some options on the command "ls". The option "-a" lists **a**ll files in a director. The option "-l" gives a **l**ong format output. This approach works so long as you don't need more than 26 options, or perhaps 52 options if you use upper and lower case letters. This approach also requires you to be able to remember "a is for **a**ll" and that "l is for **l**ong". This second issue is the real problem with single character options.

Because of this more recent commands have tended to offer long form options as an alternative. This went through a half-way stage where word-style options followed the dash. This had the problem that it stopped being clear whether "-al" meant the long option "al" or the combination of the "a" option and the "l" option.

This led to the standard form of long options which are introduced with a *double* dash rather than a single one.

```
pcphxtr01:~$ ls --all

.               .gconf           My Music         .recently-used.xbel
..              .gconfd          My Pictures      .skel
Appscfg.PWF     .gnome2          My Video         .ssh
.bash_history   .gnome2_private  .nautilus        .thumbnails
.config         .gstreamer-0.10  .ooo3            Unix Intro
.dbus           .gvfs            .pulse           .Xauthority
Desktop         .hplip           .pulse-cookie    .xsession-errors
.dmrc           .ICEauthority    .pwf-linux
.esd_auth       Library          .pyhistory
.fontconfig     .local           .recently-used
```

The long form of an option may not always be what you expect it to be. The "-l" option does not have an analogue "--long". Instead, it uses it as an instruction of what format of output is wanted. The "-l" short option has the long option equivalent "--format=long".

```
pcphxtr01:~$ ls --format=long
total 4
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 Appscfg.PWF
drwxr-xr-x 1 y250 y250 512 2009-11-25 19:48 Desktop
drwxr-x--- 1 y250 y250 512 2009-11-25 13:08 Library
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Music
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Pictures
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Video
drwxr-xr-x 1 y250 y250 512 2009-11-25 19:39 Unix Intro
```

This may seem exceptionally long-winded but note how it has opened up the opportunity for many more formats of output than the traditional two short and long forms. We also gain the understanding of what the options mean. Finally, the verbosity of the options is balanced by the use of command line editing and history. We won't be typing them as often as you might think.

Note that we can't combine these long options the same way we can the single character options. The command "ls -l -a" can be written "ls -la". The command "ls --format=long --all" cannot be written "ls --format=longall".

Exercise 6.

Try `ls` with its various `--format` sub-options: `across`, `commas`, `horizontal`, `long`, `single-column`, `verbose`, and `vertical`.

When do `horizontal` and `vertical` produce different results?

[10 minutes]

Reading the fine manual

There are 39 single character options on `ls`. (Some are upper case so there can be more than 26.) There are also 39 long format options, some of which correspond to the short options. How are we supposed to keep track of this?

Some commands, such as `ls`, offer built in help facilities:

```
pcphxtr01:~$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

…

Exit status is 0 if OK, 1 if minor problems, 2 if serious trouble.

Report bugs to <bug-coreutils@gnu.org>.
```

Note that the information may go flying off the top of the screen. We will see how to use a utility, "`more`", to solve this problem later.

Not all commands are as well written, though. Unix has a command for printing out the manual pages for commands (and other things too) called "`man`" (short for "**man**ual").

```
pcphxtr01:~$ man ls
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents


SYNOPSIS
       ls [OPTION]... [FILE]...

…
```

To advance the output by a screen, press the space bar and to quit press [Q]. This is actually the `more` utility mentioned above but `man` has it built in so we don't need to call it explicitly.

The sections of a manual page are fairly standard:

```
NAME
       ls - list directory contents
```

The name section repeats the name of the command and gives a one-line summary of what the command does. This is used when searching for a command (which we will see soon).

```
SYNOPSIS
       ls [OPTION]... [FILE]...
```

The synopsis gives a very short outline of how to issue the command. Square brackets indicate that something is optional and the ellipsis ("…") indicates that there can be more than one of them. So this says that the "`ls`" command is followed by zero or more options and then zero or more file names. (Directories and files are lumped together for this purpose.)

```
DESCRIPTION
     List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

     Mandatory arguments to long options are mandatory for short options too.
```

```
       -a, --all
              do not ignore entries starting with .
```

The description section is where the meat of the manual page lies. This describes what the command does and then lists all the options and explains what they do.

```
AUTHOR
       Written by Richard Stallman and David MacKenzie.

COPYRIGHT
       Copyright © 2008 Free Software Foundation, Inc.  License GPLv3+: GNU
GPL version 3 or later <http://gnu.org/licenses/gpl.html>
       This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

The author section identifies the person or persons who write the application. The copyright section is typically where the copyright holders state the licence under which the application is released.

```
REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.
```

Many manual pages quote an address where bugs should be reported. If you find a bug in any application on a UCS platform, please report it to our Help Desk[8] instead. We'll take it from there.

```
SEE ALSO
       The full documentation for ls is maintained as a Texinfo manual.  If
the info and ls programs are properly installed at your site, the command

              info coreutils 'ls invocation'

      should give you access to the complete manual.
```

The final section can be used either to direct the reader to further information or to related commands. This manual page refers the reader to another source of information provided by the "`info`" command. We don't describe this command in this course as its user interface can be rather confusing.

Exercise 7.
Use "`man ls`" to find the short and long options to reverse the order that `ls` lists its files in. Try them.

[5 minutes]

There is a common expression used among the Unix community: "RTFM".
It is used by an expert when a user asks the busy geek a question that can be answered by perusing the manual page. It stands for "**R**ead **T**he **F**ine **M**anual".
Honest.

8 help-desk@ucs.cam.ac.uk

Launching graphical applications from the command line

This section requires that you be running in a graphical terminal window rather than in a text console. Please remember to be patient when switching between consoles.
If you are already running a web browser, please kill it off before starting this section.

Now that we have a command line we can use it to launch more useful commands than just `ls`. For example we can launch the Firefox web browser.

Background commands

For these examples, please quit any currently running browsers you may have.

We give the command "`firefox`", with a lower case "`f`".

```
pcphxtr01:~$ firefox↵
```

(For reasons that will become clear soon we are explicitly showing when the Return key, [↵], is being pressed.)

The Firefox web browser launches but in the terminal window the prompt has not been returned. We type "`ls`" (and press [↵]) but nothing happens (yet).

```
pcphxtr01:~$ firefox↵
ls↵
```

Now we quit the browser from the browser's menus (File→Quit) and we see that the prompt comes back, the `ls` command is repeated at it, and then run:

```
pcphxtr01:~$ firefox↵
ls↵
pcphxtr01:~$ ls
Appscfg.PWF  Desktop  Library  My Music  My Pictures  My Video  Unix Intro
pcphxtr01:~$
```

This means that each application launched is going to tie up a terminal window and any future commands typed in that terminal window will have to wait for the current command to finish before they are run. We can do better than that by running commands "in the background". To do this we follow the command with an ampersand, "`&`":

```
pcphxtr01:~$ firefox &
[1] 7941
pcphxtr01:~$
```

This time the prompt did come back immediately; the shell did not wait for the command to complete before asking for further instructions. We could now run `ls` again if we wanted for immediate results.

The "`[1]`" means that this is the first job we have in the background for this session. The number, `7941` in the example above, is a numerical identifier (the "process id") for this backgrounded command. We don't need to know about it, but yours will almost certainly be a different number.

During the course of this chapter you may see some warning messages appear, for example:

```
pcphxtr01:~$ firefox &
[1] 7941

pcphxtr01:~$ *** nss-shared-helper: Shared database
disabled (set NSS_USE_SHARED_DB to enable).

NPP_GetValue()

NPP_GetValue()
```

Don't worry about these. Graphical commands are quite noisy like this because their authors know that you don't get to see the messages if you launch the application graphically.

Note that because the command is running in the background these messages arrive “asynchronously”. This means thet they arrive whenever they want and not in response to you doing anything. If they cause you to lose track of your prompt, just press [↵] to get another.

There is one more feature to observe. We started the Firefox application from this shell and this shell gets informed when it finishes. If we close down Firefox from its menus (File→Quit) then we get a notification the next time a prompt is produced by the shell:

```
pcphxtr01:~$
[1]+  Done                    firefox
pcphxtr01:~$
```

The message “Done” indicates that the program terminated normally. You may get other messages if the program crashes.

Exercise 8.

1. Run the xeyes command in the background. It should get a “[1]”.
2. Run Firefox too. It should get a “[2]”.

[5 minutes]

Job control

What can you do if you have already launched a graphical application but forgot to add the ampersand?

There are facilities for taking a running job and moving it into the background. The process comes in two stages: first we stop the running program (“stop” as in “pause” rather than “finish”) and then we restart it in the background.

We will use a second instance of xeyes as an example. First we start it in the foreground (i.e. without the ampersand):

```
pcphxtr01:~$ xeyes
```

To stop the job we press [Ctrl]+[Z]:

```
pcphxtr01:~$ xeyes
^Z
[3]+  Stopped                 xeyes
pcphxtr01:~$
```

The “[3]” means that this is the third command we have either backgrounded or stopped (if the two commands from the exercise are still running). The “Stopped” says that it's stopped, obviously. This is followed by the command itself.

At this point we have the prompt back but the `xeyes` program isn't active; it's stopped. Try moving a terminal window over the `xeyes` window. You will see that the `xeyes` program doesn't track the pointer any more or even redraw itself properly; it's *completely* inactive.

Now we restart it in the background. To do this we issue the command "`bg`" (for "**b**ack**g**round").

```
pcphxtr01:~$ bg
[3]+ xeyes &
```

Again we get a response indicating what has happened. The "`[3]`" matches the identifier we saw when we stopped it. We also get the command repeated but this time with a trailing ampersand to indicate that it's running in the background and indeed we do have `xeyes` running in the background as if we had started it with an ampersand in the first place. If you move the terminal window over the `xeyes` window you will see it redraw itself correctly.

If we had stopped the command with [Ctrl]+[Z] and changed our mind we can always restart the command in the foreground with the "`fg`" ("**f**ore**g**round") command.

If you want to know what jobs you currently have running in the background, issue the "jobs" command:

```
pcphxtr01:~$ jobs
[1]   Running                 xeyes &
[2]-  Running                 firefox &
[3]+  Running                 xeyes &

pcphxtr01:~$
```

The number in square brackets is called the "job number" and we can use it to identify a particular process. If we had thee three jobs running in the background and wanted to foreground the second of them (firefox) then we could do that with "`fg %2`" where the number after the percentage sign is the job number of the job being brought to the foreground. If we wanted to background it again we could do that with [Ctrl]+[Z] and `bg` again. It would still be job number 2. This sort of pushing and pulling of jobs into the background tends to be a minority interest. Typically you will start a job in the background with the ampersand or you won't want it in the background at all.

Killing background jobs

Job control is also tied to another useful facility: killing rogue processes. Suppose a command had gone mad and was refusing to quit as you desperately clicked on the quit button. If a process doesn't die when you click its [×] button in the title bar then usually the graphical environment will wait sixty seconds or thereabouts and prompt you for whether or not you want the process killed more emphatically (while warning you that this will lose any unsaved work). Alternatively, we can use the command line.

We can start with the job numbers in square brackets. At the moment, if you have been following the notes, we have two instances of `xeyes` and one instance of `firefox` running. Note that our first instance of xeyes has job number `1`, i.e. is labelled "`[1]`" in the `jobs` output. If we run the command "`kill %1`" then the process corresponding to "[1]" is killed, in our case one of our `xeyes`. A message will appear that the job has been killed the next time you get a prompt.

```
pcphxtr01:~$ kill %1

pcphxtr01:~$
[1]-  Terminated              xeyes

pcphxtr01:~$ jobs
[2]-  Running                 firefox &
[3]+  Running                 xeyes &
```

If a command gets really stuck then there is a stronger version of the "`kill`" command. By default kill politely requests that a command should wind up its business and terminate. If this fails, there is an option "`kill -KILL`" which causes the process to be abruptly killed by the operating system. Note that you can only kill processes which "belong" to you.

```
pcphxtr01:~$ jobs
[2]-  Running                 firefox &
[3]+  Running                 xeyes &

pcphxtr01:~$ kill -KILL %3

pcphxtr01:~$
[3]+  Killed                  xeyes

pcphxtr01:~$ jobs
[2]+  Running                 firefox &
```

[5 minutes]

Exercise 9.

1. In the Firefox that you are running in the background navigate to another page.
2. Kill the Firefox instance with the "`kill -KILL`" command and its job number.
3. Start Firefox again. You should get a warning window:
   "Your last Firefox session closed unexpectedly. You can restore the tabs and windows from your previous session, or start a new session if you think the problem was related to a page you were viewing."
4. Select the option to restore the previous session. You should appear back at the last page you were looking at.

Why would you want job control?

What's the point of job control? After all, you can always launch another terminal window.

Backgrounding can be used for much more than just graphical applications however. Any job that is going to take time to complete can be backgrounded. These jobs can be run in a purely text environment where you can't just open another terminal. Alternatively, as we will see later, you may be running the program (graphical or otherwise) on a remote system where you only have one connection established. Backgrounding jobs is often a lot less hassle than establishing another connection.

What would the GUI do?

We can ask the windowing system to open a file with "whatever application it would have used if we had double clicked on the icon in the graphical interface". The command to do this is called "`gnome-open`". (The style of window interface we use on PWF Linux is called "GNOME".)

```
pcphxtr01:Unix Intro$ pwd
/home/y250/Unix Intro

pcphxtr01:Unix Intro$ ls
Play  story.txt  Treasure Island  Work

pcphxtr01:Unix Intro$ gnome-open story.txt

pcphxtr01:Unix Intro$ gnome-open Treasure\ Island/hispaniola.png

pcphxtr01:Unix Intro$
```

This launches `gedit` (the **G**NOME **edit**or) for the text file `story.txt` and `eog` ("**e**ye **o**f **g**nome", the default GNOME picture viewer) for the graphical file `hispaniola.png`.

Because the command is only used for graphical applications it automatically "detaches" the applications (`gedit`, `eog`, etc.) from the terminal so it does not need to be backgrounded.

The application cannot usefully open non-existent files. It will not launch the application suitable for a file of that name as a quick way to start an editor with an empty file, for example.

To create an empty file, `fubar.txt` say, use the "`touch`" command: "`touch fubar.txt`".

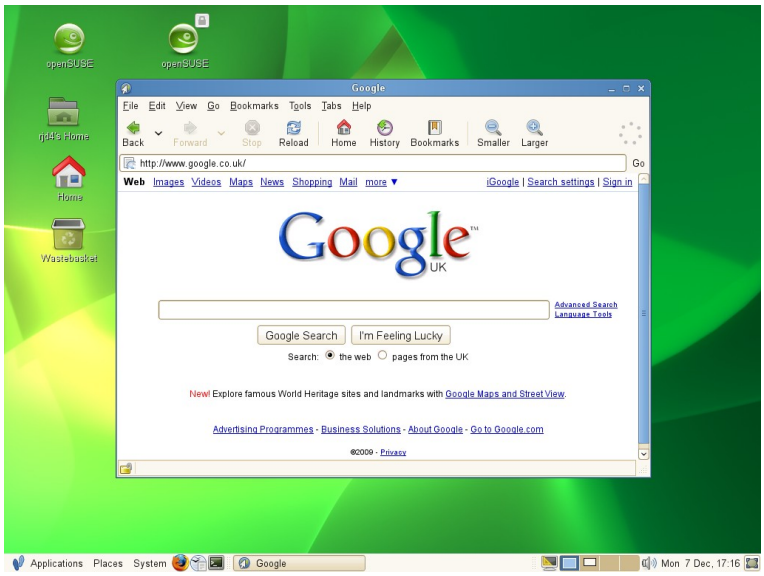
[5 minutes]

Exercise 10.

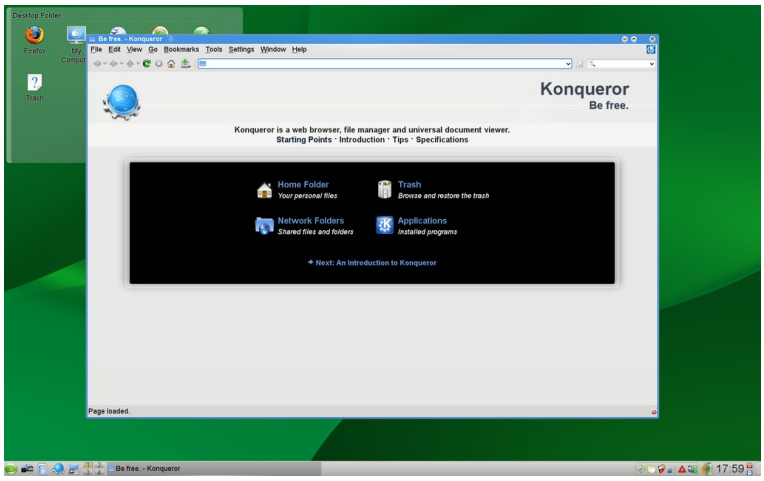
1. Run `gnome-open` on each of the files in `Work/Project Epsilon`. (It will only have this name if you have done the previous exercises. It was originally called `Project Alpha`.)
2. Close down the applications and then try to work out the direct commands to use to run the same applications as `gnome-open` did. (e.g. `gedit story.txt` is the equivalent of `gnome-open story.txt`) Don't forget to background them. (Hint: Help→About in an application typically identifies the application.)

Just for interest

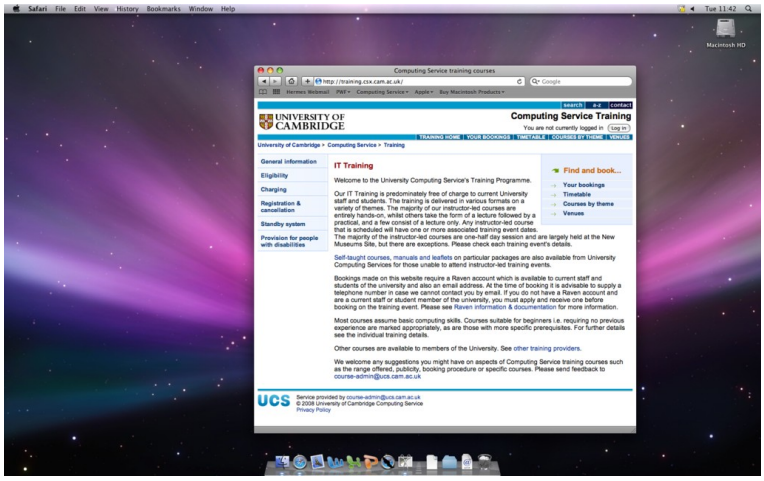
The other standard windowing environment for Linux is called “KDE” (as opposed to GNOME) and the corresponding command for it is called “`kde-open`”. On MacOS X (a different flavour of Unix) the command is called simply “`open`” as there is no choice of windowing environment.



GNOME



KDE



MAC OS X

Command line editing

We have already seen how useful tab completion is. It is not the only assistance that the shell can offer us.

Changing the command line

Suppose we have typed in a command but not hit the [↵] key yet:

```
pcphxtr01:~$ ls Unix\ Intro
```

(We are showing the cursor as a solid block. Usually it's blinking which is hard to do on paper.) It's at this point we realise that we meant to type "`ls -l`" rather than plain "`ls`". If we press the left arrow key, [←], at this point the cursor moves back. We tap it enough times to move back to just after the "`ls`":

```
pcphxtr01:~$ ls Unix\ Intro
```

at this point we simply type "␣`-l`" to insert the option. (We must remember to type that leading space to split the command from its options.)

```
pcphxtr01:~$ ls -l Unix\ Intro
```

At this point we can hit [↵]. There is no need for us to move back to the end of the line.

```
pcphxtr01:~$ ls -l Unix\ Intro
total 2
drwxr-xr-x 1 y250 y250 512 2009-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2009-04-28 20:00 Treasure Island
drwxr-xr-x 1 y250 y250 512 2009-04-28 21:34 Work

pcphxtr01:~$
```

Alternatively, suppose we had mistyped the "`ls -l`" as "`ls -k`":

```
pcphxtr01:~$ ls -k Unix\ Intro
```

We can move the cursor back with the left arrow, [←], to just after the "`-k`":

```
pcphxtr01:~$ ls -k Unix\ Intro
```

and press the backspace key, [⟵], once to delete the "`k`":

```
pcphxtr01:~$ ls - Unix\ Intro
```

Then we type the "`l`" that we wanted in the first place:

```
pcphxtr01:~$ ls -l Unix\ Intro
```

Then we hit return, [↵]:

```
pcphxtr01:~$ ls -l Unix\ Intro
total 2
drwxr-xr-x 1 y250 y250 512 2009-04-23 18:20 Play
drwxr-xr-x 1 y250 y250 512 2009-04-28 20:00 Treasure Island
drwxr-xr-x 1 y250 y250 512 2009-04-28 21:34 Work

pcphxtr01:~$
```

In addition to moving left you can, of course, move right with [→], but not beyond the end of the line.

Be careful to distinguish the left arrow key, [←], from the backspace key, [⟵], in these notes.
The left arrow key [←] is typically part of the cluster of four arrow keys on the keyboard ([←], [↑], [↓], [→]) between the main keypad and the numeric keypad to the right.
The backspace key, [⟵], sits on its own at the top right of the main keypad and is shown with a longer arrow.

There are more options than simply moving forwards and backwards one character at a time. To move to the start of the line press [Home] (or [Ctrl]+[A]), and for the end of the line press [End] (or [Ctrl]+[E]). To move a word at a time use [Ctrl]+[←] and [Ctrl]+[→]. There is a summary of all these movement options at the end of the notes.

History

As well as moving the cursor left and right you can also move it up and down. This gives you access to the shell's history mechanism.

Suppose we type four commands:

```
pcphxtr01:Desktop$ cd
pcphxtr01:~$ pwd
/home/y250
pcphxtr01:~$ cd Unix\ Intro/
pcphxtr01:Unix Intro$ ls
Play  Treasure Island  Work
```

These four commands exist in the shell's memory as if they were lines in a file with a fifth, blank line for the command the shell is waiting for that we've not started typing yet:

```
cd
pwd
cd Unix\ Intro/
ls
                                    ← you are here
```

If we press the up arrow, [↑], the command shown on the command line moves back through this history. If we press it three times we end up with the `pwd` command back on our command line.

```
cd
pwd                                 ← you are here
cd Unix\ Intro/
ls
```

We press [↵] to run the command:

```
pcphxtr01:~$ pwd
/home/y250/Unix Intro/
pcphxtr01:~$
```

If we overshoot by typing too many [↑] we can also type [↓] to move back down the list of lines too.

If our command line history has a command which is almost exactly what we want but not quite then we can also scroll back through the commands and then use the other arrow keys and backspace key to edit the historical line to give us the line we want.

If you want to see the “history file” for real, give the command “`history`”.

[5 minutes]

Exercise 11.
This example illustrates an additional way to use the history mechanism.

1. Change directory to the `Unix Intro` directory.
2. Run `ls -l`.
3. Change directory into the Work subdirectory.
4. Press [Ctrl]+[R].
5. Notice the prompt change to "`(reverse-i-search)`':`"
6. Start to type the "`ls -l`" command as far as the first "`l`" (i.e. just one letter)
7. Notice how the "`ls -l`" command is found and the last "`l`" is indicated by the prompt. ([Ctrl]+[R] triggers a backwards search.)
8. Press the "s" key.
9. Notice how the prompt jumps to the "`ls`".
10. Hit [↵] to issue the command.

[5 minutes]

Exercise 12.
This example takes it a bit further.

1. Issue the command "`cp lorem.txt lorem2.txt`".
2. Press [Ctrl]+[R] again.
3. Press "`l`". Note that the shell finds the last "`l`" in the previous instruction.
4. Press [Ctrl]+[R] again. Note that the shell finds the previous "`l`".
5. Press [Ctrl]+[R] again. Note that this time the prompt jumps back to the last "`l`" in the "`ls -l`" command.
6. Hit [↵] to issue the command.

(In practice you would probably just type the "s" to jump back to that command.)

Clearing the screen

Not strictly command line editing but related is the facility to clear your screen. To clear your screen you can issue the command "`clear`" which does exactly what you would expect. However, there is a more powerful way to do it: [Ctrl]+[L].

Simply pressing [Ctrl]+[L] at the prompt does exactly the same as the `clear` command but, unlike `clear`, [Ctrl]+[L] can be typed at any point. Suppose we have some command output on the screen already and start to type a command (but don't press [↵]):

```
pcphxtr01:~$ pwd
/home/y250/Unix Intro/

pcphxtr01:~$ ls -
```

We can then press [Ctrl]+[L] at that point:

```
pcphxtr01:~$ pwd
/home/y250/Unix Intro/

pcphxtr01:~$ ls -[Ctrl]+[L]
```

to clear the screen but leave the partially typed command at the top of the screen ready to be continued:

```
pcphxtr01:~$ ls -
```

We can now carry on typing the command on an otherwise clear terminal:

```
pcphxtr01:~$ ls -a
.  ..  fubar.txt  Fun  lorem.txt  Play  story.txt  Treasure Island  Work
```

Running applications in the CLI

After our brief excursion into graphical applications, we will return to the pure text world.

Reading plain text files

The classic command for reading a plain text file is called “more”. We can see this if we move to the Work directory and apply it to the lorem.txt file:

```
pcphxtr01:Work$ pwd
/home/y250/Unix Intro/Work

pcphxtr01:Work$ more lorem.txt
TOP OF FILE


Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec at purus sed m
agna aliquet dignissim. In rutrum libero non turpis. Fusce tempor, nulla sit a
met pellentesque feugiat, nibh quam dapibus dui, sit amet ultrices enim odio n
ec ipsum. Etiam luctus purus vehicula erat. Duis tortor lorem, commodo eu, sod
ales a, semper id, diam. Praesent nisl justo, placerat id, rutrum et, vulputat
e ut, metus. Pellentesque habitant morbi tristique senectus et netus et malesu
ada fames ac turpis egestas. Etiam non neque. Curabitur dui. Praesent mi erat,
aliquam eget, aliquet lobortis, pharetra quis, lacus. Nulla facilisis, purus
eget porttitor bibendum, nisi augue auctor lectus, et mollis odio nisi in urna
. Nam felis tortor, porttitor in, ultrices vitae, bibendum non, purus. Cras lu
ctus.

Sed lacus justo, sollicitudin eu, interdum sed, fermentum id, sapien. Class ap
tent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymen
aeos. Nunc purus. In in purus sit amet tellus scelerisque molestie. In in tort
or. Pellentesque viverra, nibh quis feugiat condimentum, metus neque condiment
um lectus, ac commodo turpis justo sit amet nisl. Donec mollis vestibulum feli
s. Aliquam ornare, felis eu suscipit lacinia, neque lectus hendrerit sem, ac v
ulputate est tortor et ligula. Suspendisse quis sapien a urna laoreet elementu
m. Pellentesque nisl ante, tempus ac, porta vel, malesuada vel, sapien. In tor
tor justo, sollicitudin vel, aliquet sed, consequat ac, enim. Proin elit odio,

--More--(40%)
```

(The author has cheated in the screen above. The pwd and more commands will both have been scrolled off the top of the screen by the text of the file.)

The line “TOP OF FILE” is in the file. We have added it for your navigational convenience.

The more command is named after its prompt which indicates that there is more of the file to follow. We have seen something very similar in the the man command pages its output.

If we press the space bar once we get the next screenful:

```
 convallis ac, mollis nec, iaculis et, lectus.
Pellentesque ullamcorper leo eu est. Aliquam metus. Cras sem augue, mattis ege
t, congue vitae, adipiscing in, dolor. Fusce elementum mollis urna. Pellentesq
ue quam. Duis pede tortor, euismod non, varius vitae, consectetuer vel, dui. S
uspendisse potenti. Nullam vehicula, justo euismod imperdiet vulputate, turpis
lacus elementum nulla, ut posuere velit ipsum id elit. Maecenas at justo id r
isus tristique tristique. Vivamus auctor viverra felis. Fusce nonummy commodo
lacus. Morbi et nisi eget nulla iaculis semper. Ut sit amet eros. Quisque in r
isus. Duis id tellus nec magna condimentum facilisis. Fusce feugiat. Curabitur
eleifend tincidunt purus. Etiam ligula mi, mollis vitae, dapibus et, posuere
vel, lacus.


Duis erat. Mauris metus purus, scelerisque ac, pulvinar et, iaculis eu, mauris
. Duis a lectus. Vivamus dolor nisl, aliquet a, venenatis id, consectetuer ut,
lorem. Suspendisse nisi lectus, sollicitudin non, condimentum vel, nonummy ve
l, lacus. Cras nunc justo, tincidunt vel, vulputate non, aliquet euismod, turp
is. Proin sagittis placerat lectus. Donec in lorem. In lacinia, leo ac luctus
tincidunt, nunc pede pulvinar tortor, in molestie tellus sem quis tellus. Maec
enas vel enim. Mauris tincidunt nibh quis mauris ullamcorper pretium. Duis con
sequat commodo risus. Vivamus rutrum. Vivamus dolor augue, imperdiet consectet
uer, dictum at, eleifend et, sem. Suspendisse sed eros. Integer magna purus, e
lementum eget, egestas id, porta id, nunc. Pellentesque habitant morbi tristiq

--More--(82%)
```

and if we press [B] we go **b**ack a screenful. If we keep pressing the space bar we will eventually reach the end of the file, when `more` will terminate. Alternatively we can just press [Q] to **q**uit immediately.

There is a more modern version of `more` called "`less`" as a pun on the original's name. This is very similar to `more` in that it uses the space bar to page through a document but it does not "fall off the end" when the file reaches the end. Instead you must explicitly press [Q] to quit. The `less` command also reverts the screen to the state it was before the command was run once it is finished.

Searching plain text files

A common requirement is to be able to search through a plain text file for particular words or phrases. For this we will use the text of the story Treasure Island, sitting in `Treasure Island/story.txt` or `Unix Intro/story.txt` depending on whether you have completed the exercises. The search command is called "`grep`":

```
pcphxtr01:Unix Intro$ grep Rum story.txt
"Rum," he repeated. "I must get away from here. Rum! Rum!"
but you're on'y a boy, all told. Now, Ben Gunn is fly. Rum wouldn't
fathom and a half of water. We all pulled round again to Rum Cove,

pcphxtr01:Unix Intro$
```

Note that the search is case sensitive. Lines containing "rum" with a lower case "r" are not printed. Also note that while "Rum" appears three times in the first line the line is only printed out once.

We can search for more than one word by quoting together the phrase to search for. Recall that quotes lump words together so they are treated as a single item.

```
pcphxtr01:Unix Intro$ grep "Ben Gunn" story.txt
"Ben Gunn," he answered, and his voice sounded hoarse and awkward,
like a rusty lock. "I'm poor Ben Gunn, I am; and I haven't spoke with
…
Ben Gunn was on deck alone, and as soon as we came on board he began,
father of a family. As for Ben Gunn, he got a thousand pounds, which
```

However, note that you won't catch lines from the file like this:

```
shame and lies and cruelty, perhaps no man alive could tell. Yet there
```

```
were still three upon that island--Silver, and old Morgan, and Ben
Gunn--who had each taken his share in these crimes, as each had hoped in
vain to share in the reward.
```

The two words, “Ben” and “Gunn”, have been split over a line break and have not been detected.

We can search for lower case “rum” also:

```
pcphxtr01:Unix Intro$ grep rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
up my chest. I'll stay here a bit," he continued. "I'm a plain man; rum
…
knuckled under, put up his weapon, and resumed his seat, grumbling like
…
     Yo-ho-ho, and a bottle of rum!"
"this won't do. Stand by to go about. This is a rum start, and I can't
the rum, Darby!"
```

Note that it matched a line containing the word “g**rum**bling”. We can tell `grep` to search for whole words only with the “`-w`” (“**w**ord”) option:

```
pcphxtr01:Unix Intro$ grep -w rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"

pcphxtr01:Treasure Island$
```

We can search for “Rum” and “rum” (and “rUm” etc.) by requesting a case insensitive search with the option “`-i`” (“**i**nsensitive”):

```
pcphxtr01:Unix Intro$ grep -i rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
knuckled under, put up his weapon, and resumed his seat, grumbling
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"

pcphxtr01:Unix Intro$
```

If we want just the word “rum” but in either case we combine the `-i` and `-w` options:

```
pcphxtr01:Unix Intro$ grep -iw rum story.txt
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
…
"this won't do. Stand by to go about. This is a rum start, and I
the rum, Darby!"

pcphxtr01:Unix Intro$
```

Again, just as with the options to `ls`, the options “`grep -iw`”, “`grep -wi`”, “`grep -i -w`”, and “`grep -w -i`” are all equivalent.

Exercise 13.
How many times does “yo-ho-ho” (regardless of case) appear in the text of Treasure Island?

Use `grep` for the search and count the instances manually.

[5 minutes]

Counting text

Another operation which can be useful on plain text is counting the words, lines or even characters. Unix has a command called “wc” (“**w**ord **c**ount”) that does this:

```
pcphxtr01:Unix Intro$ wc story.txt
  7857  71516 390927 story.txt

pcphxtr01:Unix Intro$
```

This tells us that the file contains 7,857 lines, 71,516 words, and 390,927 characters. We can demand just some of the information with the three `wc` options: `-l` for the line count, `-w` for the word count, and `-c` for the character count:

```
pcphxtr01:Unix Intro$ wc -l story.txt
7857 story.txt

pcphxtr01:Unix Intro$
```

Exercise 14.
How many words are there in the `lorem.txt` file?

[5 minutes]

Editing plain text files

If you want to edit a plain text file then the graphical editors such as `gedit` (the default graphical text editor, as given by `gnome-open`) are probably easiest for you. This is what we will use in this course.

There are two plain text editors which can be used in text consoles. These are “`emacs`” (which also has a graphical form) and “`vi`”. These are the grand old men of the Unix world and require training before they can be used. The UCS offers courses on both.

Telling the time

The “`date`” command seems fairly straightforward at first glance; it gives the date and the time:

```
pcphxtr01:~$ date
Tue Apr 28 20:37:12 BST 2009

pcphxtr01:~$
```

However, we can modify the format of the output to give just some of that information. The `date` command accepts an argument called a “format string” which controls the look of its output:

```
pcphxtr01:~$ date +"%d %m %Y"
28 04 2009

pcphxtr01:~$
```

The string in quotes after the plus sign is the format string. The letters after percentage characters are converted to elements of the date and time. `%d` is converted to the day of the month, `%m` to the numerical month of the year and `%Y` to the year. Characters without preceding percentage characters are simply

repeated, like the spaces in the example above. The reference sheet at the back of these notes contains a set of useful formatting options.

The format string on the date command is one of the places where you are likely to need the quotes. If you have any spaces in your format string then you must quote it. Otherwise the date command will take everything from the "+" to the first space as the format string and either ignore or get confused by everything after that space.

```
pcphxtr01:~$ date +"%Y %B %d"
2010 March 01

pcphxtr01:~$ date +%Y %B %d
date: extra operand `%B'
Try `date --help' for more information.

pcphxtr01:~$
```

Exercise 15.
Work out the format string for the `date` command so that the time looks like this:
`2009-04-28 23:57:37`

(Do use the crib sheet at the back of these notes.)

[5 minutes]

Exercise 16.
Change the format string for the `date` command in the previous exercise so that the time looks like this:

```
Date: 2009-04-28
Time: 23:57:37
```

[5 minutes]

(This should be the output of a *single* date command. Recall that a %n in a format string is converted into a line break.)

Repeating the command line

The next command seems rather pointless. Its true utility will only become apparent later. The "`echo`" command repeats whatever you give it as arguments.

```
pcphxtr01:~$ echo one two three
one two three

pcphxtr01:~$
```

We can use it to see how quotes and escaping don't make it through to the command itself. They act purely as instructions to the shell.

```
pcphxtr01:~$ echo "one two three"
one two three

pcphxtr01:~$ echo 'one two three'
one two three

pcphxtr01:~$ echo one\ two\ three
one two three

pcphxtr01:~$
```

A command line calculator

There are plenty of graphical applications that act like a pocket calculator. You can launch one in graphical mode with the menu selection Applications → Utilities and Accessories → Gnome Calculator.

However, it's often much faster to work in the command line and there is a command line application called "bc" ("**b**asic **c**alculator") which fulfils this purpose.

Used on its own, bc accepts input from the keyboard and its input is ended either by [Ctrl]+[D] or the command "quit". It does not have a prompt:

```
pcphxtr01:~$ bc
1+2
3
56-65
-9
[Ctrl]+[D]
pcphxtr01:~$
```

Because the times and divide symbols ("×" and "÷") do not appear on the standard keyboard, computers tend to use the asterisk and forward slash symbols ("*" and "/") for multiplication and division instead:

```
pcphxtr01:~$ bc
4*2
8
4/2
2
[Ctrl]+[D]
pcphxtr01:~$
```

However, bc truncates to zero decimal places by default:

```
pcphxtr01:~$ bc
1/2
0
2/3
0
[Ctrl]+[D]
pcphxtr01:~$
```

This can be fixed by setting a parameter called "scale" within the application to be the number of decimal places to be used:

```
pcphxtr01:~$ bc
scale=5
1/2
.50000
2/3
.66666
[Ctrl]+[D]
pcphxtr01:~$
```

The scale parameter sets the number of decimal places, not the number of significant figures:

```
pcphxtr01:~$ bc
scale=5
2/3
.66666
20/3
6.66666
2/300000
0
[Ctrl]+[D]
pcphxtr01:~$
```

The bc application can work to arbitrary precision. You can set `scale` to be as large as you like.

[5 minutes]

Exercise 17.
Using bc, calculate the following:

1. 11×11111111
2. 111×1111111
3. 1111×111111
4. 11111×11111

and calculate these to ten decimal places:

5. $\frac{355}{113}$
6. $\frac{22}{7} - \frac{223}{71}$

Just for interest

There is another command line calculator called "dc". Most people discover this application by mistake. They wanted to type "cd" and they accidentally typed "dc" instead. The "dc" calculator is a "Reverse Polish" calculator where you type the numbers and then the operation (and then the instruction to print the result). We will give only one example, as it's almost certainly not what you want to use:

```
pcphxtr01:~$ dc
5 4 + p
9
[Ctrl] + [D]
pcphxtr01:~$
```

If you find yourself in dc when you meant to type "cd", use [Ctrl]+[D] to exit:

```
pcphxtr01:~$ dc
[Ctrl]+[D]
pcphxtr01:~$
```

Redirecting data and piping commands

The “cat” command is a program for con**cat**enating one or more plain text files. You can concatenate a single file but it is rarely useful.

For example in the Work directory we have this:

```
pcphxtr01:Work$ cat abc.txt def.txt ghi.txt
ABC
ABC
ABC
ABC
…
GHI
GHI
GHI
GHI
GHI

pcphxtr01:Work$
```

The output containing the three files combined went to the terminal. What if we wanted to combine them into a new file? We could power up an editor and combine them there but there is a slicker way.

Standard output

All the Unix commands that deal with linear data (such as plain text) typically spit their output to the terminal by default where that linear stream of data typically rushes past you. This output stream is known technically as ”standard output” and by default a program's standard output goes to the terminal where it is being run. But it can be redirected.

```
pcphxtr01:Work$ cat abc.txt def.txt ghi.txt > combined.txt

pcphxtr01:Work$
```

The “>” redirection (think of it as an arrow) has taken the standard output of the cat command and redirected it from the terminal into a file combined.txt. Note that the redirection overwrites any content of the file that was previously there.

```
pcphxtr01:Work$ more combined.txt
ABC
…
GHI
GHI

pcphxtr01:Work$ cat abc.txt def.txt > combined.txt

pcphxtr01:Work$ more combined.txt

ABC
…
DEF
DEF

pcphxtr01:Work$
```

There is a variant of “>” which *appends* to the end of any pre-existing file (and which still creates the file if it does not): “>>”.

```
pcphxtr01:Work$ cat ghi.txt >> combined.txt
pcphxtr01:Work$ more combined.txt
ABC
…
GHI
pcphxtr01:Work$
```

[5 minutes]

Exercise 18.
Create a file `lorem2.txt` which is two copies of `lorem.txt`, one after the other. Run wc on `lorem.txt` and `lorem2.txt` to check that all three counts have doubled. If you have a `lorem2.txt` file left over from a previous exercise do not worry about over-writing it.

(Do not use an editor!)

Standard input

In addition to standard output, Unix commands also have the concept of “standard input”. This is where commands can get their data from. If a command takes a file name as an argument then it is going to get its data from that file, but those commands can often also take their input from the keyboard directly. In this case we use [Ctrl]+[D] to mark “end of input”.

Be careful with [Ctrl]+[D] because if you just type it at the shell when there's no command pulling in input the shell interprets it as “no more input for me” and exits.

We have met wc already:

```
pcphxtr01:Work$ wc abc.txt
 9  9 36 abc.txt
pcphxtr01:Work$
```

We can also tell it to get its input from its standard input (the keyboard) by omitting any file name:

```
pcphxtr01:Work$ wc
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
[Ctrl]+[D]
     2     15     75
pcphxtr01:Work$
```

Note that no file name is quoted in wc’s output line. Standard input is anonymous.

Because [Ctrl]+[D] is potentially so dangerous there is a special shell syntax to say “end the input with *this*”:

```
pcphxtr01:Work$ wc <<END
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
END
     2     15     75
pcphxtr01:Work$
```

Note that the ending string, “END”, does not contribute to the word count.

Also note that there is nothing special about “END”. It is just the string of characters that follows “<<”:

```
pcphxtr01:Work$ wc <<STOP
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
STOP
     2     15     75

pcphxtr01:Work$
```

There is no need for the end marker to be a word, either. Punctuation works too and "!" is a traditional marker:

```
pcphxtr01:Work$ wc <<!
The quick brown fox jumps over the lazy dog.
The cow jumped over the moon.
!
     2     15     75

pcphxtr01:Work$
```

Finally we can redirect the standard input from a pre-existing file with the "<" redirector. Again, think of it as an arrow but this time pointing *into* the command rather than out of it.

```
pcphxtr01:Work$ wc < combined.txt
   27     27    108

pcphxtr01:Work$
```

Again, notice that it produces anonymous output for standard input and contrast it with the similar command line:

```
pcphxtr01:Work$ wc combined.txt
   27     27    108    combined.txt

pcphxtr01:Work$
```

Piping

Why would we ever want this? We can always give the file name on the command line, after all.

The full power of standard input and output only becomes clear when we combine the two. We can take the standard output of one command and feed it directly into the standard input of a second. This simple construction allows us to combine Unix commands in very powerful ways.

It is done by placing the "|" character (pronounced "pipe") between the two commands:

```
pcphxtr01:Work$ cat lorem.txt combined.txt | wc
     40     568    3801

pcphxtr01:Work$
```

If we move back to the `Treasure Island` directory we can see a classic use of piping and the `more` command:

```
pcphxtr01:Work$ cd ../Treasure\ Island/

pcphxtr01:Treasure Island$ grep -iw rum story.txt | more
             Yo-ho-ho, and a bottle of rum!"
called roughly for a glass of rum. This, when it was brought to him,
…
"Rum," he repeated. "I must get away from here. Rum! Rum!"
the stranger. I got the rum, to be sure, and tried to put it down his
--More--
```

We no longer have to have output rush past us on the screen.

Exercise 19.
How many lines in Treasure Island contain the word “rum” in any case?

(How should you combine `grep` and `wc`?)

[5 minutes]

Note that “the number of lines containing the word ‘rum’” is *not* the same as “the number of times the word ‘rum’ occurs”.

A very common use of piping is to the more command. If a command's output is too long to fit on a single screen then piping it through more paginates the output.

Exercise 20.
Run these two commands:

```
ls --help
ls --help | more
```

[1 minute]

Page through the output of the second command. You have the choice of this and “`man ls`”.

This addresses the uncontrollable output issue we met with “`ls --help`” earlier.

File name globbing

The next command line trick we will see is called “wild carding” or “globbing”. This allows us to express a set of files without having to list them all manually.

Asterisk

Observe this use of the `echo` command in the `Work` directory:

```
pcphxtr01:Work$ echo abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt nonsense.txt
abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt nonsense.txt
pcphxtr01:Work$
```

The trick behind globbing is that the shell will take certain special characters and convert them into lists of file names. So, for example, the expression “`*.txt`” is converted into the list of files that end in “`.txt`”. The “`*`” stands for “anything”.

```
pcphxtr01:Work$ echo *.txt
abc.txt combined.txt def.txt ghi.txt lorem2.txt lorem.txt nonsense.txt
pcphxtr01:Work$
```

Globbing does not work inside quotes or if the asterisk is preceded by a backslash. (Remember that this makes characters special to the shell, like space, ordinary parts of file names. Well, it works on asterisks too.)

```
pcphxtr01:Work$ echo "*.txt"
*.txt
pcphxtr01:Work$ echo '*.txt'
*.txt
pcphxtr01:Work$ echo \*.txt
*.txt
pcphxtr01:Work$
```

Also, if a glob (“wild card”) doesn't match anything it passes through unaltered.

```
pcphxtr01:Work$ echo *.foo
*.foo
pcphxtr01:Work$
```

It is not `echo` that is doing this; it is the *shell*. All globbing works equally well with any command:

```
pcphxtr01:Work$ wc *.txt
    9     9    36 abc.txt
   27    27   108 combined.txt
    9     9    36 def.txt
    9     9    36 ghi.txt
   26  1082  7386 lorem2.txt
   13   541  3693 lorem.txt
    9   535  3664 nonsense.txt
  102  2212 14959 total
pcphxtr01:Work$
```

The asterisk can expand into nothing:

```
pcphxtr01:Work$ echo abc.txt*
abc.txt

pcphxtr01:Work$
```

Question mark

There are other globs. The asterisk, "`*`", expands into "anything". The question mark, "?", expands into "any *one* character":

```
pcphxtr01:Work$ echo abc.t*
abc.txt

pcphxtr01:Work$ echo abc.t?
abc.t?

pcphxtr01:Work$ echo abc.t??
abc.txt

pcphxtr01:Work$ echo abc.t???
abc.t???

pcphxtr01:Work$
```

Square brackets — only for the keen

The third glob is the most difficult. The question mark, "?", expands into "any one character". The last glob allows us to say "any one character from this set" or even "any one character not from this set". Obviously this glob is going to be most complex as we have to be able to specify the set.

The glob "`[aeiou]`" means "any one of 'a', 'e', 'i', 'o', or 'u'":

```
pcphxtr01:Work$ echo abc.t[xyz]t
abc.txt

pcphxtr01:Work$ echo abc.t[abc]t
abc.t[abc]t

pcphxtr01:Work$
```

We can negate this membership with the syntax "`[^aeiou]`" to mean "any one character that is *not* 'a', 'e', 'i', 'o', or 'u'":

```
pcphxtr01:Work$ echo abc.t[^xyz]t
abc.t[^xyz]t

pcphxtr01:Work$ echo abc.t[^abc]t
abc.txt

pcphxtr01:Work$
```

The "new" globs — only for the very keen

In the old days when all computers knew about was the classic ASCII character set (no accents, no non-American letters) there was a trick added to this glob syntax so that ranges of letters could be expressed. Documentation dating back to the dark ages before internationalisation may talk about using globs like "`[a-z]`" to mean any lower case letter, and "`[A-Z]`" to mean any upper case letter.

This is no longer true.
The ordering of characters that this trick replied on no longer hold. The old ASCII alphabet went 0,1,2,3,4,5,6,7,8,9,A,B,C,D,…,X,Y,Z,a,b,c,d,…,x,y,z. Today the letters match according to complex rules depending on the language being spoken at the time and there are new globs to allow for this.

The glob "`[[:lower:]]`" is special and matches any single lower case letter in whatever language (more

properly “locale”) you happen to be using, “`[[:upper:]]`” matches any upper case letter, and “`[[:digit:]]`” matches any one digit.

If you wanted to match any one lower case letter or any one digit then you could use the expression “`[[:lower:][:digit:]]`” or “`[[:digit:][:lower:]]`” (though there's actually a shorter version for this case). If you wanted to match any digit or the letter “`N`” you could use “`[N[:digit:]]`” or “`[[:digit:]N]`”.

The set of these new globs is given here and repeated at the end of these notes.

`[:alnum:]`	Any alphabetic character (upper or lower case) or any digit.
`[:alpha:]`	Any alphabetic character (upper or lower case).
`[:blank:]`	Any horizontal white space (space or tab, essentially).
`[:digit:]`	Any of the ten digits.
`[:lower:]`	Any lower case alphabetic character.
`[:upper:]`	Any upper case alphabetic character.

Environment variables

Certain elements of the Unix world, including the command line interpreter itself can be influenced by a collection of settings known as “the environment”. Each of these settings is individually referred to as an “environment variable”.

We can see the entire environment with the command “`env`”. This produces one line of output for each setting, shown in the form

```
VARIABLE=value
```

with the name of an environment variable traditionally given in upper case. There are many environment variables so the output is best piped through `more` or `less`:

```
pcphxtr01:~$ env | more
MODULE_VERSION_STACK=3.1.6
NNTPSERVER=nntp-serv.cam.ac.uk
INFODIR=/usr/local/info:/usr/share/info:/usr/info
…
```

To illustrate environment variables we will need to use our graphical interface again. Environment variables work everywhere but this demonstration needs a resizable window.

We can search for the string “`TERM`” (all upper case) in the output of `env` by piping to the `grep` command. We see that the `TERM` environment variable is set to `xterm` and that the `COLORTERM` variable is set to `gnome-terminal`:

```
pcphxtr01:~$ env | grep TERM
TERM=xterm
COLORTERM=gnome-terminal
pcphxtr01:~$
```

The terminal window is a Gnome Terminal window. This is a modern version of an old sort of terminal window called an `xterm`. Most programs use the value of the `TERM` environment variable to work out what sort of terminal it is and, therefore, how to talk to it.

Running `env` and `grep`ping the results is slow and inefficient. There is a simple mechanism to determine the value of any variable (and there is another sort other than environment variables). Recall that the `echo` command repeats its arguments only after the shell has rewritten them (as happened with file name globbing). The shell also has a syntax for converting the names of variables into their corresponding values.

```
pcphxtr01:~$ echo "${TERM}"
xterm
pcphxtr01:~$
```

Just this once we will show you a short cut. In this specific case the double quotes and the curly brackets were unnecessary. In this specific case you could have got away with just this:

```
pcphxtr01:~$ echo $TERM
xterm
pcphxtr01:~$
```

But rather than explain when you do and don't need the double quotes and when you do and don't need the braces (curly brackets) we will get into the good habit of always using them.

You *do* always need the dollar character, though. It is the signal that triggers the conversion from variable name to variable value.

```
pcphxtr01:~$ echo TERM
TERM
pcphxtr01:~$
```

Let's see the `TERM` environment variable in action. We are going to need a long piece of text and we will use

the story.txt file in the Treasure Island directory. (If your copy is still in the parent directory, Unix Intro, then move it back to Treasure Island.) We will maximise our terminal window to fill the screen and read the first screenful of treasure.txt with more:

```
pcphxtr01:Treasure Island$ more story.txt
TREASURE ISLAND
by Robert Louis Stevenson
…
     5.  THE LAST OF THE BLIND MAN  . . . . . .  36
     6.  THE CAPTAIN'S PAPERS . . . . . .  41
--More--(0%)
```

We notice that the number of lines shown matches the increased number of rows in the expanded window. To do this, more needed to be able to communicate with the terminal to learn how many rows it had. There are standard ways to do this but they hinge on the TERM environment variable identifying the type of terminal correctly.

We are going to unset the TERM environment variable with the appropriately named "unset" command.

We quit from more and run the command "unset TERM" and check with echo to see that it has no value any more. (Unset variables show up as blank values by default.)

```
pcphxtr01:Treasure Island$ unset TERM
pcphxtr01:Treasure Island$ echo "${TERM}"

pcphxtr01:Treasure Island$
```

Now we repeat the more command.

This time we see that only twenty-four lines of output are printed (including the "--more--(0%)" prompt) and that the prompt is no longer in inverse video.

```
pcphxtr01:Treasure Island$ more story.txt
TREASURE ISLAND
by Robert Louis Stevenson
…
               --So be it, and fall on!  If not,
--More--(0%)
```

The more command doesn't know how deep the terminal is so it is guessing at twenty-four lines (a common value for fixed size terminals). It also doesn't know how to generate inverse video any more. It doesn't even know if the terminal *can* produce inverse video.

Now we will see how to set the value of an environment variable. The same command sets the value of a previously unset environment variable and resets the value of an existing one.

```
pcphxtr01:Treasure Island$ export TERM=xterm
pcphxtr01:Treasure Island$ more story.txt
…
     5.  THE LAST OF THE BLIND MAN  . . . . . .  36
     6.  THE CAPTAIN'S PAPERS . . . . . .  41
--More--(0%)
```

This time we see a proper screenful again and an inverse video prompt from more.

Why "export"? The word implies (correctly) that the variable is not just for the shell itself but should be "exported to" any other commands the shell launches (such as more). The other kind of variable mentioned

above is called a “shell variable” and is used by the shell only and not passed into any of the commands launched from the shell. We will not use shell variables here.

But what are the environment variables used for? There are over a hundred set on a typical PWF Linux session! We've only seen TERM so far. We will only focus on three more.

The PATH environment variable

For this demonstration to work we will need a new terminal window. Again, this is more for the demonstration than for any fundamental property of environment variables.

Almost all the commands in Unix correspond to a file containing the instructions for that command, typically in the computer's machine code. You can find out what file a command corresponds to with the “`type`” command:

```
pcphxtr01:~$ type more
more is /bin/more
```

But how did the shell know to look in the directory “`/bin`” for a file called “`more`”?

The PATH environment variable has as its value a list of directories separated by colons. This is the list of directories the shell will go looking in.

```
pcphxtr01:~$ echo "${PATH}"
/usr/lib/mpi/gcc/openmpi/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X1
1R6/bin:/usr/games:/opt/kde3/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/opt/novel
l/iprint/bin:/opt/real/RealPlayer
pcphxtr01:~$
```

Because it takes time to search through each of these directories the shell remembers where it found each command after the first time it uses it successfully. This is a process called “hashing”. (The sort of record the shell uses to remember this information is called a “hash table”.) If we use `more` and then ask about it again we get a slightly different answer:

```
pcphxtr01:~$ type more
more is hashed (/bin/more)
pcphxtr01:~$
```

This is why we wanted a fresh terminal window; we needed an unused `more` command.

We will start yet another new terminal window and, before we try to run any command (which will get them hashed), we unset the PATH environment variable. Then we try to use `more`, or `type`:

```
pcphxtr01:~$ unset PATH
pcphxtr01:~$ type more
bash: type: more: not found
pcphxtr01:~$ more /etc/motd
bash: more: No such file or directory
pcphxtr01:~$
```

In the absence of PATH to help it search directories, the shell cannot find programs for the commands you type.

Close down the window with that broken session. It will only confuse.

It is actually a major advantage of the environment not being passed back up to the parent or across to fellow child processes that if you corrupt your environment the corruption stays localised.

The PS1 environment variable

The shell responds to other environment variables. The "PS1" environment variable controls the prompt the shell uses. The prompt we have seen so far looks like this:

```
pcphxtr01:~$
```

and contains your login id, your machine name and your current working directory. Now, we look at the PS1 environment variable:

```
smaug:~$ echo "${PS1}"
\h:\W\$
smaug:~$
```

In the PS1 environment variable the backslash is used to mark ordinary characters as having special meaning (as opposed to taking special characters and making them ordinary).

\h	The machine name, also known as the **h**ost name.
:	No backslash: this is just an ordinary character.
\W	The name of the current **w**orking directory.
\$	This is just the same as "$" if you are not the super user. It changes to "#" if you are.
␣	There is actually a trailing space on the value. It has no backslash; this is just an ordinary character.

There are many other special codes like this and a longer list is given at the end of the notes.

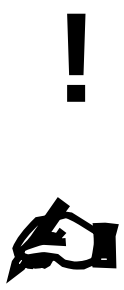
For example, "\t" gives the time:

```
pcphxtr01:~$ export PS1="\t\$ "
17:21:28$
```

The HOME environment variable

The last significant environment variable we will mention (but which we rarely, if ever, modify) is HOME. This identifies your home directory and where the cd command takes you if it is given no argument.

```
pcphxtr01:~$ echo "${HOME}"
/home/y250
pcphxtr01:~$ cd
pcphxtr01:~$ pwd
/home/y250
pcphxtr01:~$ export HOME=/tmp
pcphxtr01:~$ cd
pcphxtr01:~$ pwd
/tmp
pcphxtr01:~$
```



Close down this window too.

Exercise 21.
Exactly how many environment variables do you have set? (Consider how to connect env and wc with a pipe.)
Is the number the same for a graphical window and a text console?

[5 minutes]

Remote access to other Unix systems

Most computers live on the network. Unix systems typically offer a secure mechanism to log in to another system you have an account on. The command for this is "`ssh`" ("secure shell") and as the name "shell" suggests it gives a command line on the remote system.

All users of the PWF can use their accounts on the PWF Linux servers that can be contacted from anywhere in the world by `ssh`. This course will not attempt to describe the cryptography used but will guide you through its implications.

The first part of `ssh`'s security is to check that you are connecting to the system you think you are. In the `ssh` world every machine has a "fingerprint". This appears as a sequence of numbers (expressed base 16 just to make it look more bizarre) which can be securely checked from a workstation (like your PC) which has never made contact with it before. However, among UCS systems we arrange for the fingerprints to be known in advance so you don't get bothered by them.

Remote login between cooperating systems

We connect to the system `soup.linux.pwf.cam.ac.uk` with the command "`ssh soup.linux.pwf.cam.ac.uk`". You can safely ignore the "not in list of known hosts" warning. We attach the fingerprint to a server whose network address might change. This is the warning that the network address of this service isn't known up front.

```
pcphxtr01:~$ ssh soup.linux.pwf.cam.ac.uk
RSA host key for IP address '193.60.95.74' not in list of known hosts.

y250@soup.linux.pwf.cam.ac.uk's password:
```

We enter our PWF password as we used to log in and we will connect through to `soup`. The password will not be repeated on the screen.

```
pcphxtr01:~$ ssh soup.linux.pwf.cam.ac.uk
RSA host key for IP address '193.60.95.74' not in list of known hosts.

y250@soup.linux.pwf.cam.ac.uk's password: password

Welcome to PWF Linux 2008/2009.


If you have any problems, please email Help-Desk@ucs.cam.ac.uk.

…

soup:~$
```

Notice how the machine name in the prompt has changed.

Remote login to a new system

If you connect to a system which doesn't have fingerprint arrangements sorted out in advance you will get a challenge like this:

```
pcphxtr01:~$ ssh ent.csi.cam.ac.uk
The authenticity of host 'ent.csi.cam.ac.uk (131.111.10.87)' can't be
established.
RSA key fingerprint is a6:40:a8:56:f1:32:92:fa:c5:27:16:a2:21:1a:76:40.
Are you sure you want to continue connecting (yes/no)? no
```

(You are welcome to try to connect to `ent` but you should expect to fail if you aren't a member of the UCS.)

If you are moving away from your own Unix computer and you expect to be connecting back you need to know the fingerprint in advance. This is not a password; you are invited to write fingerprints down.

The ssh fingerprint for the public PWF Linux servers is `74:32:9b:4c:52:47:fd:ad:1b:0e:b8:a7:0f:31:3d:99`.

Once we have accepted a fingerprint on a system we will not be challenged again. If the remote system changes its fingerprint (which it should only do if it has been successfully hacked) then our login attempts will be rejected as insecure. The `ssh` command believes that the connection has been hijacked by another computer claiming to be the one you wanted and had been to before.

The `ssh` command assumes we have an account on the remote system with the same name as the account on the workstation. If this is not the case we can give an account name by preceding the machine name with "`user@`".

[5 minutes]

Exercise 22.

1. Start up a fresh terminal window.
2. Give the "`w`" command. This lists the currently logged in users. It should list just you, probably more than once.
3. Log in to the system `soup.linux.pwf.cam.ac.uk`.
4. Run the `w` command there too. You should see more users.
5. Log out with [Ctrl]+[D].

File transfer

In addition to logging on to remote systems we may also want to transfer files to or from them. In addition to `ssh` ("**s**ecure **sh**ell") there is a related program "`scp`" ("**s**ecure **c**o**p**y"). This behaves in exactly the same way as `cp` except that one of the target or destination is actually a reference to a remote system. There is also an "`sftp`" program which is an interactive file transfer program that lets you navigate a the far end.

Because we share home directories across all the PWF Linux systems we will be copying in and out of `/tmp` on the remote system.

Fetching files and directories

There is a file on the machine `smaug.linux.pwf.cam.ac.uk` called "`/tmp/fetchme.txt`". To fetch it into the current working directory we could run the following command:

```
pcphxtr01:~$ scp smaug.linux.pwf.cam.ac.uk:/tmp/fetchme.txt fetchme.txt
y250@smaug.linux.pwf.cam.ac.uk's password:
fetchme.txt 100%   86     0.1KB/s   00:00

pcphxtr01:~$ ls -l fetchme.txt
-rw-r--r-- 1 y250 y250  86 2009-09-01 18:12 fetchme.txt

pcphxtr01:~$
```

Note how we define a file on a remote computer: `machine_name:file_path` with a colon separating the two components.

Just as with `ssh`, `scp` assumes you have the same login id on the remote system as on the local one. If you have a different name on the remote system then you specify with as you did for `ssh`, with "`user@`" before the `machine_name:file_path` element. So if your remote user id was "`bob`" you would run the command:

```
pcphxtr01:~$ scp bob@smaug.linux.pwf.cam.ac.uk:/tmp/fetchme.txt fetchme.txt
```

If we are happy for the file name to remain the same there is a trick to save on the typing. We can say "copy it into this directory" in which case the copy will leave it with the same file name:

```
pcphxtr01:~$ scp bob@smaug.linux.pwf.cam.ac.uk:/tmp/fetchme.txt .
```

Recall that "`.`" means "the current directory".

We can rename the file as we copy it simply by giving a different name as the second argument:

```
pcphxtr01:~$ scp smaug.linux.pwf.cam.ac.uk:/tmp/fetchme.txt newdata.txt
```

To fetch a directory and everything in it, we must specify a recursive copy, just as we had to do to copy a directory within the system. Unfortunately, the `scp` program hasn't moved to the modern upper case "`-R`" option for recursion so we have to use the lower case "`-r`":

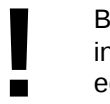
```
pcphxtr01:~$ scp -r smaug.linux.pwf.cam.ac.uk:/tmp/fetchable .
```

Sending files and directories

To send data rather than to fetch it we simply use the same syntax for specifying a remote file but on the second argument rather than the first.

To copy a file from the current working directory to a remote location we specify it like this:

```
pcphxtr01:~$ scp newdata.txt smaug.linux.pwf.cam.ac.uk:/tmp/y250_data.txt
```



Because we are all using `/tmp` on the remote system, please put your user id into the file names you create there. This way your files won't collide with the equivalent files from other people doing the course.

Interactive file transfer

So far we have been able to send or fetch files but in both cases we need to know the remote location and we have had no opportunity to send some files and fetch others. The second file transfer program, "`sftp`", will allow us to do that but stops us transferring directories recursively (for no readily apparent reason).

The `sftp` program is interactive, so rather than issue a single command, as with `scp`, you launch the `sftp` program to connect to a remote system and then issue a series of instructions within the `sftp` program telling it to change directories at either end, to fetch or send files, to list directories at either end etc.

We launch `sftp` by simply identifying the remote computer. Notice that the prompt changes to indicate that we are now inside the `sftp` program rather than the shell. To draw out a particular point we will move into the `Unix Intro` directory before launching the program.

```
pcphxtr01:~$ cd Unix\ Intro

pcphxtr01:Unix Intro$ sftp linux.pwf.cam.ac.uk
Connecting to linux.pwf.cam.ac.uk...
 RSA host key for IP address '193.60.95.74' not in list of known hosts.
y250@linux.pwf.cam.ac.uk's password:

sftp>
```

If we had a different account name then we would have said

```
pcphxtr01:Unix Intro$ sftp bob@linux.pwf.cam.ac.uk
```

Notice how we do not specify a location. This is the first difference from `scp`. We always start in our home directory at the far end.

```
sftp> pwd
Remote working directory: /home/y250

sftp>
```

The Unix command `pwd` in the `sftp` program gives information about the *remote* end. To get the local working directory, use the `sftp`-only command "`lpwd`" ("**l**ocal **pwd**"):

```
sftp> lpwd
Local working directory: /home/y250/Unix Intro
sftp>
```

This pattern is repeated for several Unix commands. The original Unix command inside sftp works on the remote system and the same command prefixed with an “l” (for “local”) works on the local system. For example to change directory at the remote end we use “cd” and to change directory locally we use “lcd”:

```
sftp> cd /tmp
sftp> lcd /home/y250
sftp> pwd
Remote working directory: /tmp
sftp> lpwd
Local working directory: /home/y250
sftp>
```

The `ls` and `lls` commands list files at either end and support the `-l` and `-a` options.

To actually transfer files we use two commands within `sftp`: “`get`” and “`put`”.

```
sftp> lls
appscfg.pwf  Desktop  My Music     My Pictures  My Video  newdata.txt  Unix
Intro
sftp> put newdata.txt y250_example.txt
Uploading newdata.txt to /tmp/y250_example.txt
newdata.txt                                   100%   86     0.0KB/s   00:00
sftp> get fetchme.txt another.txt
Fetching /tmp/fetchme.txt to another.txt
/tmp/fetchme.txt                              100%   86     0.1KB/s   00:00
sftp>
```

To exit the `sftp` program, either enter [Ctrl]+[D] or the command “`quit`”. Notice how you return to the shell as you left it. The internal `lcd` command only affected the session within `sftp`.

```
sftp> quit
pcphxtr01:Unix Intro$
```

A full set of `sftp` commands is given by the `sftp` command “`help`”. A set of the most useful ones is given in the appendices to these notes.

```
sftp> help
Available commands:
cd path                       Change remote directory to 'path'
lcd path                      Change local directory to 'path'
…
?                             Synonym for help
sftp>
```

Exercise 23.

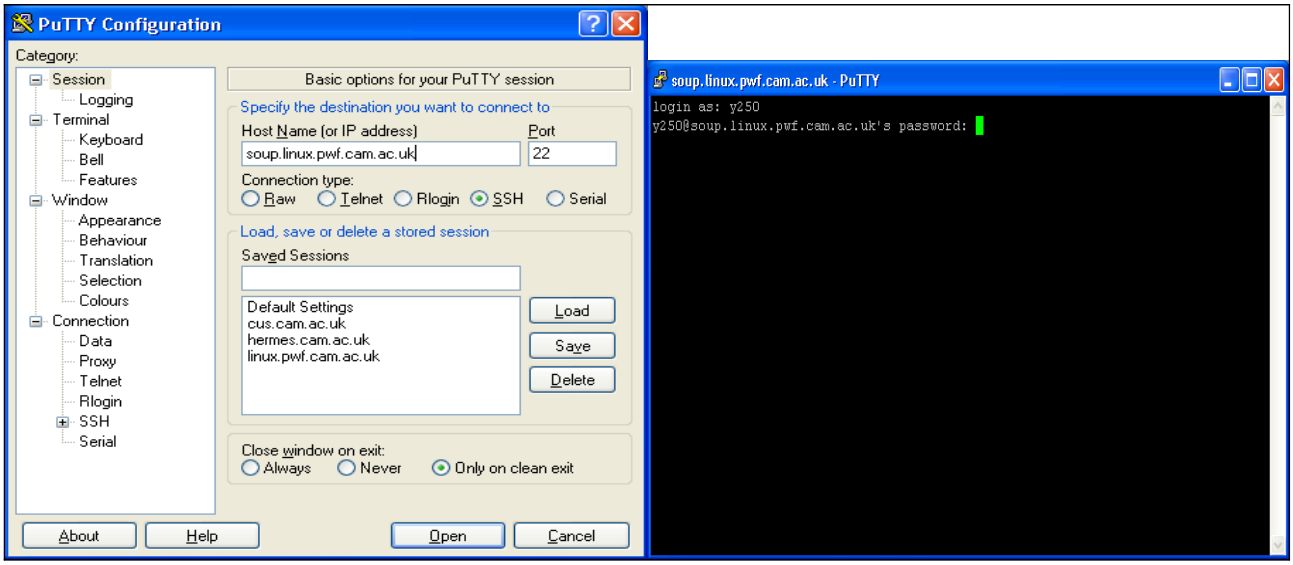
[10 minutes]

1. Create a file named after your userid containing a sentence about your name:
   `echo "My name is Julian." > y250.txt`
   (You would use different words for “Julian” and “y250”.)
2. Use `scp` to transfer the file `y250.txt` into the `/tmp` directory on a machine `soup.linux.pwf.cam.ac.uk`. Your current login and password will work for that machine.
3. Use `sftp` to connect to `soup.linux.pwf.cam.ac.uk`.
4. Move to `/tmp` on the remote system.
5. List the set of these files present:
   `ls y???.txt`
6. Fetch a different one from your own, `y251.txt` say.
7. Disconnect from `soup.linux.pwf.cam.ac.uk`.
8. Read the file you have just fetched:
   `more y251.txt`
   `My name is Fred.`
9. Append a line of your own:
   `echo "Hello, Fred." >> y251.txt`
10. Try (and fail) to put it back into `/tmp` on `soup` with `scp`.
11. Try (and succeed) to put it back with a different name, `y251-fred.txt`, still using `scp`.

Just for interest

This is a Unix course so we don't talk about Windows much. However, now that we have seen how to connect to a remote Unix command line from a Unix box with `ssh`[9], we might ask how to connect from a Windows box.

The best Windows application for this purpose is called “`putty`”:



The `putty` application can be downloaded free from `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`, courtesy of its author, Simon Tatham.

If all you want to do is transfer files between your Windows machine and a Linux server, you may find WinSCP useful: `http://winscp.net/eng/docs/introduction`.

9 Mac OS X counts as a Unix box; the `ssh` command is available in its `Terminal` application.

Trivial shell scripts

Finally, one of the advantages of the command line is that a record or "script" can be made of the commands issued and that script can then be kept for later reuse or given to other people for them to use. It's much easier to pass to a colleague a file of explicit commands than to describe in hand-waving text how to navigate a GUI application.

We will open a plain text editor from the menu system with Applications→Word and Text Processing→gedit. We will enter a few trivial shell commands into it and save it as a file `commands.sh` in our home directory. The following is what should go in the file, not what you should run!

```
pwd
date
ls -l
cd "Unix Intro"
ls -l
```

Now we launch a terminal window. This will start in our home directory where we have just created the file.

Recall that the name of our shell is "`bash`". That's also the command to run another one of these command line interpreters.

We run the command "`bash commands.sh`":

```
pcphxtr01:~$ bash commands.sh
/home/y250
Thu Nov 26 20:05:37 GMT 2009
total 7
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:59 another.txt
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 Appscfg.PWF
-rw-r--r-- 1 y250 y250  37 2009-11-26 20:05 commands.sh
drwxr-xr-x 1 y250 y250 512 2009-11-26 19:21 Desktop
drwxr-xr-x 1 y250 y250 512 2009-11-26 19:37 fetchable
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:35 fetchme.txt
drwxr-x--- 1 y250 y250 512 2009-11-25 13:08 Library
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Music
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Pictures
drwxr-x--- 1 y250 y250 512 2009-11-25 19:00 My Video
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:36 newdata.txt
drwxr-xr-x 1 y250 y250 512 2009-11-26 19:17 Unix Intro
-rw-r--r-- 1 y250 y250  16 2009-11-26 20:00 y250.txt
-rw-r--r-- 1 y250 y250  28 2009-11-26 20:02 y551.txt
total 6
-rw-r--r-- 1 y250 y250    0 2009-11-25 20:03 fubar.txt
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:37 Fun
-rw-r--r-- 1 y250 y250 3693 2009-11-25 19:39 lorem.txt
drwxr-xr-x 1 y250 y250  512 2009-11-25 19:34 Play
drwxr-xr-x 1 y250 y250  512 2009-11-26 19:17 Treasure Island
drwxr-xr-x 1 y250 y250  512 2009-11-26 19:08 Work

pcphxtr01:~$
```

This isn't quite a "command". The command was "`bash`" and we told it what file of commands to run. Ideally we would like to just be able to give the command "`commands.sh`". We will get there in two stages.

First, we will do away with having to give the command `bash`.

Recall that Unix has three types of permission of a file, labelled with "`rwx`": a file can be **r**eadable, **w**riteable, and e**x**ecutable. At the moment this file is only readable and writeable by us:

```
pcphxtr01:~$ ls -l commands.sh
-rw-r--r-- 1 y250 y250 37 2009-04-28 23:32 commands.sh

pcphxtr01:~$
```

We will make it executable by changing its permissions (also known as its "mode") with the "chmod" command ("**ch**ange **mod**e"). The syntax of this command is an extreme example of Unix complex conciseness.

```
pcphxtr01:~$ chmod a+x commands.sh

pcphxtr01:~$ ls -l commands.sh
-rwxr-xr-x 1 y250 y250 37 2009-04-28 23:32 commands.sh

pcphxtr01:~$
```

The "a+x" means "for **a**ll classes of user (owner, group member, other) add (**+**) the e**x**ecute permission".

The commands.sh file is now ready to be executed, but the shell will never find it because your home directory is not on your PATH and that's where the shell looks. We can tell it where to look by giving the exact file name for the command so that it doesn't have to go looking.

```
pcphxtr01:~$ /home/y250/commands.sh
/home/y250
Thu Nov 26 20:07:35 GMT 2009
total 7
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:59 another.txt
…
drwxr-xr-x 1 y250 y250  512 2009-11-26 19:08 Work

pcphxtr01:~$
```

It's awkward having to keep track of the directory the file it in. Recall that the directory "." means "this directory". We typically use that to tell the shell to run a command from a file this directory:

```
pcphxtr01:~$ ./commands.sh
/home/y250
Thu Nov 26 20:09:22 GMT 2009
total 7
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:59 another.txt
…
drwxr-xr-x 1 y250 y250  512 2009-11-26 19:08 Work

pcphxtr01:~$
```

Finally we will get rid of that annoying leading "/home/y250/" or "./". We need to put the executable file somewhere on the PATH.

There is a traditional place for personal, "quickie" shell scripts to go. That is in a directory called "bin" in your home directory[10]. The way PWF Linux is configured (actually the way the openSUSE system it is based on is configured) automatically adds that directory to your PATH if the directory exists when you first log in. Unfortunately it didn't exist.

We will create a directory called "bin" in our home directory and move the commands.sh file into that directory.

10 "bin" stands for "**bin**aries", not "trash can". Once upon a time there were no scripts and every program had to be compiled into a binary executable file.

Exercise 24.

[10 minutes]

1. Create a subdirectory of your home directory called "`bin`".
   `mkdir ${HOME}/bin`
2. Move your `commands.sh` script into the `bin` directory.
   `mv commands.sh ${HOME}/bin`
3. Check the current value of the `PATH` environment variable. Record the first couple of directories it lists.
   `echo "${PATH}"`
4. Run the command "`commands.sh`" (without any leading "`./`"). Observe that the shell cannot find the command.
   `commands.sh`
5. Log out (System → Log out) and back in again. The course lecturer will give you the password for your account. (n.b. You need to log out all the way; you can't just close the window and open another one.)
6. Check the new value of the `PATH` environment variable. Record the first couple of directories it lists. Your `bin` directory should be a the front of the list.
7. Run the command "`commands.sh`" (without any leading "`./`"). This time it should work.

```
pcphxtr01:~$ commands.sh
/home/y250
Thu Nov 26 20:11:17 GMT 2009
total 7
-rw-r--r-- 1 y250 y250  86 2009-11-26 19:59 another.txt
…
drwxr-xr-x 1 y250 y250  512 2009-11-26 19:08 Work
pcphxtr01:~$
```

Congratulations! You have just written your first Unix shell script.

This logging in and logging out trick is specific to PWF Linux and OpenSUSE Linux. It's also specific to a directory called `bin` in your home directory. This won't work for any other directory and it quite possibly won't work for other flavours of Linux.
(Many of them have `${HOME}/bin` on the `PATH` regardless of whether the directory exists so there's no need for this trick.)
Now that you have a `${HOME}/bin` directory you do not need to repeat the "logging in again" trick. It's done for good. Each time you log in this directory will be added to your `PATH` automatically.

Real shell scripts can have "comments". These are sections of the script that have no effect on what the script does and exist purely to aid the (human) reader to understand what the script is about. They can be used, too, to quote authors and contact details etc.

Comments are defined by a "#" ("hash") character. Everything from the hash to the end of the line is treated as a comment and ignored by the shell:

```
# My first shell script!

pwd                 # Print the current directory
date                # Print the date and time
ls -l               # List the files
cd "Unix Intro"     # Change directory
ls -l               # List the files in the new directory
```

There is one last change we ought to make for a proper shell script. Scripts can be written in various languages, not just shell. If there's no special instructions then the operating system assumes that it is in shell. This is what happened with our script. However it is traditional to be explicit.

The "special instructions" lie on the first line. It starts with "`#!`" followed by the full path to the interpreter whose language the script is written in. So, if we wanted to be explicit about the shell we would start with

"`#!/bin/bash`" as the first line. Adding this line is the final change we will make.

```
#!/bin/bash

# My first shell script!

pwd                   # Print the current directory
date                  # Print the date and time
ls -l                 # List the files
cd "Unix Intro"       # Change directory
ls -l                 # List the files in the new directory
```

Note that because "#" is the comment character the first line counts as a comment as far as the shell is concerned and won't affect the operation of the script itself.

Incidentally, any scripting language that uses "#" as its comment character can have scripts defined this way. So Python (file `/usr/bin/python`) has scripts starting with "`#!/usr/bin/python`", Perl has "`#!/usr/bin/perl`", etc.

Now, obviously simply listing random instructions like this isn't a good way to write shell scripts. However, the shell has a far broader syntax than we have seen here and the course "Simple Shell Scripting for Scientists" takes it much further.

[15 minutes]

Exercise 25.
Recall the commands `echo`, `who`, and `date` (with its formatting strings).

Create a shell script called "`now`" that will work like this:

```
pcphxtr01:~$ now

The time is: 17:04
The date is: April 29
These people are logged on:
rjd4     pts/0     16:12    1.00s  0.10s  0.00s now
rjd4     pts/1     16:07   42.00s  0.10s  0.10s -bash

pcphxtr01:~$
```

Appendices

Command summary

This is a very quick summary of all the command line commands we cover in this course.

Command	Example	Action
`bash`	`bash commands.sh`	This is the name of the command line interpreter we've been using all along!
`bc`		Command line calculator.
`bg`		Background a command stopped with [Ctrl]+[Z].
`cat`	`cat abc.txt def.txt`	Concatenate one or more files.
`cd`	`cd ../Work`	Change directory.
`chmod`	`chmod a+x commands.sh`	Change the mode (permissions) of a file. Use "`chmod a+x`" on a shell script.
`clear`		Clear screen. (Better to use [Ctrl]+[L].)
`cp`	`cp island.jpg map.jpg`	Copy a file. (Use the "`-R`" option to recursively copy a directory.)
`date`	`date +"%H:%k:%M"`	Give the date and time. Output can be formatted.
`echo`	`echo *.txt`	Repeat the command line arguments passed to it.
`env`		Print out the set of environment variables.
`exit`		Exit the shell if you don't want to use [Ctrl]+[D].
`export`	`export TERM=xterm`	Set the value of an environment variable.
`fg`		Foreground a command stopped with [Ctrl]+[Z].
`grep`	`grep rum story.txt`	Search a file for a text string.
`history`		List the previous command lines.
`jobs`		List all the backgrounded jobs currently running.
`less`	`less lorem.txt`	Page through a file, a screenful at a time. A recent version of `more`.
`ls`	`ls Work`	List the contents of a directory.
`mkdir`	`mkdir Fun`	Make a directory.
`more`	`more lorem.txt`	Page through a file, a screenful at a time.
`mv`	`mv island.jpg ../map.jpg`	Move (rename) a file or directory.
`rm`	`rm nonsense.txt`	Remove a file. (Use the "`-R`" option to recursively remove a directory.)
`rmdir`	`rmdir Fun`	Remove an *empty* directory.
`scp`		Remotely copy a file or directory tree to or from a remote system.
`sftp`		Interactively transfer files to or from a remote system.
`ssh`		Establish a connection to a remote system to run commands on it.
`touch`	`touch newfile.txt`	Create an empty file.
`type`	`type more`	Find where a command comes from.
`unset`	`unset TERM`[11]	Unset a variable.
`w`		Show who is logged on, what they are doing and how busy the system is.
`wc`	`wc lorem.txt`	Count the lines, words and characters in a file.
`who`		Show who is logged in.

We also met a number of graphical applications:

Command	Default for these file types	
`eog`	GIF, JPEG, PNG, SVG	"Eye of Gnome" graphics viewer
`evince`	PDF	Document viewer
`firefox`	HTML	Firefox web browser
`gedit`	TXT	Text editor
`xeyes`		Silly test application to track the cursor

11 "`unset TERM`": *Don't do this!*

Date formats

Year		
`%C`	20	Century
`%Y`	2009	Four digit year
`%y`	09	Two digit year
Month		
`%b`	Apr	Abbreviated month name
`%B`	April	Full month name
`%m`	04	Two digit numerical month
Day		
`%j`	118	Day of year (1…366)
`%d`	28	Two digit day of month
`%a`	Tue	Abbreviated day of week
`%A`	Tuesday	Full day of week
`%u`	2	Numerical day of week (1…7, 1=Monday)
`%w`	2	Numerical day of week (0…6, 0=Sunday)
Hour		
`%H`	21	Hour of the day (0…23)
`%I`	09	Hour of the day (1…12)
Minute		
`%M`	07	Minute of the hour
Second		
`%S`	23	Second of the minute
`%s`	1240949377	Seconds since 1970-01-01 00:00:00 GMT
Useful		
`%n`		New line
`%t`		Tab

There are two useful modifiers. If "`%M`" were to give "`07`" then "`%_M`" would give "␣7" and "`%-M`" would give "7".

Globbing

`*` Any string of characters, including the empty string.

`thing*`	matches: **thing**.txt, **thing**, **thing**s does not match: thin, think
`th*ing`	matches: **th**ink**ing**, **thing**, **th**e␣shin**ing** does not match: ting, think
`th*in*`	matches: **thin**g.txt, **thin**g, **thin**gs, **th**ank**in**g, **th**e\ shin**in**g, **thin**, **thin**ks

`?` Any single character.

`thing.???`	matches: **thing.**txt, **thing.**dat, **thing.**jpg does not match: thing.jpeg, thing

`[…]` Any single character from the set in the brackets.

`thing.t[xyz]t`	matches: **thing.tx**t does not match: thing.tot

`[^…]` Any single character not in the set.

`thing.t[^xyz]t`	matches: **thing.to**t does not match: thing.txt

The character class globbing expressions `[:…:]` all appear inside the standard `[…]` brackets of the glob, so we get doubly nested square brackets. So "`[N[:digit:]]`" matches "N" or "any one digit".

`[:alnum:]`	Any alphabetic character (upper or lower case) or any digit.
`[:alpha:]`	Any alphabetic character (upper or lower case).
`[:blank:]`	Any horizontal white space (space or tab, essentially).
`[:digit:]`	Any of the ten digits.
`[:lower:]`	Any lower case alphabetic character.
`[:upper:]`	Any upper case alphabetic character.

PS1 codes

These codes can be placed inside the PS1 environment variable. There are more, but these are the useful ones.

Time codes

`\D{format}`		The date where the format is given by a format string using the %-codes listed above.
`\d`	`Tue 10 Sep`	The date in this specific format
`\t`	`17:28:26`	24-hour time, with seconds
`\T`	`05:28:26`	12-hour time, with seconds
`\A`	`17:28`	24-hour time
`\@`	`05:28`	12-hour time

Other codes

`\h`	`pcphxtr01`	Short machine name
`\H`	`pcphxtr01.pwf.cam.ac.uk`	Full machine name
`\l`	`2`	Terminal number, `/dev/pts/2`→`2`
`\u`	`y250`	User name
`\W`	`Work`	Name of the current working directory
`\w`	`/home/y250/Unix Intro/Work`	Absolute path of the current working directory

Command line cursor control

There are often two ways to do these operations. One way avoids the use of the cursor keys but requires the memorisation of some other letters.

[Ctrl]+[F]	[→]	Move right one character.
[Ctrl]+[B]	[←]	Move left one character.
[Alt]+[F]	[Ctrl]+[→]	Move right one word.
[Alt]+[B]	[Ctrl]+[←]	Move left one word.
[Ctrl]+[A]	[Home]	Move to start of line
[Ctrl]+[E]	[End]	Move to end of line.
[Ctrl]+[W]		Remove the word to the left of the cursor.
[Ctrl]+[K]		Remove the line to the right of the cursor.
[Ctrl]+[U]		Remove the line to the left of the cursor.
[Ctrl]+[P]	[↑]	Go back one line in history.
[Ctrl]+[N]	[↓]	Go forwards one line in history.

sftp commands

Any Unix command can be run on the local system by preceding it with a "`!`". Where there is no `l-` local version of a command we show the `!-` version. The `!-` version always works, except for "`lcd`".

Remote	Local	
`cd`	`lcd`	Change directory
`ls`	`lls`	List directory contents
`pwd`	`lpwd`	Print working directory
`mkdir`	`lmkdir`	Make a directory
`rmdir`	`!rmdir`	Remove empty directory
`rm`	`!rm`	Remove file
`get remote_name`		Fetch a remote file, keeping its name.
`get remote_name local_name`		Fetch a remote file, changing its name.
`put local_name`		Put a file onto the remote system, keeping its name.
`put local_name remote_name`		Put a file onto the remote system, changing its name.
`help`		Show the complete set of `sftp` commands.
`quit`		Quit `sftp`.

Environment variables

HOME

Specifies your home directory.

You should never change this value.

```
pcphxtr01:~$ echo "${HOME}"
/home/y220
```

PATH

Specifies the list of directories where the operating system goes looking for executable files to run the commands you issue.

You should only ever add to this value. Removing directories from it that are provided by the system may break some system facilities. On PWF Linux and OpenSUSE Linux your `${HOME}/bin` directory is added to your `PATH` by the system if and only if it exists when you log in.

```
pcphxtr01:~$ echo "${PATH}"
/home/y220/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/g
ames:/opt/kde3/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/opt/novell/iprint/bin:/
opt/real/RealPlayer
```

PS1

Specifies your shell prompt.

Characters preceded by a backslash, “\”, are translated into system data. Ordinary characters are used unchanged. A trailing space is often a good idea.

`\h`	The machine name, also known as the **h**ost name.
`\t`	The **t**ime
`\W`	The name of the current **w**orking directory.
`\$`	This is just the same as “$” if you are not the super user. It changes to “#” if you are.

```
pcphxtr01:~$ echo "${PS1}"
\h:\W\$
```

TERM

Specifies your terminal type.

This should be set by the system and you should not need to change it. Commands that need to know the parameters of your terminal will fail if this is unset or incorrectly set. (e.g. `more` needs to know how many rows your screen has.)

```
pcphxtr01:~$ echo "${TERM}"
xterm
```